Albert Rubio (Ed.)

Termination

6th International Workshop, WST 2003
Valencia, Spain, June 13-14, 2003
Proceedings

Volume Editor

Albert Rubio
Universitat Politècnica de Catalunya
Barcelona, Spain
Email: rubio@lsi.upc.es

Proceedings of the 6th International Workshop on Termination, WST'03
Valencia, Spain, June 13-14, 2003

ISBN:

Depósito Legal:

Impreso en España.

Technical Report DSIC-II/15/03,
http://www.dsic.upv.es
Departamento de Sistemas Informáticos y Computación,
Universidad Politécnica de Valencia, 2003.

Preface

This report contains the proceedings of the *6th International Workshop on Termination* (WST'03). It was held June 13-14, 2003 in Valencia, Spain, as part of RDP, the *Federated Conference on Rewriting, Deduction and Programming*, together with the Int. Conference on Rewriting Techniques and Applications (RTA'03), the Int. Conference on Typed Lambda Calculi and Applications (TLCA'03), the Int. Workshop on First-order Theorem Proving (FTP'03), the annual meeting of the IFIP Working Group 1.6 on Term Rewriting, the Int. Workshop on Rule-Based Programming (RULE'03), the Int. Workshop on Unification (UNIF'03), the Int. Workshop on Functional and (Constraint) Logic Programming (WFLP'03), and the Int. Workshop on Reduction Strategies in Rewriting and Programming (WRS'03).

This workshop delves into all aspects of termination of processes. Though, the halting of computer programs, for example, is undecidable, methods of establishing termination play a fundamental role in many applications and the challenges are both practical and theoretical. From a practical point of view, proving termination is a central problem in software development and formal methods for termination analysis are essential for program verification. From a theoretical point of view, termination is central in mathematical logic and ordinal theory.

Previous WST workshops were held in St. Andrews (1993), La Bresse (1995), Ede (1997), Dagstuhl (1999) and Utrecht (2001).

Apart from the abstracts presentation sessions, for the first time, this year the Workshop on Termination has promoted an Exhibition/Competition of Termination Proof systems. This report includes the 15 abstracts and the 9 system descriptions presented at the workshop.

May 2003 Albert Rubio

Program Commitee

Thomas Arts	(Gothenburg)
Michael Codish	(Beer-Sheva)
Juergen Giesl	(Aachen)
Jean Goubault-Larrecq	(Cachan)
Deepak Kapur	(Albuquerque)
Delia Kesner	(Paris)
Albert Rubio	(Barcelona, Chair)
Danny De Schreye	(Leuven)
Andreas Weiermann	(Muenster and Utrecht)

Table of Contents

Abstracts:

System Descriptions:

One Loop at a Time

Michael Codish Samir Genaim Maurice Bruynooghe
John Gallagher Wim Vanhoof

Extended Abstract

Classic techniques for proving termination require the identification of a measure mapping program states to the elements of a well founded domain and to show that this measure decreases with each iteration of a loop in the program. This is a *global* termination condition — there is a single measure which must be shown to decrease over all of the loops in the program. In this abstract we look at systems based on *local* termination conditions which allow the involvement of different well founded domains and termination measures for different loops in the program. We illustrate the practical advantages of applying local criteria in automated termination proving systems and demonstrate how Ramsey's Theorem clarifies their formal justification.

Consider the following Prolog program defining the Ackermann function. There are three recursive calls in the program giving rise to loops in the executions of the program.

```
ackermann(0,N,s(N)).
ackermann(s(M),0,Res) :-
  ackermann(M,s(0),Res).

ackermann(s(M),s(N),Res) :-
  ackermann(s(M),N,Res1),
  ackermann(M,Res1,Res).
```

It is straightforward to observe for each recursive call a suitable measure to guarantee the termination of the corresponding loop. In the first and third recursive calls, the size of the first argument decreases from s(M) to M and in the second recursive call from s(N) to N. However, this type of argument on its own does not guarantee a proof of termination. The classic approach requires to find a single global measure such that the size decreases for each of the three recursive calls. For the above example the lexicographic order over the tuple consisting of the sizes of first and second argument will do. Moreover, as demonstrated by the following program, considering individual "syntactic" loops in this way does not always lead to a correct argument for termination.

```
p(s(X),Y) :- p(X,s(Y)).
p(X,s(Y)) :- p(s(X),Y).
p(0,0).
```

Here, it is easy to observe for each of the two recursive calls, a suitable measure to guarantee the termination of the corresponding loop. Yet the program does not terminate as execution can oscillate indefinitely between the two recursive clauses.

The approach based on finding termination measures for individual loops basically requires us to show that all loops terminate. But care must be taken to consider also

loops obtained by composition of others. As the previous example shows, it does not suffice to consider the individual recursive calls in a program. As a loop, we consider any pair of subsequent calls to the same predicate. In the example, there are – apart from the loops introduced by the direct recursive calls – additional loops such as those indicated by the first and last atoms in: `p(s(s(X)),Y)` $\leadsto$ `p(s(X),Y)` $\leadsto$ `p(X,Y)` and `p(s(X),Y)` $\leadsto$ `p(X,s(Y))` $\leadsto$ `p(s(X),Y)`. It is the latter which represents the potential for nonterminating execution.

Basing termination on individual loops has two main advantages: First, local termination measures are simpler. They are easier to identify, both by the human user as well as when aiming for automated proofs. Moreover, there are many cases in which proofs based on local measures involve applying linear functions while corresponding global measures involve nonlinear functions or lexicographic orders. As a consequence, analyzers implemented to use linear techniques go farther when based on local instead of global termination conditions. Second, when programs contain termination bugs, the local approach indicates more clearly which loops are problematic.

In the logic programming context, a termination test based on local measures is first introduced in [6]. This work provides the basis for the Termilog analyzer described in [4] where size information between subsequent calls is represented by graphs. The nodes in such a graph represent program arguments and edges represent relations between their sizes. A similar approach is considered in the functional setting as presented in [3]. Codish and Taboch continue this line of work in [2] providing a formal semantic basis which makes loops observable and an implementation based on linear constraints on size variables instead of on graphs. The resulting analyzer is TerminWeb [7].

This abstract contributes a simple and formal proof of correctness for the analysis of termination based on local measures. We focus on the semantic based approach as described in [2]. Here, the semantic objects are binary clauses. A binary clause $h \leftarrow b$ indicates that a call to h will lead eventually to a call to b. A loop is a binary clause in which the head and the body specify the same predicate in the program. By definition, the binary clauses in the semantics of a program are closed under composition (or unfolding). This means that each loop, including those that consist of a composition of recursive calls, is represented by a single binary clause in the semantics of the program

In the logic programming setting *norms* are functions which determine the size of data-objects, or terms. Program states, or atoms, are measured by so called, *level mappings*. The user typically specifies the norm to be used and then the system searches for level-mappings which guarantee termination. Examples of norms are the *term-size* norm which measures the number of nodes in the tree representing a term, and the *list-length* norm measuring the length of a list structure.

The TerminWeb analyzer provides an approximation to the set of loops in a program. This is a finite set of "abstract" binary clauses describing size and instantiation information. Let us focus here, only on the size information. Abstract binary clauses are of the form $h \leftarrow \mu, b$ where μ is a linear constraint on the sizes of the terms bound to the arguments of the atoms in h and in b. For example, choosing the term-size norm, the following is a description, obtained using TerminWeb, of the loops in any execution of the Ackermann program:

```
ackermann(A,B,C) :- [A=D,E<B], ackermann(D,E,F).
ackermann(A,B,C) :- [D<A], ackermann(D,E,F).
```

The first abstract binary clause represents all concrete loops in which the sizes of the first and second arguments are stable and decreasing respectively. The second abstract binary clause represents all concrete loops in which the size of the first argument is decreasing.

The main observation in our approach is that non-termination of a program implies "non-termination" of a single abstract binary clause. Thus a proof of termination is obtained by showing termination for each individual recursive abstract binary clause. This is a major strength of the technique as it enables one to apply different measure functions to argue the termination of the different loops in the program.

The following proposition, a rephrasing of the proposition in [2], captures the property that allows termination to be established by considering the termination of the individual loops, as represented by the binary clauses. We note that the proof given in [2] does not cover the general case where different measures are applied to different loops.

Proposition: Let P be a logic program, $\left|\cdot\right|_a$ a norm function and G^a an initial query pattern specifying the arguments in the initial query which are rigid[1] with respect to the given norm. Let $unf_a(P)$ and $calls_a(P,G^a)$ denote the corresponding abstract binary clauses and abstract calls with respect to this norm. If for each abstract call pattern $p(\bar{t})$ in $calls_a(P,G^a)$ and matching recursive abstract binary clause of the form $p(\bar{x}) \leftarrow \mu, p(\bar{y})$ in $unf_a(P)$ there exists a level mapping f such that $p(\bar{t})$ is rigid with respect to f and $\mu \models f(p(\bar{x})) > f(p(\bar{y}))$ then P terminates for all initial queries described by G^a.

We present here a proof of this proposition based on Ramsey's Theorem [5].

Theorem: [Ramsey's Theorem] Let $A = \{\langle a,b\rangle \mid a,b \in \mathbb{N} \textit{ and } a < b\}$, L be a finite set of colors and let $F : A \mapsto L$ be a function associating the elements of A with colors from L. Then, there is a color $f \in L$ and an infinite set $X \subseteq \mathbb{N}$ such that $F(\langle a,b\rangle) = f$ for each $a,b \in X$ for which $a < b$.

Proof for the proposition: Given the premise of the proposition we show that the assumption that the program does not terminate leads to a contradiction. Let L be the set of level mappings applied in the premise. The set L must be finite as one level mapping is chosen for each of a finite number of abstract recursive binary clauses. If the program does not terminate then there must exist an infinite sequence of calls from $calls_a(P,G^a)$ which arise in a derivation of the program. Moreover, there must be a subsequence of calls to the same predicate with the same call pattern. We denote this sequence as $C \equiv p_1 \rightsquigarrow p_2 \rightsquigarrow \cdots \rightsquigarrow p_m \rightsquigarrow \cdots$. Each pair of calls p_i and p_j in C such that $i < j$ corresponds to some abstract recursive binary clause and is hence associated with a level mapping $f \in L$ satisfying $f(p_i) > f(p_j)$.

Adopting the notation from Ramsey's Theorem, let us define a function $F : A \mapsto L$ which maps each i,j such that $i < j$ with precisely that level mapping $f \in L$ associated with p_i and p_j. Then, according to the theorem there exists a level map-

[1] A term is rigid with respect to a given norm if its size does not change under instantiation.

ping $f \in L$ and an infinite set $X \subseteq \mathbb{N}$ such that $F(\langle i, j \rangle) = f$ for each $i, j \in X$ such that $i < j$. This means that there exists an infinite subsequence C' of C of the form $p_{i_1} \leadsto p_{i_2} \leadsto \cdots \leadsto p_{i_k} \leadsto \cdots$ involving precisely the calls $\{p_i \mid i \in X\}$ and implying that $f(p_{i_k}) > f(p_{i_{k+1}})$ for each $k \in \mathbb{N}$. But since all of these calls are rigid with respect to the level mapping f this chain cannot be infinite. □

Termination analyses based on different level mappings for different loops turns out to be quite powerful. In [1] we demonstrate how termination of different loops can be shown using different norm functions and not only using different level mappings. We suggest that a useful heuristic in termination analysis is to let the system determine level mappings for different loops combining one norm for each data-type defined in the program.

Acknowledgement: Thanks to Uri Abraham who pointed us to Ramsey's Theorem.

References

[1] Maurice Bruynooghe, Michael Codish, John Gallagher, Samir Genaim, and Wim Vanhoof. Termination analysis through combination of type based norms. Technical report, Dept. of Computer Science, Ben-Gurion University, 2003.

[2] M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.

[3] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. Proceedings of POPL'01.

[4] N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 63–77, Leuven, Belgium, 1997. The MIT Press.

[5] F.P Ramsey. On a problem of formal logic. *Proc. London Math. Society*, 30:264–286, 1930.

[6] Y. Sagiv. A termination test for logic programs. In V. Saraswat and K. Ueda, editors, *Proc. 1991 Int. Logic Programming Symposium*, 519–532. MIT Press, 1991.

[7] Cohavit Taboch, Samir Genaim, and Michael Codish. Terminweb: A semantic based termination analyser for logic programs, 2002. `http://www.cs.bgu.ac.il/~mcodish/TerminWeb`.

Proving termination with adornments

Alexander Serebrenik, Danny De Schreye

Department of Computer Science, K.U. Leuven
Celestijnenlaan 200A, B-3001, Heverlee, Belgium
E-mail: {Alexander.Serebrenik,Danny.DeSchreye}@cs.kuleuven.ac.be

Developing reliable software is one of the most important challenges posed by modern society. In this context verifying correctness of the software becomes crucial. In our work we consider one aspect of program correctness, namely, termination. Logic programming provides a framework with a strong theoretical basis for tackling the problem of termination. On the other hand, due to the declarative formulation of programs, the danger of non-termination may increase. As a result, termination analysis received considerable attention in logic programming (see e.g. [1, 5, 7, 9]). Termination proofs often are based on the concept of a *norm* (a *level mapping*), a function mapping terms (atoms) of the program to the natural numbers.

Recently, using *type information* became a prominent trend in termination analysis for symbolic computations [2, 3, 13]. On the other hand, the study of termination of numerical programs led to the emerging of the *adorning technique* [11, 12]. Both approaches are based on the idea of distinguishing between different subsets of values for variables, and deriving norms and level mappings based on these subsets. Therefore, we would like to investigate these similarities and propose a framework for integration. In the current paper we discuss some preliminary results in this direction.

The key notion of the integrated framework is the notion of *a set of adornments*, partitioning the domain of the predicates arguments into a finite set of pairwise disjoint segments. A set of adornments is usually derived from the program itself. As a basis for a set of adornments one can use types [2], sets of integers [11], and even interargument relations [8]. Given a program and a set of adornments, the program can be transformed such that a predicate p^a in the transformed program P^a is called if and only if the corresponding predicate p is called in the original program P and the arguments of the call are in the segment a. One can show that under certain conditions this transformation preserves termination. In this way (implicit) information on the domain is made explicit in the transformed program. It should be observed, that instead of one level mapping required for P that is supposed to decrease along all possible computations, in order to prove termination of P^a one can find a number of potentially less sophisticated different level mappings for each one of the "cases".

In the examples we have considered, such level-mappings can be constructed automatically, and thus, they play a key role in automation of the approach.

After the transformation step is performed, existing termination analysers can be applied to infer termination of the transformed program. Since the transformation above preserved termination, termination of the original program is implied as well. The importance of the transformation can be illustrated by observing that for such examples as *dist* [6], none of the termination analysers available (cTI [9], TALP [10], TermiLog [7], TerminWeb [4], and Hasta-La-Vista [11]) is powerful enough to prove termination of the original program, while all of them succeed in proving termination of the transformed one.

We summarise the discussion above, by claiming that the main contribution of this work is twofold. First, it provides a link between two different but related approaches to termination analysis based on type information by integrating both of them in one framework. Second, the integrated approach turned out to be powerful enough to prove termination of such example as *dist* [6], that could not have been analysed by the previous techniques. As a future work we consider implementing the approach and evaluating its robustness experimentally.

References

1. K. R. Apt, E. Marchiori, and C. Palamidessi. A declarative approach for first-order built-in's in Prolog. *Applicable Algebra in Engineering, Communication and Computation*, 5(3/4):159–191, 1994.
2. M. Bruynooghe, M. Codish, S. Genaim, and W. Vanhoof. Reuse of results in termination analysis of typed logic programs. In M. V. Hermenegildo and G. Puebla, editors, *Static Analysis, 9th International Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 477–492. Technical University of Madrid (Spain), School of Computer Science, Springer Verlag, 2002.
3. M. Bruynooghe, W. Vanhoof, and M. Codish. Pos(T): Analyzing dependencies in typed logic programs. In D. Bjørner, M. Broy, and A. V. Zamulin, editors, *Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, PSI 2001, Akademgorodok, Novosibirsk, Russia, July 2-6, 2001, Revised Papers*, volume 2244 of *Lecture Notes in Computer Science*. Springer Verlag, 2001.
4. M. Codish and C. Taboch. A semantic basis for termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
5. S. Decorte, D. De Schreye, and H. Vandecasteele. Constraint-based termination analysis of logic programs. *ACM TOPLAS*, 21(6):1137–1195, November 1999.
6. N. Dershowitz and C. Hoot. Topics in termination. In C. Kirchner, editor, *Rewriting Techniques and Applications, 5th International Conference*, volume 690 of *Lecture Notes in Computer Science*, pages 198–212. Springer Verlag, 1993.
7. N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing*, 12(1-2):117–156, 2001.
8. N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. Unfolding mystery of the *mergesort*. In N. Fuchs, editor, *Proceedings of the 7^{th} International Workshop on Logic Program Synthesis*

and Transformation, volume 1463 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

9. F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs. In M. Maher, editor, *Proceedings of the 1996 Joint International Conference and Syposium on Logic Programming*, pages 7–21. The MIT Press, 1996.
10. E. Ohlebusch. Automatic termination proofs of logic programs via rewrite systems. *Applicable Algebra in Engineering, Communication and Computing*, 12(1-2):73–116, 2001.
11. A. Serebrenik and D. De Schreye. Inference of termination conditions for numerical loops in Prolog. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conferencerence, Proceedings*, volume 2250 of *Lecture Notes in Computer Science*, pages 654–668. Springer Verlag, 2001.
12. A. Serebrenik and D. De Schreye. On termination of logic programs with floating point computations. In M. V. Hermenegildo and G. Puebla, editors, *9th International Static Analysis Symposiumsium*, volume 2477 of *Lecture Notes in Computer Science*, pages 151–164. Springer Verlag, 2002.
13. W. Vanhoof and M. Bruynooghe. When size does matter - Termination analysis for typed logic programs. In A. Pettorossi, editor, *Logic-based Program Synthesis and Transformation, 11th International Workshop, LOPSTR 2001, Selected Papers*, volume 2372 of *Lecture Notes in Computer Science*, pages 129–147. Springer Verlag, 2002.

Approximating Dependency Graphs without using Tree Automata Techniques

Nao Hirokawa and Aart Middeldorp

University of Tsukuba, Tsukuba 305-8573, Japan

1 Introduction

In the dependency pair method of Arts and Giesl [2], one of the most popular methods for automatically proving (innermost) termination of TRSs, a TRS is transformed into groups of ordering constraints such that (innermost) termination of the system is equivalent to the solvability of these groups. The number and size of these groups is determined, among others, by the approximation used to estimate the dependency graph. The nodes of the dependency graph $\mathsf{DG}(\mathcal{R})$ of a TRS $\mathcal{R}$ are the dependency pairs of $\mathcal{R}$ and there is an arrow from $s \to t$ to $u \to v$ if and only if there exist substitutions σ and τ such that $t\sigma \to^* u\tau$. In the innermost dependency graph $\mathsf{IDG}(\mathcal{R})$ the requirement $t\sigma \to^* u\tau$ is strengthened to (1) $t\sigma \xrightarrow{\mathsf{i}}{}^* u\tau$ and (2) $s\sigma$ and $u\tau$ are in normal form. Here $\xrightarrow{\mathsf{i}}$ denotes the innermost rewrite relation. Since the relations $\to^*$ and $\xrightarrow{\mathsf{i}}{}^*$ are not computable in general, $\mathsf{DG}(\mathcal{R})$ and $\mathsf{IDG}(\mathcal{R})$ have to be approximated in order to arrive at a mechanizable criterion for (innermost) termination.

In Section 2 we discuss approximations for the dependency graph and in Section 3 we do the same for the innermost dependency graph.

2 Dependency Graph

The following definition stems from [1].

Definition 1. *Let $\mathcal{R}$ be a TRS. The nodes of the* estimated *dependency graph $\mathsf{EDG}(\mathcal{R})$ are the dependency pairs of $\mathcal{R}$ and there is an arrow from $s \to t$ to $u \to v$ if and only if $\mathsf{REN}(\mathsf{CAP}(t))$ and u are unifiable. Here CAP replaces all outermost subterms with a defined root symbol by distinct fresh variables and REN replaces all occurrences of variables by distinct fresh variables.*

Middeldorp [3] showed that better approximations of the dependency graph are obtained by adopting tree automata techniques. These techniques are however computationally expensive. In a more recent paper Middeldorp [4] showed that the estimation of Definition 1 can be improved by symmetry considerations without incurring the overhead of tree automata techniques.

Definition 2. *Let $\mathcal{R}$ be a TRS over a signature $\mathcal{F}$ and let $\mathcal{S} \subseteq \mathcal{R}$. The result of replacing all outermost subterms of a term t with a root symbol in $\mathcal{D}_{\mathcal{S}}^{-1}$ by*

fresh variables is denoted by $\mathsf{CAP}_{\mathcal{S}}^{-1}(t)$. *Here* $\mathcal{D}_{\mathcal{S}}^{-1} = \{\mathrm{root}(r) \mid l \to r \in \mathcal{S}\}$ *if* $\mathcal{S}$ *is non-collapsing and* $\mathcal{D}_{\mathcal{S}}^{-1} = \mathcal{F}$ *otherwise. The nodes of the* estimated* *dependency graph* $\mathsf{EDG}^*(\mathcal{R})$ *are the dependency pairs of* $\mathcal{R}$ *and there is an arrow from* $s \to t$ *to* $u \to v$ *if and only if both* $\mathsf{REN}(\mathsf{CAP}(t))$ *and* u *are unifiable, and* t *and* $\mathsf{REN}(\mathsf{CAP}_{\mathcal{R}}^{-1}(u))$ *are unifiable.*

For instance, with the new estimation proving the termination of Toyama's famous rule $\mathsf{f}(\mathsf{a},\mathsf{b},x) \to \mathsf{f}(x,x,x)$ becomes trivial as there are no cycles in the estimated* dependency graph.

A comparison between the estimation of Definition 2 and the tree automata based approximations described in [3] can be found in [4]. From the latter paper we recall the identity $\mathsf{EDG}(\mathcal{R}) = \mathsf{EDG}^*(\mathcal{R})$ for collapsing $\mathcal{R}$.

3 Innermost Dependency Graph

The new estimation in Definition 4 is inspired by the one of Definition 2.

Definition 3 ([1]). *Let* $\mathcal{R}$ *be a TRS. The nodes of the* estimated *innermost dependency graph* $\mathsf{EIDG}(\mathcal{R})$ *are the dependency pairs of* $\mathcal{R}$ *and there is an arrow from* $s \to t$ *to* $u \to v$ *if and only if* $\mathsf{CAP}_s(t)$ *and* u *are unifiable with mgu* σ *such that* $s\sigma$ *and* $t\sigma$ *are in normal form. Here* CAP_s *replaces all outermost subterms* different from s *with a defined root symbol by distinct fresh variables.*

Definition 4. *Let* $\mathcal{R}$ *be a TRS. The nodes of the* estimated* *innermost dependency graph* $\mathsf{EIDG}^*(\mathcal{R})$ *are the dependency pairs of* $\mathcal{R}$ *and there is an arrow from* $s \to t$ *to* $u \to v$ *if and only if both* $\mathsf{CAP}_s(t)$ *and* u *are unifiable with mgu* σ *such that* $s\sigma$ *and* $t\sigma$ *are in normal form, and* $\mathsf{REN}(\mathsf{CAP}_{\mathcal{U}(t)}^{-1}(u))$ *and* t *are unifiable with mgu* σ *such that* $s\sigma$ *and* $t\sigma$ *are in normal form. Here* $\mathcal{U}(t)$ *denotes the set of* usable rules *[1] for the term* t.

The following example shows that we cannot omit REN from $\mathsf{REN}(\mathsf{CAP}_{\mathcal{U}(t)}^{-1}(u))$ without violating the soundness condition $\mathsf{IDG}(\mathcal{R}) \subseteq \mathsf{EIDG}^*(\mathcal{R})$, which is essential for inferring innermost termination.

Example 5. Consider the TRS $\mathcal{R} = \{\mathsf{f}(x,x) \to \mathsf{f}(\mathsf{g}(x),x), \mathsf{g}(\mathsf{h}(x)) \to \mathsf{h}(x)\}$. There are two dependency pairs: (1): $\mathsf{F}(x,x) \to \mathsf{F}(\mathsf{g}(x),x)$ and (2): $\mathsf{F}(x,x) \to \mathsf{G}(x)$. Since $\mathsf{F}(\mathsf{g}(\mathsf{h}(x)),\mathsf{h}(x)) \xrightarrow{\mathsf{i}} \mathsf{F}(\mathsf{h}(x),\mathsf{h}(x))$, $\mathsf{IDG}(\mathcal{R})$ contains arrows from (1) to (1) and (2). However, $\mathsf{CAP}_{\mathcal{U}(\mathsf{f}(\mathsf{g}(x),x))}^{-1}(\mathsf{F}(\mathsf{g}(x),x)) = \mathsf{CAP}_{\mathcal{R}}^{-1}(\mathsf{F}(\mathsf{g}(x),x)) = \mathsf{F}(\mathsf{g}(x),x)$ does not unify with $\mathsf{F}(x',x')$.

Note that in the above example $\xrightarrow{\mathsf{i}}$ differs from $(\xleftarrow{\mathsf{i}})^{-1}$. It is also not difficult to show that replacing CAP^{-1} by CAP_v^{-1} (or CAP_s^{-1}) would make Definition 4 unsound.

4 Comparison

The following theorem summarizes the relationships between the various approximations.

Table 1. Dependency graph statistics.

TRS	DPs	EDG arrows	EDG SCCs	EDG cycles	EDG* arrows	EDG* SCCs	EDG* cycles	EIDG arrows	EIDG SCCs	EIDG cycles	EIDG* arrows	EIDG* SCCs	EIDG* cycles
[2]:3.23	2	4	1	3	2	1	1	4	1	3	2	1	1
[2]:3.44	4	4	2	2	0	0	0	4	2	2	0	0	0
[2]:3.45	4	5	3	3	2	3	2	5	3	3	2	3	2
[2]:3.48	6	17	2	8	12	2	4	17	2	8	12	2	4
[2]:4.20a	3	3	2	2	1	2	1	2	1	1	0	1	0
[2]:4.20b	4	7	2	4	5	2	3	5	2	2	3	2	1
[2]:4.21	6	12	2	6	8	2	4	6	2	2	2	2	0
[2]:4.37b	4	6	3	3	3	2	2	2	2	2	2	2	2
[5]:2.8	8	24	3	7	24	3	7	19	3	3	18	3	3
[5]:2.51	3	8	1	6	7	1	5	8	1	6	7	1	5
[5]:2.52	9	36	4	17	35	4	16	36	4	17	35	4	16
[5]:4.31	3	4	2	2	4	2	2	4	2	2	2	1	1
[5]:4.44	4	4	2	2	0	2	0	4	2	2	0	2	0
[5]:4.59	6	12	3	5	4	3	2	12	3	5	4	3	2

Theorem 6. *For any TRS $\mathcal{R}$, the following inclusions hold:*

$$\begin{array}{ccccc} \mathsf{DG}(\mathcal{R}) & \subseteq & \mathsf{EDG}^*(\mathcal{R}) & \subseteq & \mathsf{EDG}(\mathcal{R}) \\ \cup\mathsf{I} & & \cup\mathsf{I} & & \cup\mathsf{I} \\ \mathsf{IDG}(\mathcal{R}) & \subseteq & \mathsf{EIDG}^*(\mathcal{R}) & \subseteq & \mathsf{EIDG}(\mathcal{R}) \end{array}$$

Unlike the inclusion $\mathsf{EDG}^*(\mathcal{R}) \subseteq \mathsf{EDG}(\mathcal{R})$, the inclusion $\mathsf{EIDG}^*(\mathcal{R}) \subseteq \mathsf{EIDG}(\mathcal{R})$ need not become an equality for collapsing $\mathcal{R}$, due to the use of usable rules in the second part of Definition 4. An (artificial) example illustrating this is provided by the TRS consisting of the rules $\mathsf{f}(\mathsf{a},\mathsf{b}) \to \mathsf{f}(\mathsf{f}(\mathsf{a},\mathsf{a}),\mathsf{f}(\mathsf{b},\mathsf{b}))$ and $\mathsf{g}(x) \to x$.

Table 1 lists all examples in [2] and Sections 3 and 4 of [5] where the new estimations make a difference.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *TCS*, 236:133–178, 2000.
2. T. Arts and J. Giesl. A collection of examples for termination of term rewriting using dependency pairs. Technical Report AIB-2001-09, RWTH Aachen, 2001.
3. A. Middeldorp. Approximating dependency graphs using tree automata techniques. In *Proc. IJCAR*, volume 2083 of *LNAI*, pages 593–610, 2001.
4. A. Middeldorp. Approximations for strategies and termination. In *Proc. 2nd WRS*, volume 70(6) of *ENTCS*, 2002.
5. J. Steinbach and U. Kühler. Check your ordering – termination proofs and open problems. Technical Report SR-90-25, Universität Kaiserslautern, 1990.

Proving termination by the Reduction Constraint Framework

Cristina Borralleras[1] and Albert Rubio[2]

[1] Universitat de Vic, Spain
Email: cristina.borralleras@uvic.es
[2] Universitat Politècnica de Catalunya, Barcelona, SPAIN
Email: rubio@lsi.upc.es

We will show how to translate the termination proof of a term rewrite system (TRS) using the *Monotonic Semantic Path Ordering* (MSPO) [BFR00] into a constraint solving problem. By using the definition of MSPO a disjunction of constraints is obtained, such that, if any of these constraints can be solved, then the TRS is proved to be terminating.

Our constraints have the same semantics as the ones obtained by the successful Arts and Giesl's *Dependency Pair* method (DP) [AG00], and, in particular, one of the constraints obtained from the definition of the MSPO coincides with the one given by the DP method.

On the one hand, this shows that DP can be seen as a particular case of our method, and the examples show that the constraint considered by DP is not always the best one to be considered. On the other hand, since both kind of constraints share the same semantics, we can reuse all techniques developed to solve DP constraints like, for instance, the DP graph or many other ideas developed in [AG00]. Although we have not used the ideas of other works like [GA02] or [Mid01] yet, they can easily be adapted to our context. On the contrary, some techniques we have developed in our implementation can be used to solve DP constraints as well. The fact that both methods share the same kind of constraints are good news, since, from now on, the necessary research on development and implementation of constraint solving techniques can directly be made for both methods at once.

Additionally, we study the application of our techniques to prove termination of innermost rewriting. In order to reuse our framework, the TRS is modified by adding constraints to the rules, which approximate the restrictions imposed by the strategy. A constrained rule can be applied if the substitution satisfies the constraint. Hence, a TRS is innermost terminating if its constrained version is terminating.

A constrained TRS is terminating if all instances of each constrained rule (i.e. the instances satisfying the constraint) are included in a *monotonic well-founded ordering*. Therefore, we can apply MSPO but taking the constraints of the rules into account. This is done by inheriting the constraints when applying MSPO. As a result, we obtain a disjunction of constrained constraints, which, to avoid confusion, will be called *decorated constraints* (note that the constraints coming from the rules are added to the literals of the constraints coming from MSPO).

Using this decorated constraints we can cover all techniques applied in the DP method for innermost rewriting, and furthermore, we have developed a new constraint

solving mechanism which allows us to avoid, in many cases, the use of narrowing proposed in the DP method.

The same ideas applied to the innermost strategy can be applied, as well, to other strategies that can be approximated by means of constrained rules, like, for instance, rewriting with priorities [Pol98].

Finally, the results reported in the very recent papers [TG03], which combines dependency pairs with the *size-change principle* (proposed in [LJBA01] to verify termination of functional programs automatically), and [HM03], which presents some interesting implementation ideas about the cycle analysis and the generation of term interpretations, can both be easily adapted to our context. Note that the latter is an example of the research on constraint solving techniques that directly apply to both methods, and, in fact, in our system we have independently developed and implemented similar ideas for the generation of term interpretations.

Our method has been implemented in a system called "Termptation" (available at `//www.lsi.upc.es/~albert/`) which automatically proves termination of rewriting and innermost rewriting. A detailed description of this framework can be found in [Bor03].

References

[AG00] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.

[BFR00] C. Borralleras, M. Ferreira and A. Rubio. Complete monotonic semantic path orderings. *Proc. of the 17th International Conference on Automated Deduction*, LNAI 1831:346–364, Pittsburgh, USA, 2000. Springer-Verlag.

[Bor03] C. Borralleras. Ordering-based methods for proving termination automatically. PhD thesis, Universitat Politècnica de Catalunya, Dept. LSI., 2003. Available at `//www.lsi.upc.es/~albert/cristinaphd.ps`.

[GA02] J. Giesl, T. Arts. Verification of Erlang Processes by Dependency Pairs. In *Applicable Algebra in Engineering, Communication and Computing*, 12(1,2):39-72, 2001.

[HM03] N. Hirokawa and A. Middeldorp. Automating the Dependency Pair Method. In *Proceedings of the 19th International Conference on Automated Deduction (CADE'03)*, Miami, Florida, USA, 2003.

[LJBA01] C. S. Lee, N. D. Jones and A. M. Ben-Amram. The size-change principle for program termination. In *Proceedings POPL'01*, pages 81–92, 2001.

[Mid01] A. Middeldorp. Approximating Dependency Graphs using Tree Automata Techniques. In *Proceedings of the International Joint Conference on Automated Reasoning*, LNAI 2083:593-610, Siena, Italy, 2001. Springer-Verlag.

[Pol98] J. van de Pol. Operational semantics of rewriting with priorities. *Theoretical Computer Science*, 200(1-2):289–312, 1998.

[TG03] R. Thiemann and J. Giesl. Size-Change Termination for Term Rewriting. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA'03)*, Valencia, Spain, June 2003.

On the Complexity of Deciding the Derivation Length in Term Rewriting Systems

Michele Flammini, Paola Inverardi, Domenico Mango and Monica Nesi

Dipartimento di Informatica, Università di L'Aquila
via Vetoio, loc. Coppito, I-67100 L'Aquila Italy

email: {flammini,inverard,mango,monica}@di.univaq.it

Verification techniques based on rewriting can play an active role in many fields of software system analysis. To this respect, rewriting based tools can become complementary to model-checking or finite state based tools. However, in order to be practical these techniques have to exhibit a reasonable degree of efficiency, therefore, according to [5], a systematic study of the *quantitative* aspects of term rewriting is in order.

Complexity issues on term rewriting systems have been variously studied in the literature. The time complexity of determining the derivation length in a term rewriting system (*TRS* for short) is a well-known and investigated issue. From a computational complexity point of view, in general this problem is intractable, since termination is even undecidable [6]. A series of papers have then shown that undecidability also holds if the *TRS* contains only three rules [8], only two rules [3], one rule [1] and one rule which is left-linear, non-overlapping and variable preserving [2].

In this paper we investigate a numerical refinement of the termination problem, from now on referred to as *Min-DL*, in which, given a term t, a *TRS* R and a numerical bound k, we are asked to determine if there exists a derivation of t to normal form of length at most k. This problem originated from the need of looking for criteria to suitably bound the length of a terminating derivation in rewriting based tools for the analysis of software systems. In particular, we got interested in this problem when dealing with a rewriting strategy [7] that simulates the completion process of an equational theory in a bottom-up manner. In that case, given a term, it is important to give a *measure* of the search space originated by that term, when rewritten in all possible ways through terminating derivations.

Like the classical proofs of undecidability of termination, some of our negative results are accomplished by simulating Turing machines, although there are several differences between NP-hardness and undecidability simulations. First, we can restrict to Turing machines whose running time is bounded by a suitable polynomial function evaluated on the input size. This allows us to get simpler simulations since we can bound the portion of tape scanned during the computation. On the other hand, we are forced to use non-deterministic Turing machines and, under the restrictions imposed on the structure of the considered *TRS*, we must guarantee that the length of the computations in the Turing machines and

of the corresponding maximal derivations in the TRS are polynomially related. In fact, even if non-determinism does not influence the decidability of a problem, if $P \neq NP$ it affects the tractability in terms of polynomial solvability (see [4] for examples of polynomial simulations).

The problem *Min-DL* is NP-hard, as it does not even belong to the class NP. In fact, k can be exponential in $|t|+|R|$ and derivations of polynomial length can generate terms of exponential size with respect to $|t|+|R|$. Therefore, if polynomial time solvability is concerned, suitable restrictions on the set of allowable instances must be defined.

Definition 1. *Given a term t, a TRS R and a numerical bound k, the triple $\langle t,R,k\rangle$ is p-feasible for a given polynomial p if the following conditions are satisfied:*

1. $k \leq p(|t|+|R|)$;
2. *if* $t \xrightarrow{j} t'$, *then* $|t'| \leq p(|t|+|R|+j)$.

Hence, if $\langle t,R,k\rangle$ is a p-feasible triple, then k and the size of any term derived from t in at most k steps are polynomially bounded (according to p) in the size of t and R.

Given any polynomial p, let *Min-DL*$_p$ be the restriction of *Min-DL* on the set of the instances such that $\langle t,R,k\rangle$ is a p-feasible triple. Then clearly *Min-DL*$_p$ is in NP. In fact, by the p-feasible property, every derivation of length at most k can be described in polynomial space with respect to the size of t and R, and it is possible to check in polynomial time if it is a derivation in R from t to normal form of at most k steps.

Unfortunately, *Min-DL*$_p$ is NP-complete in most cases, even under significant restrictions on the structure of the TRS. In fact, following the line of the results on the indecidability of termination, if one is interested in having a low number of rules the following theorem can be proved.

Theorem 1. *The* Min-DL$_p$ *problem is NP-complete for a suitable polynomial p under the restriction that the TRS R has only one rule.*

The proof is inspired to Dauchet's ideas in [1, 2], namely we provide a general transformation from a non-deterministic Turing machine NT. More precisely, we give a TRS R and a term t such that NT has an accepting computation on a given input x if and only if there is a derivation from t to normal form of at most k steps. By exploiting similar arguments, it is possible to prove the following theorem.

Theorem 2. *The* Min-DL$_p$ *problem is NP-complete for a suitable polynomial p under the restriction that the term t has a unique normal form and the TRS R has only two rules.*

Even if we do not claim it explicitly, the same negative results also hold for the analogous *Max-DL* decision problem in which we are asked if there is a derivation of t to normal form having length more than k.

In conclusion, we have shown that even the non-determinism induced by a single rewriting rule is sufficient to make the problem intractable. Similar negative results also hold if we consider the maximum derivation length.

Many questions remain open. First of all, are the two problems polynomially solvable when the *TRS* has only one rule and the term a unique normal form?

All the *TRSs* in the transformations given in the proofs of the above theorems contain overlapping or self-overlapping rules: what about non-overlapping *TRSs*? As it can be easily checked, under this assumption, if the *TRS* is right-linear or more in general in each rule the number of occurrences of each variable in the left-hand side is at least equal to the number of occurrences in the right-hand side, a minimum or maximum length derivation can be determined in polynomial time. In fact, if there is a term t that matches the left-hand side of a rule r_1 and a subterm t' of t that matches a rule r_2, since the *TRS* is non-overlapping, t' is a subterm of another subterm t'' of t that in the application of r_1 matches a variable of its left-hand side. Then, since the number occurrences of t'' cannot be increased by applying r_1, if we apply r_2 after r_1 we cannot get a longer derivation to normal form. On the contrary, if in each rule of the *TRS* the number of occurrences of each variable in the left-hand side is at most equal to the number of occurrences in the right-hand side, if we apply r_1 after r_2 we cannot get a longer derivation to normal form.

Although the results shown in the paper exhibit a high negative valence, we believe there can still be room for searching *TRS* characterizations (beyond the ones described above) which permit to suitably constrain the *bad* non-determinism in a *TRS* and allow polynomial solvability.

References

1. M. Dauchet. Termination of rewriting is undecidable in the one-rule case. In *13th Symposium on Mathematical Foundation of Computer Science (MFCS)*, volume 324 of *Lecture Notes in Computer Science*, pages 262–270. Springer-Verlag, 1988.
2. M. Dauchet. Simulation of Turing machines by a left-linear rewrite rule. In *3rd International Conference on Rewriting Techniques and Applications (RTA)*, volume 355 of *Lecture Notes in Computer Science*, pages 109–120. Springer-Verlag, 1989.
3. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.
4. M.R. Garey and D.S. Johnson. *Computers and Intractability.* W.H. Freeman, 1979.
5. D. Hofbauer and C. Lautemann. Termination proofs and the length of derivations (Preliminary version). In *3rd International Conference on Rewriting Techniques and Applications (RTA)*, volume 355 of *Lecture Notes in Computer Science*, pages 167–177. Springer-Verlag, 1989.
6. G. Huet and D.S. Lankfork. On the uniform halting problem for term rewriting systems. Technical Report 282, INRIA, Le Chesnay, France, 1978.
7. P. Inverardi and M. Nesi. A Strategy to Deal with Divergent Rewrite Systems. In *Proceedings of CTRS '92*, volume 656 of *Lecture Notes in Computer Science*, pages 458–467. Springer-Verlag, 1992.
8. R. Lipton and L. Snyder. On the halting of tree replacement systems. In *Proc. of the Conference on Theoretical Computer Science*, pages 43–46, Univ. of Waterloo, Waterloo, Canada, 1977.

Proving Liveness in Ring Protocols by Termination

Hans Zantema[1] and Jürgen Giesl[2]

[1] Department of Computer Science, TU Eindhoven, P.O. Box 513,
5600 MB Eindhoven, The Netherlands, h.zantema@tue.nl
[2] LuFG Informatik II, RWTH Aachen, Ahornstr. 55,
52074 Aachen, Germany, giesl@informatik.rwth-aachen.de

We consider the following family of protocols on a ring of processes (similar to a token ring protocol). Every process is in one of finitely many states. Depending on the state of a particular process and the states of a number of its neighbors a step can be done, by which the states of the process and of its neighbors may change. For a protocol of this shape we want to prove the following property:

> Starting from an initial configuration satisfying some property, in every infinite run of the protocol, after finitely many steps one will reach a configuration in which none of the processes is in state no.

Here no is a particular state of a process, typically meaning 'not received'. Thus, the above property means that eventually some message will be received by every process.

This property to be proved is an example of a liveness property: we have to prove that in every ongoing computation eventually 'something good' will happen. In this case the 'something good' is the receipt of the message by all processes. Proving liveness properties is closely related to proving termination as we pointed out in [3].

As an instance of such a protocol we consider the case in which apart from no a process can be in state sent or in state rec (received). Initially at least one of the processes is in state rec which means that it has received a message (token). Now the protocol is defined as follows:

> If a process is in state rec then it may send its message to its right neighbor which then will be in state rec, while the process itself then will be in state sent.

Clearly, at least one process will always be in state rec, and this procedure can go on forever. In [3] we introduced an approach to use termination techniques for term rewriting in order to prove liveness properties of protocols. For example, our approach in [3] can verify the desired property for the particular ring protocol above. However, for other instances in our family of ring protocols the technique from [3] may fail. This is partly due to the sound but incomplete automation of that method and partly due to the fact that the approach in [3] works for networks of processes of arbitrary shape. Hence there, the behavior of the ring topology must also be encoded in the protocol description. But then it can be difficult to state the desired property in the form of a liveness property as permitted in [3].

Therefore, in the present paper we introduce an alternative technique to the one in [3] which is especially designed for ring protocols. It states that for proving the desired liveness property it suffices to prove termination of the string rewriting system (SRS) describing the steps that can be done in the protocol.

First we explain how the protocol is described by such an SRS. We assumed that every process can be in one of finitely many states. Let the alphabet for the SRS correspond to these states. Now the states of a consecutive number of processes in the ring can be described by a string over this alphabet. Depending on this string, by the protocol rule it may be replaced by a string of the same length. For instance, in the above example the corresponding SRS consists of the three rules

$$\begin{aligned} \mathsf{rec\ rec} &\to \mathsf{sent\ rec} \\ \mathsf{rec\ sent} &\to \mathsf{sent\ rec} \\ \mathsf{rec\ no} &\to \mathsf{sent\ rec}. \end{aligned}$$

Theorem 1. *Let R be the SRS corresponding to a ring protocol of the given shape. Assume that the symbol* no *does not occur in any right-hand side of R. Then the corresponding liveness property holds on arbitrary ring size if and only if R is terminating.*

The condition that no does not occur in any right-hand side is essential, for instance if R consists of the single rule $\mathsf{rec\ no} \to \mathsf{no\ rec}$ then R is terminating but the corresponding liveness property does not hold.

By using this theorem, the liveness property for the above example can be proved by the simple observation that the corresponding SRS is terminating by the recursive path order [2].

Now we consider a more complicated example for which the methods from [3] require a solution of a TRS termination problem for which our attempts to use standard techniques all failed.

> If a process is in state rec then it may send its message to its two right neighbors which then will be both in state rec, while the process itself then will be in state sent.

This protocol rule may have to be applied an exponential number of times (exponential in the size of the ring) before all processes receive the message. For this protocol, the corresponding SRS consists of the nine rules

$$\mathsf{rec}\ p\ q \to \mathsf{sent\ rec\ rec}$$

where p, q run over the three symbols rec, sent, and no. Now a direct application of the recursive path order fails, but termination can easily be proved by the dependency pair technique [1] in combination with the recursive path order (by choosing all symbols to be equal in the precedence). As an alternative termination proof, note that a string rewriting system is terminating iff its reversed variant (where each string in the rules is reversed) is terminating. For the reversed system $q\ p\ \mathsf{rec} \to \mathsf{rec\ rec\ sent}$, the polynomial interpretation $[\mathsf{sent}](x) = 2x+1, [\mathsf{rec}](x) = 2x$ suffices for proving termination. Hence, the desired liveness property is verified.

The kind of string rewriting involved in Theorem 1 is in fact *ring rewriting* rather than string rewriting: the objects that are rewritten are not strings having a begin and an end, but they are rings, being strings in which the begin and the end are connected and in which one abstracts from the position representing this connection. More precisely,

for any alphabet Σ we define the set $\mathsf{Ring}(\Sigma)$ of rings over Σ by $\mathsf{Ring}(\Sigma) = \Sigma^*/\sim$ where $\sim$ is the equivalence relation on Σ^* defined by

$$u \sim v \iff \exists u_1, u_2 \in \Sigma^* : u = u_1 u_2 \wedge v = u_2 u_1.$$

Writing $[u]$ for the equivalence class of u w.r.t $\sim$, for an SRS R we define the corresponding ring rewrite relation $\circ\!\!\rightarrow_R$ on $\mathsf{Ring}(\Sigma)$ by

$$[u] \circ\!\!\rightarrow_R [v] \iff \exists u', v' : u \sim u' \wedge v \sim v' \wedge u' \rightarrow_R v'.$$

Now the liveness property from Theorem 1 can be stated as follows: in every infinite $\circ\!\!\rightarrow_R$-reduction after finitely many steps a ring is obtained not containing the symbol no. Surprisingly, for the SRSs given in the two examples the relation $\circ\!\!\rightarrow_R$ is not terminating since they contain the rules $\mathsf{rec\ sent} \rightarrow \mathsf{sent\ rec}$ and $\mathsf{rec\ sent\ rec} \rightarrow \mathsf{sent\ rec\ rec}$, respectively, both giving rise to the cyclic reduction $[\mathsf{rec\ sent\ rec}] \circ\!\!\rightarrow_R [\mathsf{sent\ rec\ rec}] = [\mathsf{rec\ sent\ rec}]$. The proof of Theorem 1 can be given using the following observation. Let R be an SRS in which the symbol no does not occur and let $[\mathsf{no}\ u] \circ\!\!\rightarrow_R [v]$. Then there exists a string w such that $u \rightarrow_R w$ and $[\mathsf{no}\ w] = [v]$. This shows that for a terminating SRS R, rules without no can only be applied finitely many times, and rules with no on the left- but not on the right-hand side decrease the number of occurring no-symbols.

Apart from liveness, one can wonder how termination of $\circ\!\!\rightarrow_R$ relates to termination of $\rightarrow_R$ for arbitrary SRSs R. It is easily seen that termination of $\circ\!\!\rightarrow_R$ implies termination of $\rightarrow_R$, but the above example shows that the converse does not hold. This leads to following question: given an SRS R, how to prove termination of $\circ\!\!\rightarrow_R$? It turns out that arguments using decrease of weight, and a version of semantic labelling can be given for this kind of ring rewriting, but for techniques like recursive path order or dependency pairs which strongly rely on the consideration of term structure, we did not (yet) succeed in finding a variant capable of proving termination of $\circ\!\!\rightarrow_R$.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
2. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.
3. J. Giesl and H. Zantema. Liveness in rewriting. In *Proc. 14th RTA*, LNCS, 2003. To appear. Extended version appeared as Technical Report AIB-2002-11, RWTH Aachen, Germany. Available from `http://aib.informatik.rwth-aachen.de`.

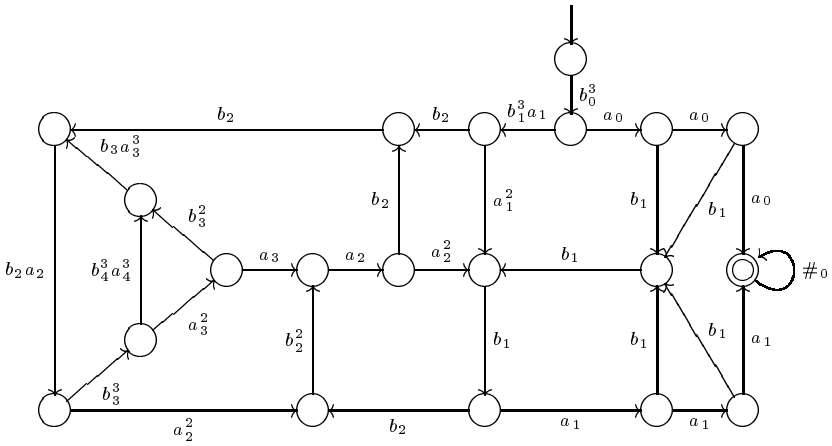
Match-Bounded String Rewriting Systems and Automated Termination Proofs

Alfons Geser[1], Dieter Hofbauer[2], and Johannes Waldmann[3]

[1] National Institute of Aerospace, 144 Research Drive,
Hampton, Virginia 23666, USA. Email: geser@nianet.org
[2] Fachbereich Mathematik/Informatik, Universität Kassel
D-34109 Kassel, Germany. Email: dieter@theory.informatik.uni-kassel.de
[3] Fakultät für Mathematik und Informatik, Universität Leipzig
D-04109 Leipzig, Germany. Email: joe@informatik.uni-leipzig.de

1 Introduction

We present two new methods for automatically finding termination proofs for string rewriting systems, using results from formal language theory. For example, the following automaton proves that *Zantema's system* $Z = \{a^2b^2 \to b^3a^3\}$ is terminating since Z is match-bounded for its forward closures, see Proposition 5 below.



In many rewriting systems, the flow of information is restricted in some way, which may imply nice properties like termination, linearly bounded derivational complexity, or preservation of regular languages. E.g., McNaughton [McN01] and independently Ferreira and Zantema [FZ95] have used extra letters to indicate absence of information flow through certain positions. Kobayashi et al. [KKST01] have used markers for the beginning and the end of a redex.

2 Match-Bounded String Rewriting Systems

A natural idea to restrict the flow of information is to bound dependencies in rule applications, i.e., *matchings* of left hand sides. For this purpose, we will annotate positions in strings by natural numbers that indicate their *matching height*. Positions in a reduct will get height $h+1$ if the minimal height of all positions in the corresponding redex was h.

Define the morphisms $\mathrm{lift}_c : \Sigma^* \to (\Sigma \times \mathbb{N})^*$ for $c \in \mathbb{N}$ by $\mathrm{lift}_c : a \mapsto (a, c)$, $\mathrm{base} : (\Sigma \times \mathbb{N})^* \to \Sigma^*$ by $\mathrm{base} : (a, c) \mapsto a$, and $\mathrm{height} : (\Sigma \times \mathbb{N})^* \to \mathbb{N}^*$ by $\mathrm{height} : (a, c) \mapsto c$. For a string rewriting system R over an alphabet Σ we define the system

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

over alphabet $\Sigma \times \mathbb{N}$. Note that this is an infinite system. Every derivation modulo $\mathrm{match}(R)$ corresponds to a derivation modulo R, and vice versa.

Definition 1. *A string rewriting system R over Σ is called* match-bounded *for $L \subseteq \Sigma^*$ by $c \in \mathbb{N}$ if $\max(\mathrm{height}(x)) \leq c$ for every $x \in \mathrm{match}(R)^*(\mathrm{lift}_0(L))$. If we omit L, then it is understood that $L = \Sigma^*$.*

This definition of match bounds from [GHW03] was inspired by Ravikumar's concept of *change bounds* [Rav97]. Note that Ravikumar's construction only applies to length-preserving systems.

Our results on match-bounded systems are proved by means of recent contributions to the theory of *deleting* string rewrite systems [HW03]. A string rewriting system R over an alphabet Σ is *$>$-deleting* for a precedence $>$ on Σ if for each rule $\ell \to r$ in R and for each letter a in r, there is some letter b in ℓ with $b > a$. The system R is *deleting* if it is $>$-deleting for some precedence $>$. If R is deleting, then R is terminating, and R has linear derivational complexity.

Proposition 1 ([HW03]). *Let R be a deleting string rewriting system over Σ. Then there are an extended alphabet $\Gamma \supseteq \Sigma$, a finite substitution $s \subseteq \Sigma^* \times \Gamma^*$, and a context-free string rewriting system C over Γ such that*

$$R^* = (s \circ C^{-*})|_\Sigma.$$

This implies that deleting systems effectively preserve regularity, and that inverse deleting systems effectively preserve context-freeness [Hib74].

Here, the decomposition result is applied to give the following results.

Proposition 2 ([GHW03]). *Given a string rewriting system R, a regular language L, and $c \in \mathbb{N}$, it is decidable whether R is match-bounded by c for L.*

Proof. Construct the system $R_c = \mathrm{match}_c(R)$, that is, the restriction of $\mathrm{match}(R)$ to the alphabet $\Sigma \times \{0, 1, \ldots, c\}$, and compute (a finite automaton for) $L_c = R_c^*(\mathrm{lift}_0(L))$. Then R is match-bounded by c for L if and only if no string in $\mathrm{lift}_c(\mathrm{lhs}(R))$ occurs as a factor in L_c. □

Proposition 3 ([GHW03]). *If R^- is match-bounded, then termination of R is decidable.*

Proof. Assume R^- is match-bounded by c. According to Proposition 1, construct the decomposition $\mathrm{match}_c(R^-)^* = (s \circ C^{-*})|_\Sigma$. Now, R is terminating if and only if there is no string x such that $s^-(C^*(x))$ is infinite. This is a decidable property since C^* is a context-free substitution and s^- is the inverse of a finite substitution. □

Example 1. The inverse $Z^- = \{b^3a^3 \to a^2b^2\}$ of Zantema's system is match-bounded by 2. Therefore, a termination proof according to Proposition 3 can be found automatically.

3 Match-Bounds for Forward Closures

Of course, if R is match-bounded, then R is terminating, and R has linear derivational complexity. But we can do better than that by restricting the investigation to right hand sides of forward closures [LM78,Der87]. For a string rewriting system R over Σ, the set of *forward closures* $\mathrm{FC}(R) \subseteq \Sigma^* \times \Sigma^*$ is defined as the least set containing R such that

- if $(u, v) \in \mathrm{FC}(R)$ and $v \to_R w$, then $(u, w) \in \mathrm{FC}(R)$ (*inside reduction*), and
- if $(u, v\ell_1) \in \mathrm{FC}(R)$ and $(\ell_1\ell_2 \to r) \in R$ for strings $\ell_1 \neq \epsilon$, $\ell_2 \neq \epsilon$, then $(u\ell_2, vr) \in \mathrm{FC}(R)$ (*right extension*).

Let $\mathrm{RFC}(R)$ denote the set of right hand sides of forward closures.

Proposition 4 ([Der81]). *A string rewriting system R is terminating on Σ^* if and only if R is terminating on* $\mathrm{RFC}(R)$.

We can obtain $\mathrm{RFC}(R)$ as a set of descendants modulo the rewriting system $R_\# = R \cup \{\ell_1\# \to r \mid (\ell_1\ell_2 \to r) \in R, \ell_1 \neq \epsilon, \ell_2 \neq \epsilon\}$ over $\Sigma \cup \{\#\}$, where right extension is simulated via the end-marker #. Indeed,

$$\mathrm{RFC}(R) = R_\#^*(\mathrm{rhs}(R) \cdot \#^*) \cap \Sigma^*.$$

Proposition 5. *If $R_\#$ is match-bounded for* $\mathrm{rhs}(R) \cdot \#^*$, *then R is terminating.*

Proof. Since $R \subseteq R_\#$ and $\mathrm{RFC}(R) \subseteq R_\#^*(\mathrm{rhs}(R) \cdot \#^*)$, it follows that R is match-bounded for $\mathrm{RFC}(R)$, thus terminating on $\mathrm{RFC}(R)$. □

Example 2. The finite automaton depicted in the introduction accepts the language $\mathrm{match}(Z_\#)^*(\mathrm{lift}_0(\mathrm{rhs}(Z) \cdot \#^*))$. The transition diagram also exhibits the match bound 4.

An implementation of our algorithms can be accessed via a CGI-interface at `http://theo1.informatik.uni-leipzig.de/~joe/bounded/`.

References

[Der81] Nachum Dershowitz. Termination of linear rewriting systems. In Shimon Even and Oded Kariv (Eds.), *Proc. 8th Int. Coll. Automata, Languages and Programming ICALP-81*, *LNCS* Vol. 115, pp. 448–458. Springer-Verlag, 1981.

[Der87] Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1–2):69–115, 1987.

[FZ95] Maria C. F. Ferreira and Hans Zantema. Dummy elimination: Making termination easier. In Horst Reichel (Ed.), *Proc. 10th Int. Symp. Fundamentals of Computation Theory FCT-95*, LNCS Vol. 965, pp. 243–252. Springer-Verlag, 1995.

[GHW03] Alfons Geser, Dieter Hofbauer and Johannes Waldmann. Match-bounded string rewriting systems. In *Proc. 28th Int. Symp. Mathematical Foundations of Computer Science MFCS-03*, LNCS. Springer-Verlag, 2003. To appear.

[Hib74] Thomas N. Hibbard. Context-limited grammars. *Journal of the ACM*, 21(3):446–453, 1974.

[HW03] Dieter Hofbauer and Johannes Waldmann. Deleting string rewriting systems preserve regularity. In *Proc. 7th Int. Conf. Developments in Language Theory DLT-03*, LNCS. Springer-Verlag, 2003. To appear.

[LM78] Dallas S. Lankford and David R. Musser. A finite termination criterion. Technical report, Information Sciences Institute, Univ. of Southern California, Marina-del-Rey, CA, 1978.

[McN01] Robert McNaughton. Semi-Thue systems with an inhibitor. *Journal of Automated Reasoning*, 26:409–431, 2001.

[Rav97] Bala Ravikumar. Peg-solitaire, string rewriting systems and finite automata. In Hon-Wai Leong, Hiroshi Imai, and Sanjay Jain (Eds.) *Proc. 8th Int. Symp. Algorithms and Computation ISAAC-97*, LNCS Vol. 1350, pp. 233–242. Springer-Verlag, 1997.

[KKST01] Yuji Kobayashi, Masashi Katsura, and Kayoko Shikishima-Tsuji. Termination and derivational complexity of confluent one-rule string-rewriting systems. *Theoretical Computer Science*, 262(1):583–632, 2001.

Dependency, Termination and Overlap Analysis of Higher-order Programs: short abstract

Neil D. Jones, DIKU, University of Copenhagen

May 7, 2003

Abstract

Some new analyses of higher-order programs are formulated and proven correct using SOS (Structural Operational Semantics). Advances over previous work: size-change analysis is extended to programs with higher-order functions; size-change graphs are computed from the program using semi-compositional SOS; correctness is easily verified because the exact and approximate semantics are closely parallel; and an SOS *overlap analysis* is described that identifies call sites where one function can call another (or itself) more than once with the same arguments.[1]

The size-change termination analysis of [6] is based on a very simple principle: Program `p` terminates on every input if every infinite sequence of function calls (that follow program control flow) would cause at least one value from a well-founded domain to decrease infinitely.[2] Algorithmically, one constructs a *data-flow graph* G_c for each call from program function `f` to function `g`, with edges annotated by size-change labels $\downarrow$ or $=$ whenever this can definitely be seen to be true (and no edge if not). The least graph set $\mathcal{S}$ containing every G_c and closed under composition is then computed. Finally, `p` is size-change terminating in the above sense iff every idempotent graph in $\mathcal{S}$ has an *in situ* decrease $\mathtt{x} \stackrel{\downarrow}{\rightarrow} \mathtt{x}$.

Surprisingly many first-order programs are size-change terminating, even though the approach seems simple-minded at first sight because it ignores all tests appearing in `p`. The method handles Ackermann's function and needs no special treatment for *general recursive* programs containing mutual recursion and parameter permutation. Further, it is relatively easy to automate, its first implemention being a version programmed to aid the Agda proof assistant [4].

The computational power is known: f is computable by some first-order size-change terminating program iff f is multiple recursive [2]. This is a respectably large function class from a practical viewpoint. Further, many *algorithms* are size-change terminating, so a program manipulation system can certify as terminating programs written naturally, e.g., without restriction to an annoying primitive recursive syntax.

[1] Memoisation at such call sites can yield exponential efficiency increases.

[2] Like many good ideas, this one has been discovered more than once. Logic Programming has a similar analysis [9], and a similar construction for determinising Büchi automata predates both that work and ours [10].

Overview of main results

Size-change termination is extended to higher-order programs, given in an ML- or Haskell-like "named combinator" syntax without lambdas. A further analysis is developed to to detect overlapping function calls.

1. An example program illustrates indirect recursion and "hidden" nonlinear function calls, requiring memoisation to avoid exponential running times.
2. An SOS is given for exact program execution. Sem_{exact} represents a functional value by a "closure" $\langle \mathtt{f}\ v_1 \ldots v_k \rangle$, where $k < arity(\mathtt{f})$.
3. An SOS $\mathrm{Sem}_{ex\text{-}d\text{-}flow}$ is given that both describes exact program execution and traces data-flow. Data-flow graphs can be extracted from any finite $\mathrm{Sem}_{ex\text{-}d\text{-}flow}$ proof tree. A minor extension yields size-change graphs.
4. Next, an approximate *control-flow semantics* $\mathrm{Sem}_{ap\text{-}c\text{-}flow}$, similar in effect to 0-CFA [?], is obtained by abstracting Sem_{exact}. This is shown to be finitely computable and to account for all possible program control flows.
5. These two semantics are combined to yield an approximate $\mathrm{Sem}_{ap\text{-}c\text{-}d\text{-}flow}$ that accounts for all possible program control flows and yields size-change graphs. The analysis is shown both correct and finitely computable.
6. An `f`-to-`g` *overlap* consists of two call sequences $cs, cs' : \mathtt{f} \to \mathtt{g}$ that 1) are distinct, 2) yield the same argument transformations, and 3) are *coupled* in the sense that any computation contains cs iff it contains cs'. Such nonlinear control flow will definitely cause computational redundancy if encountered, and is particularly expensive if `g` calls `f` again.
7. The problem of detecting exact overlap is seen to be undecidable. A final approximate program analysis $\mathrm{Sem}_{ap\text{-}overlap}$ is given that will detect systematic overlap if it is present.
8. It is shown that any HOPR (higher-order primitive recursive) program will be certified as terminating by the new method. Consequence: Function f is computable by some higher-order size-change terminating program iff it is definable in Gödel's System T, i.e., iff it is ϵ_0-recursive.

An example

Consider the following second-order HOPR program with call sites labeled $1-6$:

```
Types:
       F : • → •, Id : • → •,
       S : • → • → •,
       N : (• → •) → • → •
Definitions:
       F x    =  1: S x x
       S t    =  if t=0 then 2: Id else 3: N (4: S(t-1))
       N r i  =  5: r(i) + 6: r(i+1)
       Id z   =  z
```

Analyses:

The full paper contains details of the SOS rule systems, their usage to perform the following program analyses, and some soundness proofs.

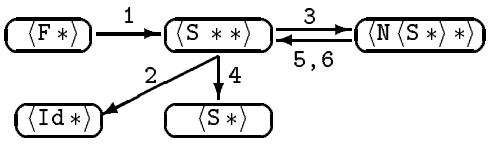
1. **Control-flow analysis, net effects of the defined functions:**

   $\langle \mathtt{F}\, * \rangle \mapsto *$ and $\langle \mathtt{Id}\, * \rangle \mapsto *$, each maps a base value into a base value.

   $\langle \mathtt{S}\ *\, * \rangle \mapsto *$, function S maps two base values into a base value.

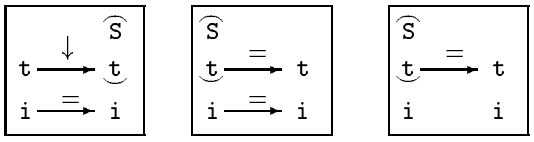
   $\langle \mathtt{N}\, \langle \mathtt{S}\, * \rangle\, * \rangle \mapsto *$, N maps a function and a base value into a base value. The function can only be the result of applying S to a single base value.

2. **Control flow analysis, effects of the function calls:**



   Call site 4 is not "really" recursive since the S argument list is incomplete. On the other hand, call sites 5 and 6 both complete the S argument list, so $3 : \mathtt{S} \to \mathtt{N}, 5 : \mathtt{N} \to \mathtt{S}$ and $3 : \mathtt{S} \to \mathtt{N}, 6 : \mathtt{N} \to \mathtt{S}$ complete recursive loops.

3. **Size-change graphs** for call sites in the two loops:[3]



   $G_3 : \mathtt{S} \to \mathtt{N}$ $G_5 : \mathtt{N} \to \mathtt{S}$ $G_6 : \mathtt{N} \to \mathtt{S}$

4. **Call-duplication analysis:**

   Consider call sequences 3536 and 3635, both taking $\mathtt{S} \to \mathtt{N} \to \mathtt{S} \to \mathtt{N} \to \mathtt{S}$. These are *coupled*: if either call sequence is performed in a computation, then so must the other. Further, they have the same effect: that $\langle \mathtt{S}\, t\, i \rangle$ calls $\langle \mathtt{S}\, (t-2)\, (i+1) \rangle$. Thus the program has a hidden nonlinearity, caused by the interaction of recursion and double usage of parameter r.

   The net effect is *call duplication*, causing the program to run for time of order $\Omega(2^x)$ on input x if, say, call-by-value is used. This can be reduced to a small polynomial by memoisation.

[3] **Remark:** By 1, the N parameter r must have the form of a closure $\langle \mathtt{S}\, \mathtt{t} \rangle$. In graphs G_3, G_5, G_6 this is written vertically, using $\frown$ and $\smile$ for $\langle$ and $\rangle$, respectively.

References

[1] Thomas Arts and Jürgen Giesl. Proving innermost termination automatically. In *Proceedings Rewriting Techniques and Applications RTA'97, Lecture Notes in Computer Science* vol. 1232, pp. 157–171. Springer, 1997.

[2] Amir M. Ben-Amram. General Size-Change Termination and Lexicographic Descent. In *The Essence of Computation: Complexity, Analysis, Transformation.*, volume 2566 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2002.

[3] Wei-Ngan Chin, Siau-Cheng Khoo, and Tat-Wee Lee. Synchronisation analysis to stop tupling. In *Programming Languages and Systems (ESOP'98)*, pages 75–89, Lisbon, 1998. Springer LNCS 1381.

[4] Catarina Coquand. The interactive theorem prover Agda. http://www.cs.chalmers.se/~catarina/agda/, 2001.

[5] Olin Shivers. Control-flow analysis in Scheme. Proceedings of PLDI, the SIGPLAN '88 Conference on *Programming Language Design and Implementation*, June 1988.

[6] Chin Soon Lee, Neil D. Jones, and Amir 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, January 2001.

[7] Neil D. Jones and Arne J. Glenstrup. Program Generation, Termination, and Binding-time Analysis. In *First ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering GPCE'02*, volume 2487 of *Lecture Notes in Computer Science*, pages 1–31. Springer, 2002.

[8] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall. Download accessible from `www.diku.dk/users/neil`, 1993.

[9] Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. Termilog: A system for checking termination of queries to logic programs. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, Jun 22–25, 1997*, volume 1254 of *Lecture Notes in Computer Science*, pages 444–447. Springer, 1997.

[10] Aravinda Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

[11] Chris Speirs, Zoltan Somogyi, and Harald Søndergaard. Termination analysis for Mercury. In Pascal Van Hentenryck, editor, *Static Analysis, Proceedings of the 4th Int. Symposium, SAS '97, Paris, France, Sep 8–19, 1997, Lecture Notes in Computer Science*, vol. 1302, pp. 160–171. Springer, 1997.

Termination Dependencies

Nachum Dershowitz*
School of Computer Science, Tel Aviv University, Israel
Email: nachumd@tau.ac.il

The innovative *dependency-pair* termination method of [1, 2] relies on two important observations:

- If a rewrite system is nonterminating, then there is an infinite derivation with at least one redex at the top of a term (see, for example, [4, p. 287]).
- If a rewrite system is nonterminating, then there is an infinite derivation in which all proper subterms of every redex are *mortal* (these are the "constricting" derivations of [11]). By "mortal", we mean that it initiates finite derivations only.

Let F be some vocabulary (of constant and function symbols) and T, the set of terms constructed from it. The dependency-pair method can be reformulated—and somewhat strengthened—in terms of two (related) quasi-orderings, as follows:

> *A rewrite system terminates if there are well-founded quasi-orderings* $\succsim$ *and* $\succsim'$ *such that*:
> 1. **(Rule)** $\ell \succsim r$ for all rules $\ell \to r$;
> 2. **(Dependency)** $\ell \succ' u$ for all subterms u of the right side r of a rule $\ell \to r$ that are not also subterms of the left side ℓ;
> 3. **(Monotonicity)** $u \succsim v$ implies $f(\cdots,u,\cdots) \succsim f(\cdots,v,\cdots)$ for all symbols $f \in F$ and (ground) terms $\cdots,u,v,\cdots \in T$; and
> 4. **(Harmony)** $u \succsim v$ implies $f(\cdots,u,\cdots) \succsim' f(\cdots,v,\cdots)$ for all symbols $f \in F$ and (ground) terms $\cdots,u,v,\cdots \in T$,
>
> *where* $\succ$ *and* $\succ'$ *are the strict partial orderings (*$\succsim \setminus \precsim$ *and* $\succsim' \setminus \precsim'$*) associated with* $\succsim$ *and* $\succsim'$.

Only $\succsim$ is required to be monotonic. We have excluded dependency inequalities for right-hand subterms that also appear on the left; this includes all variables. Harmony is called "quasi-monotonicity of $\succsim'$ with respect to $\succsim$" in [3]. (All inequalities refer to ground terms but can be lifted to free terms in the standard manner: Demanding that $u \succ v$ for terms u,v with free variables means that $u\gamma \succ v\gamma$ for all substitutions γ of ground terms for those variables; see also [2, fn. 5].)

To clarify relations between various termination methods, we first define some properties of binary term relations:

$$\begin{aligned}
\text{Sub}(\sqsupset): &\quad f(\cdots s\cdots) \sqsupset s\\
\text{Mono}(\sqsupset): &\quad s \sqsupset t \Rightarrow f(\cdots s\cdots) \sqsupset f(\cdots t\cdots)\\
\text{Harmony}(\sqsupset,\gg): &\quad s \sqsupset t \Rightarrow f(\cdots s\cdots) \gg f(\cdots t\cdots)\\
\text{Compat}(\sqsupset,\gg): &\quad s \sqsupset t \gg u \Rightarrow s \sqsupset u\\
\text{Rule}_R(\sqsupset): &\quad \ell \to r \in R \Rightarrow \ell \sqsupset r\\
\text{Reduce}_R(\sqsupset): &\quad s \to_R t \Rightarrow s \sqsupset t\\
\text{Depend}_R(\sqsupset): &\quad \ell \to r \in R \wedge s \trianglelefteq r \Rightarrow \ell \sqsupset s \vee s \trianglelefteq \ell
\end{aligned}$$

* This research was supported in part by The Israel Science Foundation (grant no. 254/01).

where R is an arbitrary rewrite system; $\to_R$ is the associated rewrite relation; and $\trianglelefteq$ ($\triangleleft$) is the (proper) subterm relation. All symbols should be understood universally: s,t,u are arbitrary ground terms; ellipses $\cdots$ represent arbitrary lists of terms; $\sqsupset, \gg$ are arbitrary binary relations over terms; and f is an arbitrary function symbol (or constant).

In what follows, let $\succsim$ and $\succsim'$ be arbitrary *well-founded* quasi-orderings. Using the above notation, we can express our version (somewhat stronger that the suggestion in [6, p. 558, *en passant*]) of the dependency method as follows:

Dependency ($\succsim, \succsim'$) [6]**:** Depend$_R(\succ')$, Rule$_R(\succsim)$, Mono($\succsim$), Harmony($\succsim$,$\succsim'$).

More precisely this means

$$\begin{array}{l}\mathrm{WFO}(\succsim) \wedge \mathrm{WFO}(\succsim') \wedge \\ \mathrm{Depend}_R(\succ') \wedge \mathrm{Rule}_R(\succsim) \wedge \mathrm{Mono}(\succsim) \wedge \mathrm{Harmony}(\succsim, \succsim')\end{array} \Rightarrow \mathrm{SN}(R)$$

where

$$\begin{array}{rl}\mathrm{WF}(\gg): & \text{no infinite descending sequences } x_1 \gg x_2 \gg \cdots \\ \mathrm{WFO}(\succsim): & \succsim \text{ is a quasi-ordering and } \mathrm{WF}(\succ) \\ \mathrm{SN}(R): & \mathrm{WF}(\to_R)\end{array}$$

Let $\widehat{F}$ be a mirror image of F: $\widehat{F} = \{\widehat{g} | g \in F\}$. Denote by $\widehat{s}$ the term $s = f(u_1, \ldots, u_n)$ with root symbol $f \in F$ replaced by its mirror image $\widehat{f} \in \widehat{F}$, that is, $\widehat{s} = \widehat{f}(u_1, \ldots, u_n)$. Let $\widehat{T}$ be T's image under $\widehat{\cdot}$. If $\succ$ is a partial ordering of $\widehat{T}$, define another partial ordering $\widehat{\succ}$ as $u \mathrel{\widehat{\succ}} v$, for terms $u, v \in T$, when $\widehat{u} \succ \widehat{v}$. The original method of [2] is approximately:

Dependency Pairs ($\succsim$) [2]**:** Depend$_R(\widehat{\succ})$, Rule$_R(\succsim)$, Mono($\succsim$).

where Mono applies to both hatted ($f \in \widehat{F}$) and bareheaded ($f \in F$) terms. A more recent version [8] of the dependency-pair method is essentially:

Variant ($\succsim, \succ'$) [8]**:** Depend$_R(\succ')$, Rule$_R(\succsim)$, Mono($\succsim$), Compat($\succ', \succsim$).

The methods of [2, 6, 8] exclude only variable subterms x of r from the requirement that $\ell \mathrel{\widehat{\succ}} x$ or $\ell \succ' x$, but the proofs remain valid for all $s \triangleleft \ell$, as in our condition Depend$_R(\succ')$. There is no need to explicitly exclude the case that s is headed by a constructor (as done in [2]). Instead, one simply makes all terms headed by a constructor smaller under $\succ$ than terms headed by defined symbols (doable, since it need not be monotonic). In fact, any term that can never have a top redex, regardless of rewrites below the top, can be made minimal, and be safely ignored in the same way.

The dependency-pair method is derived in [2] from the condition:

Main ($\succsim$) [2]**:** Depend$_R(\succ)$, Mono($\to_R \cap \succsim$).

We contribute the following variants:

Basic ($\succsim$)**:** Depend$_R(\succ)$, Reduce$_R(\succsim)$.

Intermediate ($\succsim, \succ'$)**:** Depend$_R(\succ')$, Reduce$_R(\succsim)$, Compat($\succ'$,$\succsim$).

The following summarizes dependencies between the different methods:

Lemma 1.

1. **Main** ($\succsim$) $\Rightarrow$ **Basic** ($\succsim$)
2. **Basic** ($\succsim^*$) $\Rightarrow$ **Intermediate** ($\succsim$,$\succ'$)
3. **Intermediate** ($\succsim$,$\succ'$) $\Rightarrow$ **Variant** ($\succsim$,$\succ'$)
4. **Main** ($\succsim'$) $\Rightarrow$ **Dependency** ($\succsim$,$\succsim'$)
5. **Dependency** ($\succsim$,$\widehat{\succsim}$) $\Rightarrow$ **Dependency Pairs** ($\succsim$)

where $\succsim^*$ is the transitive closure of $\succsim \cup \succsim'$ (which is well-founded when the two are compatible).

An implication $M \Rightarrow M'$ means that method M' is a special case of method M. To prove the implication, viz. that correctness of the antecedent method M implies correctness of the consequent M', one shows that the *requirements* for M' imply the requirements for M. Thus, any well-founded ordering(s) used by M should be derivatives of those used by M'.

For example, to prove Lemma 1.2, we need to show

$$\begin{aligned} \mathrm{WFO}(\succsim) \wedge \mathrm{WFO}(\succsim') \wedge \mathrm{Depend}_R(\succ') \wedge \mathrm{Reduce}_R(\succsim) \wedge \mathrm{Compat}(\succ',\succsim) \\ \Rightarrow \quad \mathrm{WFO}(\succsim^*) \wedge \mathrm{Depend}_R(\succ^*) \wedge \mathrm{Reduce}_R(\succsim^*) \end{aligned}$$

which follows from compatibility and properties of the transitive closures.

Other termination methods can be summarized using these same properties:

Standard ($\succ$) [10]**:** $\mathrm{Rule}_R(\succ)$, $\mathrm{Mono}(\succ)$.

Kamin & Lévy ($\succ$) [9]**:** $\mathrm{Rule}_R(\succ)$, $\mathrm{Mono}(\rightarrow_R \cap \succ)$.

Quasi-Simplification Ordering ($\succsim$) [4]**:** $\mathrm{Rule}_R(\succ)$, $\mathrm{Sub}(\succsim)$, $\mathrm{Mono}(\succsim)$.

Subterm ($\succsim$) [5]**:** $\mathrm{Rule}_R(\succ)$, $\mathrm{Sub}(\succsim)$, $\mathrm{Mono}(\rightarrow_R \cap \succsim)$.

We can show:

Lemma 2.

1. **Standard** ($\rightarrow_R \cap \succ$) $\Rightarrow$ **Kamin & Lévy** ($\succ$)
2. **Main** ($\succsim$) $\Rightarrow$ **Subterm** ($\succsim$)
3. **Kamin & Lévy** ($\succ^\omega$) $\Rightarrow$ **Subterm** ($\succsim$)
4. **Subterm** ($\succsim$) $\Rightarrow$ **Quasi-Simplification Ordering** ($\succsim$)

where $s \succ^\omega t$ is the well-founded multiset extension [7] of $\succ$ to the bag of all subterms of s,t.

One application of the weakened method shown here could be a "pattern-based" ordering. For the non-monotonic "surface" ordering, one can check whether the term matches a given pattern derived from a rule's left side.

References

1. Thomas Arts. *Automatically Proving Termination and Innermost Normalization of Term Rewriting Systems*. PhD thesis, Universiteit Utrecht, Utrecht, The Netherlands, 1997.
2. Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
3. Cristina Borralleras, Maria Ferreira and Albert Rubio. Complete monotonic semantic path orderings. In: David A. McAllester, ed., *Proceedings of the 17th International Conference on Automated Deduction (CADE)*, *Lecture Notes in Computer Science* 1831, Pittsburgh, PA, Springer Verlag, pages 346–364, June 2000.
4. Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, March 1982.
5. Nachum Dershowitz. Termination of rewriting. *J. Symbolic Computation*, 3(1&2):69–115, February/April 1987.
6. Nachum Dershowitz and David A. Plaisted. Rewriting. Chap. 9 in: *Handbook of Automated Reasoning*, vol. 1, A. Robinson and A. Voronkov, eds., Elsevier Science, pp. 535–610, 2001.
7. Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, August 1979.
8. Jürgen Giesl and Deepak Kapur. Dependency pairs for equational rewriting. In: *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA-01)*, Utrecht, The Netherlands, *Lecture Notes in Computer Science* 2051, Springer Verlag, pages 93–107, 2001.
9. Sam Kamin and Jean-Jacques Lévy. Two generalizations of the recursive path ordering. Unpublished note, Department of Computer Science, University of Illinois, Urbana, IL, February 1980.
10. Dallas S. Lankford. On proving term rewriting systems are Noetherian. Memo MTP-3, Mathematics Department, Louisiana Tech. University, Ruston, LA, May 1979. Revised October 1979.
11. David A. Plaisted. Semantic confluence tests and completion methods. *Information and Control*, 65(2/3):182–215, 1985.

Quasi-Ordered Gap Embedding*

—Extended Abstract—

Nachum Dershowitz and Iddo Tzameret

School of Computer Science
Tel-Aviv University
P.O.B. 39040
Ramat Aviv, Tel-Aviv 69978
Israel
email: Nachumd@tau.ac.il Tzameret@tau.ac.il

Kruskal's Tree Theorem [3], stating that finite trees are well-quasi-ordered under homeomorphic embedding, and its extensions, have played an important rôle in both logic and computer science. In proof theory, it was shown to be independent of certain logical systems by exploiting its close relationship with ordinal notation systems (cf. [6]), while in computer science it provides a common tool for proving the termination of many rewrite-systems via the *recursive path* and related orderings [1]. For demonstrating termination of rewriting, it is beneficial to use a *partial* (or *quasi*-) ordering on labels, rather than a total one.

In [7], it was shown that many important order-theoretic properties of the well-partial-ordered precedence relations on function symbols carry over to the induced termination ordering. This is done by defining a general framework for precedence-based termination orderings via (so-called) *relativized ordinal notations*. Based on a few examples, it is further conjectured that every such application of a partial-order to an ordinal notation system carries the order-theoretic properties of the partial-order to the relativized notation system. An example of such a construction, using Takeuti's ordinal diagrams, is introduced in [5] under the name *quasi-ordinal-diagrams*. The definition of these diagrams is the only result we know of that deals with gap embedding of trees and *quasi*-ordered labels.

Kříž [2] proved a conjectured extension by Harvey Friedman of the Tree Theorem, which states that finite trees labelled by ordinals are well-quasi-ordered under gap embedding. Our work extends this further to finite trees with well-quasi-ordered labels, with the following restriction (which is shown necessary): *Every node is comparable with all its ancestors.*

Let $\mathcal{T}$ be a set of *ordered* (rooted, plane-planted) finite trees, with nodes well-quasi ordered by $\precsim$, and with the above restriction. Let $t^\bullet$ denote the root of tree t. There is a *(gap) subtree* relation $\trianglerighteq$ on trees (which includes all immediate subtrees) with the following properties:

$$s \trianglerighteq t \trianglerighteq u \wedge t^\bullet \succsim u^\bullet \Rightarrow s \trianglerighteq u \tag{1}$$

$$s \trianglerighteq t \trianglerighteq u \wedge s^\bullet \precsim t^\bullet \Rightarrow s \trianglerighteq u \tag{2}$$

$$s \trianglelefteq t \Rightarrow s^\bullet \precsim t^\bullet \vee t^\bullet \precsim s^\bullet \tag{3}$$

* Supported in part by the Israel Science Foundation (grant no. 254/01).

There is also an *(gap) embedding* relation $\hookrightarrow$ on trees with the properties:

$$s \hookrightarrow t \trianglelefteq u \wedge t^\bullet \precsim u^\bullet \Rightarrow s \hookrightarrow u \tag{4}$$
$$s \hookrightarrow t \trianglelefteq u \wedge s^\bullet \precsim u^\bullet \Rightarrow s \hookrightarrow u \tag{5}$$

A *sequence* s is a partial function $s : \mathbf{N} \to \mathcal{T}$. If $s(i)$ is not defined we write $s(i) = \bot$. It is very convenient to extend the subtree relation and node ordering to empty positions of a sequence, so that: $t \trianglerighteq \bot$ and $t^\bullet \precsim \bot^\bullet$.

Let Seq be the set of ω-sequences of trees from $\mathcal{T}$. Define:

$$\begin{aligned} Ds &:= \{i \in \mathbf{N} \mid s(i) \neq \bot\} \\ \text{Bad} &:= \{s \in \text{Seq} \mid \forall i < j \in Ds.\ s(i) \not\hookrightarrow s(j)\} \\ \text{Sub } h &:= \{s \in \text{Seq} \mid \forall i \in Ds.\ h(i) \triangleright s(i)\} \\ \text{Inc } k &:= \{s \overset{\infty}{\subseteq} k \mid \forall i < j \in Ds.\ s^\bullet(i) \precsim s^\bullet(j)\} \end{aligned}$$

where $\triangleright$ is the *proper* (gap) subtree relation. We'll say that s is *infinite* when Ds is.

Since $\precsim$ is a well-quasi-ordering, Inc k is nonempty, as long as k is infinite.

Theorem 1. Bad $= \emptyset$.

In other words, for every $s \in$ Seq there exist $i < j \in Ds$ such that $s(i) \hookrightarrow s(j)$. This extends the result of Kříž to quasi-ordered labels.

Assuming the contrary, the proof builds a minimal counterexample, minimal in the sense that no infinite sequence of proper (gap) subtrees of its elements is also bad (which leads to a contradiction—as in the original proof by Nash-Williams [4]). The construction of the minimal bad sequence proceeds by ordinal induction as follows:

$H(0)$:	$h :\in$ Bad **if** Bad $\cap$ Sub $h = \emptyset$ **then return** h $h_0 :\in$ Inc lex(h)
$H(\alpha+1)$:	**if** Bad $\cap$ Sub $h_\alpha = \emptyset$ **then return** h_α $k := \text{lex}(h_\alpha)$ $\forall i \in \mathbf{N}.\ f(i) := \begin{cases} k(i) & \textbf{if } h_\alpha^\bullet(i) \precsim k^\bullet(i) \\ \bot & \textbf{otherwise} \end{cases}$ $g :\in$ Inc f $\forall i \in \mathbf{N}.\ h_{\alpha+1}(i) := \begin{cases} h_\alpha(i) & \textbf{if } i < \min Dg \\ g(i) & \textbf{otherwise} \end{cases}$
$H(\lambda)$:	$\forall i \in \mathbf{N}.\ \ell(i) := \lim_{\gamma \to \lambda} h_\gamma(i)$ **if** Bad $\cap$ Sub $\ell = \emptyset$ **then return** ℓ $h_\lambda :\in$ Inc lex(ℓ)

where the construct $s :\in S$ chooses an arbitrary s from S ($s = \bot$ if $S = \emptyset$). The function lex : Bad $\to$ Bad chooses a bad sequence of subtrees with (lexicographically) minimal

labels:

lex(h) :	$K := \mathrm{Bad} \cap \mathrm{Sub}\ h$
	for $i := 1$ **to** ∞ **do**
	$\quad m :\in \min\{a^\bullet(i) \mid a \in K\}$
	$\quad k(i) :\in \{a(i) \mid a \in K,\ a^\bullet(i) = m\}$
	$\quad K := \{a \in K \mid a(i) = k(i)\}$
	$k :\in K$
	return k

By construction, we have (for all α and i):

$$Dh_\alpha \supseteq Dh_{\alpha+1} \tag{6}$$

$$h_\alpha(i) \trianglerighteq h_{\alpha+1}(i) \tag{7}$$

$$h^\bullet_\alpha(i) \precsim h^\bullet_{\alpha+1}(i) \tag{8}$$

For each sequence h_α (for every countable ordinal α and for all $i < j \in Dh_\alpha$):

$$h_\alpha(i) \not\mapsto h_\alpha(j) \tag{9}$$

$$h^\bullet_\alpha(i) \precsim h^\bullet_\alpha(j) \tag{10}$$

The successor step of (9,10) is proved by induction; the only interesting case is $i < \min Dg \leq j$, when

$$h^\bullet_{\alpha+1}(i) = h^\bullet_\alpha(i) \precsim h^\bullet_\alpha(j) \precsim f^\bullet(j) = k^\bullet(j) = h^\bullet_{\alpha+1}(j)$$

from which (9) follows using (5). By considering the limit case, it can be seen that for all $\alpha < \beta$:

$$Dh_\alpha \supseteq Dh_\beta \tag{11}$$

Finally, it can be shown that:

1. The constructed sequences h_α are all infinite.
2. The constructed sequences h_α are each distinct.
3. The construction eventually terminates with a minimal bad sequence.

By induction, this result may be extended also to the case where every path in the tree can be partitioned into some bounded number of subpaths with comparable labels.

Moreover, the absence of such a bound yields a bad sequence with respect to gap-embedding: Let a, b, c be three incomparable elements of the node ordering. The following is an antichain with respect to gap embedding:

$$c-a-c \qquad c-b-a-c \qquad c-a-b-a-c \qquad c-b-a-b-a-c\ldots$$

Consequently, the extension of Theorem 1 to set of trees with bounded number of comparable subpaths shows that the above counterexample is *canonical*: Every bad sequence with respect to gap embedding must contain paths of unbounded incomparability.

References

1. N. Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982.
2. I. Kříž. Well-quasiordering finite trees with gap-condition. proof of Harvey Friedman's conjecture. *Ann. of Math.*, 130:215–226, 1989.
3. J. B. Kruskal. Well-quasiordering, the Tree Theorem, and Vazsonyi's conjecture. *Trans. American Mathematical Society*, 95:210–223, 1960.
4. C. St. J. A. Nash-Williams. On well-quasi-ordering finite trees. *Proc. Cambridge Phil. Soc.*, 59:833–835, 1963.
5. M. Okada and G. Takeuti. On the theory of quasi-ordinal diagrams. In: *Logic and Combinatorics* (Arcata, Calif., 1985), Contemp. Math., vol. 65, Amer. Math. Soc., Providence, RI, 1987, pp. 295–308.
6. S.G. Simpson. Nonprovability of certain combinatorial properties of finite trees. In L. A. Harrington, editor, *Harvey Friedman's research on the foundation of mathematics*, pages 87–117. Elsevier, 1985.
7. A. Weiermann. Proving termination for term rewriting systems. In *Computer Science Logic* (Berne, 1991), volume 626 of *Lecture Notes in Comput. Sci.*, pages 419–428. Springer, Berlin, 1992.

Mathematical and logical aspects of termination orderings

Andreas Weiermann
Faculteit Wiskunde en Informatica
Utrecht University
Budapestlaan 6
3584 CD Utrecht

Abstract

In the first part of the talk we study functorial properties of some termination orderings following ideas from Feferman and Girard. Using a classical result which basically goes back to Ehrenfeucht we obtain some applications to the model theory of termination orderings with respect to infinitary languages.

In the second part we investigate analytical properties of termination orderings with respect to their count functions. These count functions can be classified using Tauberian methods and methods from complex analysis. We give some applications to independence results for Peano arithmetic. Further we obtain zero one laws for segments of the nested multiset ordering (joint work with Woods).

Explicit substitutions and intersection types

PIERRE LESCANNE
LIP, Ecole Normale supérieure de Lyon
43 allée d'Italie 69364 Lyon France
e-mail : Pierre.Lescanne@ens-lyon.fr

12 mai 2003

1 Introduction

Calculi of explicit substitutions have been introduced to give an account to the substitution process in lambda calculus. The idea is to introduce a notation for substitutions explicitely in the calculus. In other words one makes substitutions *first class citizens* whereas the classical lambda calculus leaves them in the meta-theory.

Originally, the expression "explicit substitution" and the concept itself were introduced by Abadi, Cardelli, Curien and Levy [1]. The idea of the authors was to replace categorical combinators by a syntax that extend lambda calculus. Actually the idea of internalizing substitutions into the lambda calculus is older. A first credit should be given to Curien with his $\lambda\rho$ calculus [4] which is the origin of the calculus of [1]. But Curry himself expressed the same wish much earlier. For him combinatory logic was a mean to analyze substitutions. In the introduction of his famous book [5] he noted the advantage of including a calculus of substitutions into the lambda calculus. Perhaps the idea of explicit substitution is not "explicit" there, but at least the program of building a simple system, which analyzes substitutions and which departs less of our intuition than combinatory logic, appears clearly. For Curry this system should not be far from lambda calculus. Nowadays, we would call such a system a *"calculus of explicit substitution"*. Another ancestor of calculi of explicit substitutions is de Bruijn $\lambda C\xi\varphi$ [7]. As its complicated name might indicate, $\lambda C\xi\varphi$ is not an easy calculus as it mixes up the concrete and the abstract syntax in the same framework. In [2], this calculus has been revisited in modern notations. At the noticeable exception of $\lambda\chi$ [10] which uses de Bruijn levels, most of these calculi use de Bruijn indices [6] but more natural approaches have been proposed where names for variables are made explicit like in classical lambda-calculus. The most popular presentation is due to Roel Bloo and Kristoffer Rose [3], but Lins [11] proposed a calculus with the same features in 1985. The work presented here is the result of a cooperation with Daniel Dougherty and Stéphane Lengrand [9, 8].

2 The calculusλx

The calculus of explicit substitution λx extends the syntax of the classical lambda calculus with terms of the form $M\langle x := N\rangle$ called *closures*. They are terms with a body M and an explicit substitution part.

$$M, N ::= x \mid \lambda x.M \mid MN \mid M\langle x := N\rangle .$$

In λx, $\_\langle x := \_\rangle$ is a binary operator which is a binder for x in its left subterms. More specifically, in the term $M\langle x := N\rangle$, x is bound in the subterm M. The concept of free variable needs also to be extended. To emphasize the difference with usual freeness we call this new concept *availability*[1]. Roughly speaking, a variable is not available in a term if it does not appear in the term or it appears in a subtituated part of the term which is associated with a variable not itself available, in other word, a variable is not available if it appears in a term which is going to disappear later on by reduction.

$$\begin{array}{rcll} AV(x) & = & \{x\} & \\ AV(\lambda x.M) & = & AV(M) \setminus \{x\} & \\ AV(MN) & = & AV(M) \cup AV(N) & \\ AV(M\langle x := N\rangle) & = & AV(M) \cup AV(N) \setminus \{x\} & \text{if } x \in AV(M) \\ AV(M\langle x := N\rangle) & = & AV(M) \setminus \{x\} & \text{if } x \notin AV(M) \end{array}$$

The definition of λx makes an extensive use of Barendregt convention on variables, which can be stated as follows *in the same context there exists never a variable which is both bound and free*. This convention plays a key role in the following rules especially in the rule (**Abs**).

$$\begin{array}{lcll} (\lambda x.M)\ P & \to & M\langle x := P\rangle & (\mathbf{B}) \\ (MN)\langle x := P\rangle & \to & M\langle x := P\rangle\, N\langle x := P\rangle & (\mathbf{App}) \\ (\lambda x.M)\langle x := P\rangle & \to & \lambda x.(M\langle x := P\rangle) & (\mathbf{Abs}) \\ x\langle x := P\rangle & \to & P & (\mathbf{VarI}) \\ y\langle x := P\rangle & \to & y & (\mathbf{VarK}) \end{array}$$

3 Types

In order to catch strong normalization we introduce a type system with intersection called $\mathcal{E}$ (see Fig. 1). It is worth to notice that there are two rules for closures. The first called cut, says that if M receives type σ in a context Γ extended with type τ for x then $M\langle x := N\rangle$ receives the same type σ provided N gets type τ in the context Γ. This rule is straightforwardly connected through the Curry-Howard correspondance to the cut rule in natural deduction, hence its name. In addition to cut, we added a new rule, which says that in typing $M\langle x := N\rangle$ one can safely drop the type of x provided that N is typeable and x does not occur (is not available) in M. Then $M\langle x := N\rangle$ gets the same type as M.

[1] Actually there is another extension of freeness to explicit substitutions which is not what we need here.

$$\cap\,\mathsf{I}\quad \frac{\Gamma\vdash M{:}\tau_1 \qquad \Gamma\vdash M{:}\tau_2}{\Gamma\vdash M{:}\tau_1\cap\tau_2} \qquad \cap\,\mathsf{E}\quad \frac{\Gamma\vdash M{:}\tau_1\cap\tau_2}{\Gamma\vdash M{:}\tau_i} \quad i\in\{1,2\}$$

$$\mathsf{drop}\quad \frac{\Gamma\vdash M{:}\sigma \qquad N\,\text{typable}}{\Gamma\vdash M\langle x{:=}N\rangle{:}\sigma} \quad x\notin AV(M)$$

$$\mathsf{cut}\quad \frac{\Gamma,x{:}\tau\vdash M{:}\sigma \qquad \Gamma\vdash N{:}\tau}{\Gamma\vdash M\langle x{:=}N\rangle{:}\sigma}$$

$$\to\mathsf{I}\quad \frac{\Gamma,x{:}\sigma\vdash M{:}\tau}{\Gamma\vdash\lambda x.M{:}\sigma\to\tau} \qquad \to\mathsf{E}\quad \frac{\Gamma\vdash M{:}\sigma\to\tau \qquad \Gamma\vdash N{:}\sigma}{\Gamma\vdash MN{:}\tau}$$

$$\mathsf{start}\quad \frac{}{\Gamma\vdash x{:}\sigma} \quad x\in\Gamma$$

FIG. 1 – The typing system $\mathcal{E}$

4 The result

The main result which is the core of [9] says that *a term M is strongly normalizing if and only if there exists in* $\mathcal{E}$ *a context* Γ *and a type* σ *such that* $\Gamma \vdash M : \sigma$. Actually $\mathcal{E}$ is not the only system which has this property. Indeed van Bakkel and Dezani [12] proposed an alternative system with a rule they call K-cut instead of drop

$$\mathsf{K-cut}\quad \frac{\Gamma, x:\tau\vdash M:\sigma \qquad \Delta\vdash N:\tau}{} \quad \text{if } x\notin\Gamma$$

Their system characterizes also strong normalization in $\lambda\mathsf{x}$.

Références

[1] Martin Abadi, Lucas Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4) :375–416, 1991.

[2] Z. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. $\lambda\upsilon$, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 6(5) :699–722, September 1996.

[3] Roel Bloo and Kristoffer Høgsbro Rose. Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection. In *CSN '95 – Computer Science in the Netherlands*, pages 62–72, November 1995.

[4] P.-L. Curien. Categorical combinators. *Information and Control*, 69 :188–254, 1986.

[5] H. B. Curry, R. Feys, and W. Craig. *Combinatory Logic*, volume 1. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1958.

[6] N. G. de Bruijn. Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Proc. Koninkl. Nederl. Akademie van Wetenschappen*, 75(5) :381–392, 1972.

[7] N. G. de Bruijn. A namefree lambda calculus with facilities for internal definition of expressions and segments. TH-Report 78-WSK-03, Technological University Eindhoven, Netherlands, Department of Mathematics, 1978.

[8] Daniel Dougherty and Pierre Lescanne. Reductions, intersection types, and explicit substitutions. *Mathematical Structures in Computer Science*, 2001.

[9] S. Lengrand, D. Dougherty, and P. Lescanne. An improved system of intersection types for explicit substitutions. In R. Baeza-Yates, U. Montanari, and N. Santoro, editors, *Foundations of Information Technology in the era of Network and Mobile Computing*, pages 511–524. Kluwer Academic Publishers, 2002.

[10] P. Lescanne and J. Rouyer-Degli. Explicit substitutions with de Bruijn's levels. In J. Hsiang, editor, *Proceedings 6th Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*, volume 914 of *Lecture Notes in Computer Science*, pages 294–308. Springer-Verlag, 1995.

[11] R. Lins. A new formula for the execution of categorial combinators. In *Proceedings 8th International Conference on Automated Deduction, Oxford (UK)*, volume 230 of *LNCS*, pages 89–98. Springer-Verlag, 1986.

[12] S. van Bakel and M. Dezani-Ciancaglini. Characterizing strong normalization for explicit substitutions. In *LATIN'02*, volume 2286 of *LNCS*, pages 356–370. Springer-Verlag, 2002.

On Termination of a Tabulation Procedure for Residuated Logic Programming ⋆

C.V. Damásio[1] and M. Ojeda-Aciego[2]

[1] Centro Inteligência Artificial. Univ. Nova de Lisboa. Portugal (cd@di.fct.unl.pt)
[2] Dept. Matemática Aplicada. Univ. de Málaga. Spain (aciego@ctima.uma.es)

Abstract. Residuated Logic Programs allow to capture a spate of different semantics dealing with uncertainty and vagueness. A first result states that for any *definite* residuated logic program the sequence of iterations of the immediate consequences operator reaches the least fixpoint after only finitely many steps. Then, a tabulation query procedure is introduced, and it is shown that the procedure terminates every definite residuated logic program.

In this work we focus on the framework of residuated logic programming, which generalizes several approaches to the extension of logic programming techniques to the fuzzy case. Our aim here is in the use of tabling (or memoizing) methods to increase the efficiency of the previously proposed semantics. For the essentials of residuated logic programming the reader is referred to [3]. The semantics of a residuated logic program is characterised, by the post-fixpoints of the immediate consequences operator $T_{\mathbb{P}}$, whose definition can be easily generalised to the framework of residuated logic programs. Moreover, it can be shown that $T_{\mathbb{P}}$ is always increasing.

Residuated logic programs allow arbitrary combinations of operators in the body of rules, however the most frequently used fuzzy rule systems employ a single type of conjunctor, usually a *t-norm* (an associative, commutative operator on the unit interval $[0,1]$, with 1 as neutral element). For instance, in [2] we illustrated how to encode approximate deductions and fuzzy rules in control systems into residuated logic programming. The bodies of weighted rules obtained by the encoding have the simple form: $\langle A;\vartheta\rangle$ or $\langle A \longleftarrow B_1 \otimes \ldots \otimes B_n;\vartheta\rangle$, where A and B_i's are propositional variables and ϑ is a truth-value in $[0,1]$. Programs having this form are called *definite* residuated logic programs, i.e. where the body of rules is constructed by conjoining together propositional variables with a unique t-norm operator.

Our first theorem concerning termination in this context is the following:

Theorem 1. *Consider a definite residuated logic program over a continuous t-norm. Then, the $T_{\mathbb{P}}$ operator reaches its least fixpoint after finitely many steps.*

Proof (Sketch): It is well-known that every continuous t-norm is either minimum, Archimedean, or the ordinal sum of a family of continuous Archimedean t-norms. Therefore the proof can be split into two cases: the minimum t-norm and any Archimedean t-norm.

⋆ Partially supported by Integrated Action HP2001-0078 and E-42/02.

For the case of minimum the result is obvious because of the monotonicity of $T_{\mathbb{P}}$ and the fact that we have a finite program.

For the case of Archimedean t-norms the result follows from the fact that any computed value by the semantics has the general form ${\vartheta_1}^{n_1} \otimes \cdots \otimes {\vartheta_m}^{n_m}$ and such an element cannot have infinitely many strict upper bounds of that form. □

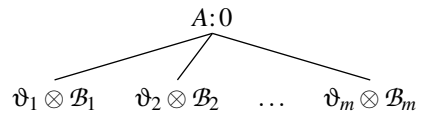
The issue now relies in the definition of an appropriate query procedure for residuated logic programs. In the one hand, the $T_{\mathbb{P}}$ operator is bottom-up but not goal-oriented and in every step the bodies of the rules are recomputed. On the other hand, the usual SLD based implementations of Fuzzy Logic Programming languages are goal-oriented but inherit the problems of non-termination and recomputation of goals. For tackling these issues, the tabulation implementation technique has been proposed in the deductive databases and logic programming communities [1].

We introduce a general tabulation procedure for residuated logic programs. The datatype we will use for the description of the method is that of a *forest*, that is, a finite set of trees. Each of these trees has a root labeled with a propositional symbol together with a truth-value from the real unit interval (called the *current value* for the *tabulated* symbol); the rest of the nodes of each of these trees are labeled with an "extended" formula (defined in [5]).

In the description of the tabulation procedure for residuated logic programming, we will assume a program $\mathbb{P}$ consisting of a finite number of weighted propositional formulas of the form $\langle A \longleftarrow \mathcal{B}, \vartheta \rangle$ together with a query $?Q$. The body is an arbitrary combination of computable monotonic functions applied to propositional variables. The purpose of the computational procedure is to give (if possible) the greatest truth-value for A that can be inferred from the information in the program $\mathbb{P}$.

Operations for Tabulation. Consider the following notation: Given a propositional symbol A, we will denote by $\mathbb{P}(A)$ the set of rules in $\mathbb{P}$ which have head A. The tabulation procedure requires four basic operations: *Create New Tree, New Subgoal, Value Update,* and *Answer Return*. The first operation creates a tree for the first invocation of a given goal. *New Subgoal* is applied whenever a propositional variable in the body of rule is found without a corresponding tree in the forest, and resorts to the previous operation. *Value update* is used to propagate the truth-values of answers to the root of the corresponding tree. Finally, *Answer Return* substitutes a propositional variable by the current truth-value in the corresponding tree. We now describe formally the operations:

Create New Tree (CNT). Given a propositional symbol A, construct the tree below, where we assume that $\mathbb{P}(A) = \{\langle A \longleftarrow \mathcal{B}_j, \vartheta_j \rangle \mid j = 1, \ldots, m\}$,



and append it to the current forest (finite list of trees). In the case that the forest did not exist, then generate a singleton list with the tree above.

New Subgoal (NS). Select a non-tabulated propositional symbol C occurring in a leaf of some tree (this means that there is no tree in the forest with the root node labeled with C), then create a new tree as indicated in CNT, and append it to the forest.

Value Update (VU). If there are no propositional symbols in a leaf, then evaluate the corresponding arithmetic formula, assume that its value is, say, s and update the current value r of the propositional symbol at the root of the tree by the value of $\sup(r,s)$.

Answer Return (AR). Select in any non-root node a propositional symbol C which is tabulated, and consider that the current value of C is r.

– If the propositional symbol has been selected in a leaf $\mathcal{B}[\ldots,C,\ldots]$, then extend the branch with $\mathcal{B}[\ldots,r,\ldots]$

$$\begin{array}{c}\mathcal{B}[\ldots,C,\ldots]\\ \vert\\ \mathcal{B}[\ldots,r,\ldots]\end{array}$$

– Otherwise, if the propositional symbol has been selected in a non-leaf node $\mathcal{B}[\ldots,C,\ldots]$ such as in the left of Fig. 1 then, if $s < r$, then update the whole branch substituting the constant s by r, as in the right of Fig. 1.

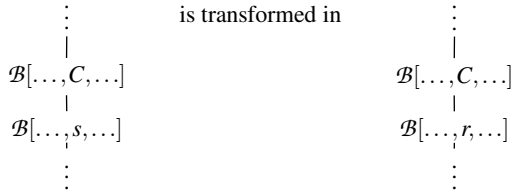


Fig. 1. Updating a branch with a current value.

Note that the only operation which changes the values of the roots of the trees in the forest is VU. Note also that the only nodes with several immediate successors are the root nodes. From there downwards, the extension is done by AR, which either updates the nodes of an existing branch or extends the branch with one new node.

A non-deterministic procedure for tabling. Now, we can state the general non-deterministic procedure for calculating the answer to a given query by using a tabling technique in terms of the previous rules.

Initial step Create the initial forest applying CNT to the query.
Next steps Non-deterministically select a propositional symbol and apply one of the operations NS, VU, and AR.

As in any non-deterministic procedure, it is necessary to show that the obtained result is independent from the different choices made during the execution of the algorithm. With this aim we prove two propositions, which provide as a consequence the independence of the ordering of applications of steps in the tabulation proof procedure, as well as soundness and completeness.

Proposition 1.

1. *The current values of a terminated forest generate a model of $\mathbb{P}$. That is, the current values are greater or equal than those given by the least fixpoint of the immediate consequences operator $T_{\mathbb{P}}$.*
2. *Given a forest (terminated or not), then for all roots C_j:r_j we have that there exists an iteration k, of the $T_{\mathbb{P}}$ operator such that $r_j \leq T_{\mathbb{P}}^k(C_j)$.*

As an easy consequence of the previous proposition we obtain:

Theorem 2.

1. *Every terminated forest calculates exactly the minimal model for the program.*
2. *The tabulation procedure terminates if and only if the minimal model is reached by iterating the $T_{\mathbb{P}}$ operator a finite number of times.*

Since in Theorem 1 we showed that for every definite residuated logic program the minimal model is reached after finitely many iterations of the $T_{\mathbb{P}}$ operator, we conclude:

Corollary 1. *The tabulation procedure is terminating for the class of definite residuated logic programs.*

Concluding remarks. A non-deterministic tabulation goal-oriented query procedure has been introduced. We have also shown that the procedure terminates for the class of definite residuated logic programs. As future work, on the one hand, we would like to study further conditions of termination for the general lattice-valued framework of residuated logic programs; on the other hand, we would also study the generalized case of multi-adjoint logic programs [4]. Another interesting research line is to attempt the extension of this technique to the case of first order definite residuated programs.

References

1. W. Chen, T. Swift, and D. S. Warren. Efficient top-down computation of queries under the well-founded semantics. *Journal of Logic Programming*, 24(3):161–199, 1995.
2. C. V. Damásio and M. Ojeda-Aciego. A tabulation proof procedure for residuated logic programming. Submitted, 2003.
3. C. V. Damásio and L. M. Pereira. Monotonic and residuated logic programs. In *Symbolic and Quantitative Approaches to Reasoning with Uncertainty, ECSQARU-2001*, pages 748–759. Lect. Notes in Artificial Intelligence 2143, 2001.
4. J. Medina, M. Ojeda-Aciego, and P. Vojtáš. Multi-adjoint logic programming with continuous semantics. In *Logic Programming and Non-Monotonic Reasoning, LPNMR'01*, pages 351–364. Lect. Notes in Artificial Intelligence 2173, 2001.
5. J. Medina, M. Ojeda-Aciego, and P. Vojtáš. A procedural semantics for multi-adjoint logic programming. In *Progress in Artificial Intelligence, EPIA'01*, pages 290–297. Lect. Notes in Artificial Intelligence 2258, 2001.

Termination of Computational Restrictions of Rewriting and Termination of Programs*

Salvador Lucas

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n, E-46022 Valencia, Spain
e.mail: slucas@dsic.upv.es

Termination of rewriting [Der87,Zan03] is often proposed as a suitable theory for proving termination of programs which are executed by rewriting. Programming languages and systems whose operational principle is based on reduction (e.g., functional, algebraic, and equational programming languages as well as theorem provers based on rewriting techniques) need, however, to break down the non-determinism which is inherent to reduction relations to make computations feasible. This is usually done by means of some *reduction strategy*, i.e., a concrete rule to specify the (non-empty set of) reduction steps which can be issued on any term which is not a normal form. Thus, termination of a program $\mathcal{R}$ (where $\mathcal{R}$ is a TRS) can be more precisely defined as the *termination of the strategy* $\mathbb{S}$ which is used to execute $\mathcal{R}$. Here, by termination of a strategy $\mathbb{S}$ for a TRS $\mathcal{R}$, we mean the termination of the reduction relation $\rightarrow_{\mathbb{S}} \subseteq \rightarrow^{+}$ associated to $\mathbb{S}$.

Traditionally, the most important question about a rewriting strategy $\mathbb{S}$ is whether it is *normalizing*, i.e., no infinite $\mathbb{S}$-sequence $t \rightarrow_{\mathbb{S}} t' \rightarrow_{\mathbb{S}} \cdots$ starts from a term t having a normal form. Obviously, every rewriting strategy $\mathbb{S}$ is forced to run forever when faced to terms t having no normal form. Then, the following property is obvious.

Proposition 1. *Let $\mathcal{R}$ be a TRS and $\mathbb{S}$ be a strategy for $\mathcal{R}$. Then, $\mathbb{S}$ is terminating if and only if $\mathcal{R}$ is normalizing and $\mathbb{S}$ is normalizing.*

This proposition says that we *cannot* obtain a terminating behavior for a program $\mathcal{R}$ running under a given strategy $\mathbb{S}$ unless the program is normalizing, i.e., *every* term t has a normal form. However, many interesting programs are not normalizing. In particular, those which can be used to deal with 'infinite' data structures. For instance, the TRS that corresponds to the following Maude program:

```
fmod SEL-FIRST-PRIMES is
  sorts Nat LNat .
  op 0     : -> Nat .
  op s     : Nat -> Nat .
  ops nil primes : -> LNat .
  op cons  : Nat LNat -> LNat [strat (1 0)] .
  op sel   : Nat LNat -> Nat .
  op first : Nat LNat -> LNat .
```

* This work has been partially supported by CICYT TIC2001-2705-C03-01 and MCYT Acción Integrada HU 2001-0019.

```
  op nats  : Nat -> LNat .
  op sieve : LNat -> LNat .
  op filter : LNat Nat Nat -> LNat .
  vars X Y M N : Nat .
  var Z : LNat .
  eq filter(cons(X,Z),0,M) = cons(0,filter(Z,M,M)) .
  eq filter(cons(X,Z),s(N),M) = cons(X,filter(Z,N,M)) .
  eq sieve(cons(0,Z)) = sieve(Z) .
  eq sieve(cons(s(N),Z)) = cons(s(N),sieve(filter(Z,N,N))) .
  eq nats(N) = cons(N,nats(s(N))) .
  eq primes = sieve(nats(s(s(0)))) .
  eq sel(s(X),cons(Y,Z)) = sel(X,Z) .
  eq sel(0,cons(X,Z)) = X .
  eq first(0,Z) = nil .
  eq first(s(X),cons(Y,Z)) = cons(Y,first(X,Z)) .
endfm
```

is not normalizing: the expression `primes` is intended to arbitrarily approximate the list of prime numbers (see [KdV03]) and has no normal form. As mentioned before, the problem is that rewriting strategies must rewrite terms which are not normal forms. The following notion permits us to avoid the limitation expressed by Proposition 1. In the following definition, $(A,\to)$ is an abstract reduction system (ARS), where $\to \subseteq A \times A$ for a given set A; also, for a reduction relation $\to' \subseteq A \times A$, $\mathsf{NF}_{\to'}$ is the set of all $\to'$-normal forms.

Definition 1. *Let $(A,\to)$ be an ARS. A binary relation $\rightharpoonup$ on A is a* computational restriction *of $\to$ if $\rightharpoonup \subseteq \to^+$ and $\mathsf{NF}_{\rightharpoonup} \neq \mathsf{NF}_{\to}$.*

When considering a TRS $\mathcal{R} = (\mathcal{F}, R)$, $\to$ is the (one-step) rewrite relation $\to_{\mathcal{R}}$ induced by $\mathcal{R}$ and we write $\mathsf{NF}_{\mathcal{R}}$ rather than $\mathsf{NF}_{\to_{\mathcal{R}}}$. Note that the condition $\mathsf{NF}_{\rightharpoonup} \neq \mathsf{NF}_{\mathcal{R}}$ (or, equivalently, $\mathsf{NF}_{\rightharpoonup} \supset \mathsf{NF}_{\mathcal{R}}$, since $\rightharpoonup \subseteq \to^+$ implies that $\mathcal{R}$-normal forms are $\rightharpoonup$-normal forms) makes the difference between the notion of rewriting strategy and that of computational restriction of rewriting:

1. every rewriting strategy $\mathbb{S}$ for a TRS $\mathcal{R}$ satisfies $\to_{\mathbb{S}} \subseteq \to^+$; however, $\mathsf{NF}_{\to_{\mathbb{S}}} = \mathsf{NF}_{\mathcal{R}}$ by definition of strategy. On the other hand,
2. every subset $\rightharpoonup$ of $\to^+$ satisfying $\mathsf{NF}_{\rightharpoonup} = \mathsf{NF}_{\mathcal{R}}$ can be just considered as a rewriting strategy $\mathbb{S}_{\rightharpoonup}$ given by

$$\mathbb{S}_{\rightharpoonup}(t) = \{t = t_1 \to t_2 \to \cdots \to t_n = s \mid t \rightharpoonup s\}$$

   for each term t.

Using *restrictions of rewriting*, we are still able to define (restricted) strategies which only consider $\rightharpoonup$-steps. In fact, the notion of computational restriction (of rewriting) as given in Definition 1 is new but already used (as well as the corresponding strategies) in the literature.

Computational restriction	Related Strategies
Demand-driven reduction [MR92]	Demand-driven strategies [AL02]
Outermost-needed reduction [Ant92]	Outermost-needed strategy [Ant92]
Context-Sensitive Rewriting [Luc98]	Context-Sensitive Strategies [Luc02a]
”	E-strategies [Eke98],
”	*Just-in-time* [Pol01]
”	Non-strict evaluation [GM03]
Lazy Rewriting [FKW00,Luc02b]	On-Demand Strategies [AEGL02]
On-demand rewriting [Luc01a]	On-Demand Strategies [AEGL02,NO01]

The use of computational restrictions of rewriting in the computational model of programs is, then, interesting. For instance, the program `SEL-FIRST-PRIMES` above is terminating thanks to the strategy annotation $(1\ 0)$ for symbol `cons` which ensures that reductions on the second argument of calls to `cons` are not allowed. In particular, the evaluation of expression `primes` with `SEL-FIRST-PRIME` yields[1]:

```
Maude> red primes .
reduce in SEL-FIRST-PRIMES : primes .
rewrites: 3 in 0ms cpu (10ms real) (~ rewrites/second)
result LNat: cons(s(s(0)), sieve(filter(nats(s(s(s(0)))), s(0), s(0))))
```

However, we can easily obtain the fourth prime number without any risk of nontermination:

```
Maude> red sel(s(s(s(0))),primes) .
reduce in SEL-FIRST-PRIMES : sel(s(s(s(0))), primes) .
rewrites: 28 in -10ms cpu (0ms real) (~ rewrites/second)
result Nat: s(s(s(s(s(s(0))))))
```

Of course, further semantic issues should also be addressed (see [Luc03]). In this paper, we are only concerned with termination.

We argue that termination of programs could be more appropriately studied as *termination of (strategies for) computational restrictions of rewriting*.

Termination of computational restrictions of rewriting is a challenging problem. To motivate this claim, think of Lankford's theorem establishing that termination of TRSs is equivalent to the existence of a well-founded ordering $>$ on terms such that $t > s$ whenever $t \to s$. This is easily generalized to computational restrictions of rewriting $\rightharpoonup$, i.e., orderings on terms are also the basis of termination analysis of computational restrictions of rewriting. In practice, however, we only want to compare the left- and right-hand sides of the rules of the TRS by using some *reduction ordering*, i.e., a stable, monotonic, and well-founded ordering on terms. These properties correspond to well-known properties of the rewriting relation $\to$ that computational restrictions of rewriting do not need to fulfill. For instance, context-sensitive rewriting (*CSR* [Luc98]) is not monotonic; lazy rewriting (*LR* [FKW00]) and on-demand rewriting (*ODR* [Luc01a]) are not stable or monotonic. Thus, reduction orderings are not completely suitable to prove termination of computational restrictions of rewriting in many cases. Some facts and questions arise:

[1] We use version 1.0.5 of Maude interpreter (available at `http://maude.cs.uiuc.edu/current/system/`).

1. Generalizations of existing reduction orderings (e.g., recursive path orderings, polynomial orderings, Knuth-Bendix orderings, etc.) have already been developed in some cases. For instance, [BLR02] extends the recursive path ordering to permit its use with *CSR*; on the othe hand, [GL02b] discusses the use of polynomial orderings for proving termination of *CSR*. Are there other suitable generalizations? Is there a generic methodology for obtaining them for a given (class of) computational restrictions?
2. Transformation techniques can also be helpful here as they are able to transform proofs of termination of computational restrictions of rewriting into proofs of termination of rewriting: see [GM03,Luc02c] for *CSR*; [Luc02b] for *LR*; and [Luc01a] for *ODR*. Is there any generic transformation which could be specialized/simplified in some cases?
3. Regarding strategies, [FGK01] describes a direct technique for directly proving termination of E-strategies and [Luc01b] shows that proofs of termination of (innermost) *CSR* can also be used for that. In [AEGL02] termination of on-demand strategies is addressed.

The presentation will further discuss and exemplify these challenges and their possible solutions.

References

[AEGL02] M. Alpuente, S. Escobar, B. Gramlich, and S. Lucas. Improving On-Demand Strategy Annotations. In Matthias Baaz and Andrei Voronkov, editors, *Proc. 9th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'02*, LNAI 2514:1-18, Springer-Verlag, Berlin, 2002.

[AL02] S. Antoy and S. Lucas. Demandness in rewriting and narrowing. Electronic Notes in Theoretical Computer Science, volume 76. Elsevier Sciences, 2002.

[Ant92] S. Antoy. Definitional Trees. In H. Kirchner and G. Levi, editors, *Proc. of 3rd International Conference on Algebraic and Logic Programming, ALP'92*, LNCS 632:143-157, Springer-Verlag, Berlin, 1992.

[BLR02] C. Borralleras, S. Lucas, and A. Rubio. Recursive Path Orderings can be Context-Sensitive. In A. Voronkov, editor *Proc. of 18th International Conference on Automated Deduction, CADE'02*, LNAI 2392:314-331, Springer-Verlag, Berlin, 2002.

[Der87] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69-115, 1987.

[Eke98] S. Eker. Term Rewriting with Operator Evaluation Strategies. In C. Kirchner and H. Kirchner, editors, *Proc. of 2nd International Workshop on Rewriting Logic and its Applications, WRLA'98*, Electronic Notes in Computer Science, 15(1998):1-20, 1998.

[FGK01] O. Fissore, I. Gnaedig, and H. Kirchner. Induction for termination with local strategies. *Electronic Notes in Theoretical Computer Science*, volume 58(2), 2001.

[FKW00] W. Fokkink, J. Kamperman, and P. Walters. Lazy Rewriting on Eager Machinery. *ACM Transactions on Programming Languages and Systems*, 22(1):45-86, 2000.

[GL02b] B. Gramlich and S. Lucas. Simple termination of context-sensitive rewriting. In B. Fischer and E. Visser, editors, *Proc. of 3rd ACM SIGPLAN Workshop on Rule-Based Programming, RULE'02*, pages 29-41, ACM Press, New York, 2002.

[GM03] J. Giesl and A. Middeldorp. Transformation Techniques for Context-Sensitive Rewrite Systems. *Journal of Functional Programming*, to appear, 2003.

[KdV03] J. Kennaway and F.J. de Vries. Infinitary rewriting. In TeReSe, *Term Rewriting Systems*, Chapter 11. Cambridge University Press, 2003.

[Luc98] S. Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1):1-61, January 1998.

[Luc01a] S. Lucas. Termination of on-demand rewriting and termination of OBJ programs. In *Proc. of 3rd International Conference on Principles and Practice of Declarative Programming, PPDP'01*, pages 82-93, ACM Press, 2001.

[Luc01b] S. Lucas. Termination of Rewriting With Strategy Annotations. In R. Nieuwenhuis and A. Voronkov, editors, *Proc. of 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR'01*, LNAI 2250:669-684, Springer-Verlag, Berlin, 2001.

[Luc02a] S. Lucas. Context-sensitive rewriting strategies. *Information and Computation*, 178(1):293-343, 2002.

[Luc02b] S. Lucas. Lazy Rewriting and Context-Sensitive Rewriting. *Electronic Notes in Theoretical Computer Science*, volume 64, Elsevier, 2002.

[Luc02c] S. Lucas. Termination of (Canonical) Context-Sensitive Rewriting. In S. Tison, editor, *Proc. of 13th International Conference on Rewriting Techniques and Applications, RTA'02*, LNCS 2378:296-310, Springer-Verlag, Berlin, 2002.

[Luc03] S. Lucas. Semantics of programs with strategy annotations. Technical Report DSIC II/08/03, Universidad Politécnica de Valencia, April 2003.

[MR92] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: the language BABEL. *Journal of Logic Programming*, 12:191-223, 1992.

[NO01] M. Nakamura and K. Ogata. The evaluation strategy for head normal form with and without on-demand flags. In K. Futatsugi, editor, *Proc. of 3rd International Workshop on Rewriting Logic and its Applications, WRLA'00, Electronic Notes in Theoretical Computer Science*, volume 36, 17 pages, 2001.

[Pol01] J. van de Pol. Just-in-time: On Strategy Annotations. *Electronic Notes in Theoretical Computer Science*, volume 57, Elsevier, 2001.

[Zan03] H. Zantema. Termination. In TeReSe, *Term Rewriting Systems*, Chapter 6. Cambridge University Press, 2003.

A Case Study on Termination

Stephan Frank[1], Petra Hofstedt[1], and Pierre R. Mai[2]

[1] Technical University of Berlin
{sfrank,ph}@cs.tu-berlin.de
[2] PMSF IT Consulting
pmai@pmsf.de

Abstract. The paper sketches a case study on termination. Using reduction systems we formally describe a system of cooperating constraint solvers. The formal description already separates the definition of cooperation strategies and the termination detection. This allows to theoretically prove termination for many strategies and, furthermore, frees the user implementing his own strategies of keeping track of termination in practice.

1 Cooperative Constraint Solving

In [1] we describe a meta-solver approach which enables the cooperative solving of mixed-domain constraint problems using several specialized solvers, none of which would be able to handle these problems on its own.

A meta-solver coordinates the work of different individual black-box solvers. A global pool contains the constraints of a constraint conjunction which we want to solve. The constraints are taken from the pool and propagated to the individual solvers which are in return requested to provide newly gained information (i.e. constraints) back via the meta-solver to the pool. This communication is handled via the functions *tell* and *project*, *tell* is used by the individual solvers to propagate the received constraints to their stores, while *project* provides information of the participating solvers to be added to the global pool. The overall *tell* – *project* cycle is repeated until either a failure occurs, then the initially given constraint conjunction is unsatisfiable, or the pool is emptied, i.e. the system could not find a contradiction.

The (partial) function *tell* is used to add a constraint c to a store C of a solver. We distinguish three cases: If the solver perceives $c \wedge C$ to be satisfiable and c to follow from C then this *redundant* propagation does not change C. If $c \wedge C$ is perceived to be satisfiable and c is non-redundant, then c is *successfully* propagated yielding a new store C' with $C' \longleftrightarrow c \wedge C$. If the solver finds out that $c \wedge C$ is unsatisfiable, the propagation *fails* and the store C remains unchanged.

Projecting a constraint store C wrt. a set Y of variables (which occur in C) generates a constraint conjunction C' containing information about Y, i.e. an implication of C and where all other variables than those from Y are eliminated.

The description of the operational semantics of our system is done by means of a reduction relation for *overall configurations*. An overall configuration $\mathcal{H} \in \Xi$ consists of a *formal disjunction* $\dot{\bigvee}_{i \in \{1,\ldots,m\}} \mathcal{G}_i$ of configurations. A *configuration* $\mathcal{G} = \mathcal{P} \odot \bigwedge_{\nu \in L} C^\nu$ consists of the pool $\mathcal{P}$ which is a set of constraints subject to

solve and the conjunction $\bigwedge_{\nu \in L} C^\nu$ of constraint stores.[1] In one derivation step configurations $\mathcal{G}_i$ are replaced by overall configurations $\mathcal{HG}_i$:

$$\begin{array}{l} \mathcal{H} = \mathcal{H}_1 \,\dot\vee\, \mathcal{G}_1 \quad \dot\vee \ldots \dot\vee\, \mathcal{H}_i \,\dot\vee\, \mathcal{G}_i \quad \dot\vee \ldots \dot\vee\, \mathcal{H}_m \,\dot\vee\, \mathcal{G}_m \quad \dot\vee\, \mathcal{H}_{m+1} \quad \Longrightarrow \\ \mathcal{H}' = \mathcal{H}_1 \,\dot\vee\, \mathcal{HG}_1 \dot\vee \ldots \dot\vee\, \mathcal{H}_i \,\dot\vee\, \mathcal{HG}_i \dot\vee \ldots \dot\vee\, \mathcal{H}_m \,\dot\vee\, \mathcal{HG}_m \dot\vee\, \mathcal{H}_{m+1} \end{array}$$

Thus, first, we define a derivation relation for configurations $\mathcal{G}_i$ and, based on this, we define a derivation relation for overall configurations $\mathcal{H}$.

Derivation relations for configurations (production level). For a configuration $\mathcal{G} = \mathcal{P} \odot \bigwedge_{\nu \in L} C^\nu$, a derivation step $\mathcal{G} \to \mathcal{HG}$ is defined by the *propagation* of constraints from the pool $\mathcal{P}$ using *tell* and the *projection* of stores C^ν. Influencing the order of projection and propagation and applying choice heuristics on constraints and stores, there are many possibilities to define strategies.

Derivation relations for overall configurations (application level). Different derivation strategies may allow the derivation of exactly one configuration, or of several configurations in parallel or concurrently.

Using this *two-step frame* different reduction systems which realize different derivation strategies for the derivation of an *initial overall configuration* $\mathcal{G}_0 = \mathcal{P} \odot \bigwedge_{\nu \in L} C^\nu$ to normal form can be described. For $\mathcal{G}_0$ the pool $\mathcal{P}$ contains the constraints which we want to solve and all constraint stores are *true*.

A derivation relation $\Longrightarrow$ as defined above can simply yield non-terminating derivations. For example, the projection of a store after a redundant propagation would usually yield the same constraints as a projection immediately before this propagation. To avoid such situations, we allow projection of a store after a successful (non-redundant) propagation only. For this we mark each store by a *tag* to indicate its changes after its last projection: A store initially starts out as *non-dirty*. For each successful propagation, we mark the solver *dirty* indicating that the internal state of the store has changed. Redundant propagations do not change the tag. Projection is only possible for dirty stores, at this, the tag is reset to be non-dirty. A derivation cannot terminate as long as there is a dirty marked store because this store can still be projected.

2 Termination in Theory

The reduction system $(\Xi, \Longrightarrow)$ describes a stepwise propagation of constraints from the pool of the configurations to the associated stores, shrinking at this the stores, and their projection which yield new constraints for propagation.

The domains of constraint solvers are in general not restricted to be finite. This is critical for termination. Therefore we require that the space of valuations of one domain for a finite set of variables can be separated into a finite set of segments and that the solution set of the projection of a store must consist of the union of a number of these segments. This allows to assign weights to stores depending on the number of segments which can be decreased finitely often only.

The second base of the termination proof of the reduction relation $\Longrightarrow$ is the tag handling scheme for stores (see Sect. 1). This ensures that after a successful

[1] L denotes the set of indices of the involved constraint solvers.

propagation the newly derived store is projected at most once and a following projection of this store is only possible after a further change by propagation.

Essentially these two conditions are necessary to show termination for reduction systems $(\Xi, \Longrightarrow)$ defined according to the above two-step frame, i.e. for many solver cooperations with very different cooperation strategies.

3 Termination in Practice

Meta-S [2] is an implementation of the theoretical framework described in Sect. 1. Since Meta-S should offer a strategy language for the user to express problem-specific solving strategies in a simple way it was essential to factor out code common to most strategies. Thus, one central aspect of the implementation was the formulation of algorithm-independent termination conditions, thereby freeing strategy implementors from keeping track of termination conditions themselves.

In order to properly detect termination of the processing loop for a given configuration we need to observe the following set of *termination criteria.* Termination of the solving process of a particular configuration ensures when:

1. None of the solvers are marked dirty (cf. the tag handling scheme, Sect. 1).
2. There remain no pending constraints in the pool.

These conditions ensure that finally all stores are projected and there remain no constraints in the pool for propagation. Thus, regardless of the actions of the strategy, termination will never be premature. Of course it is still possible for strategies to prevent the proper termination, e.g. by going into an infinite loop or by never projecting a certain variable, etc. Since the criteria can be independently tested at any point during the solving process, we "outsource" the termination testing from the solving strategy itself to formulate strategies without explicit checks for termination (see Fig. 1). Thus freed from keeping track of termination, the generic function *strategy-step* represents the place to implement strategies based on the order of constraint propagation and projection and choice heuristics for solvers and constraints (cf. production and application level, Sect. 1). Since individual strategies may employ additional optimizations, further termination checking is allowed to occur through methods on the generic function *strategy-finish*, which can restart the loop when needed.

```
restart:
    invoke strategy-restart-actions
    until pending constraints = ∅∧
          ∀solver:¬dirty(solver)
     do
        invoke strategy-step
    end until
    if strategy-finish returns then
        goto restart
    end if
```

Fig. 1: Strategy loop

References

1. Hofstedt, P.: Cooperation and Coordination of Constraint Solvers. PhD thesis, Technical University of Dresden (2001) Shaker Verlag, Aachen.
2. Frank, S., Hofstedt, P., Mai, P.: Meta-S: A strategy-oriented Meta-Solver Framework. In: Proc. of the 16th Int'l FLAIRS conference, AAAI Press (2003) to appear.

TerminWeb: Termination Analyzer for Logic Programs

Samir Genaim[1] and Michael Codish[2]

[1] Dipartimento di Informatica
Università degli Studi di Verona
Verona, Italy
genaim@sci.univr.it

[2] The Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, Israel
mcodish@cs.bgu.ac.il

Abstract. The TerminWeb analyser is a semantic based termination analyser for logic programs. It is uniqueness in that it is based on a semantics which makes the termination properties observable. Its implementation is obtained as an abstraction of an interpreter which computes the termination semantics of a given program. TerminWeb supports termination *checking* and *inference* with respect to semi-linear norms; *type-based* norms; and a combination of several norms.

1 Overview of TerminWeb

The TerminWeb analyser [8] is a semantic based termination analyser for logic programs, which focuses on universal termination. The analyser is implemented in SICStus Prolog and is available in two versions: on-line via WWW and stand-alone. The early version of TerminWeb analyser was developed, in [2], out of a collaboration with TerminLog [7]. The main contribution of [2] is basing termination analysis on semantics. Namely, providing a semantics which makes the termination properties of a program observable.

The semantics adopted in [2] is called the binary unfolding semantics first presented in [4] where it was shown to make the notion of calls observable. The semantic objects are binary clauses. A binary clause of the form $p(\overline{s}) \leftarrow q(\overline{t})$ indicates that a call to p is eventually followed by a call to q. Note that each pair of subsequent calls $p(\overline{s})$ and $q(\overline{t})$ are represented by at least one binary clause and that the loops in the program are represented by binary clauses of the form $p(\overline{t}) \leftarrow p(\overline{s})$.

Using the binary unfolding semantics, a termination analyser is obtained by approximating this semantics with respect to size and instantiation information. Then, this information is used to verify that each loop (recursive binary clause) has a decreasing measure over a well-founded domain. Note that in contrast to classic techniques, where a single *global* decreasing measure is required for proving termination, TerminWeb requires only a *local* decreasing measure for each loop.

The core of TerminWeb is a simple abstract meta-interpreter which approximates the binary unfolding semantics. The approximation is over a domain to represent size and

instantiation information with respect to a given size function (norm) which measures the size of terms. This abstract meta-interpreter operates in a uniform way over several choices of abstract domains such as monotonicity constraints [7] and convex polyhedra [3, 1]. These are represented as systems of linear constraints. The advantage of using constraints domain to represent size-relation, over using a special structures like in [7, 6], is that different domains with various precision and efficiency can be used, according to the user needs. On the top of this meta-interpreter, TerminWeb has two more components that perform termination *checking* and termination *inference*.

For termination *checking*, the user provides a program P; an initial call pattern G; and chooses a norm to be used in the analysis. Then, the analyser checks if the program terminates for the given input class as follows: (1) The set of abstract binary clauses, $[\![P]\!]^{bin}_{size}$, over a size-relations domain is computed; (2) The set of call patterns, $[\![P^G]\!]^{calls}_{gr}$, with respect to the initial goal is computed; and (3) for each call pattern $\kappa \in [\![P^G]\!]^{calls}_{gr}$ and matching recursive binary clause $p(\bar{x}) \leftarrow p(\bar{y})$, the analyser verifies that some measure f on the sizes of some of the sufficiently instantiated arguments in κ decrease, i.e $f(p(\bar{x})) > f(p(\bar{y}))$. This ensures that each loop is finitely executed.

For termination *inference*, the user provides a program and chooses a size function (norm) to be used in the analysis. Then, the analyser infers a set of call patterns for which termination is guaranteed. For example, for the classic $append/3$ relation and the *list-length* norm, termination inference results in $append(x,y,z) \leftarrow x \vee z$ with the interpretation that the query $append(x,y,z)$ terminates, if x or z are bound to closed lists. Termination inference in TerminWeb is based on an observation that links termination checking and termination inference through *backwards analysis* as presented in [5]. Backwards analysis is concerned with the following type of question: Given a program and assertions at given program points, what are the weakest requirements on the inputs to the program which guarantee that the assertions will hold whenever the execution reaches these points. For termination inference, these assertions are extracted from the program loops (binary clauses) and specified to guarantee the rigidity of at least one decreasing set of arguments in each loop.

Termination *inference* in TerminWeb is performed as follows: (1) The set of binary clause is computed exactly as in termination checking; (2) assertions that guarantee the rigidity of at least one decreasing set of arguments in each binary clause is extracted; and (3) these assertions are passed together with the program to the backwards analysis engine, which infers the required modes. The use of a standard framework for backwards analysis provides a formal justification for termination inference and leads to a simple and efficient implementation. This also facilitates a formal comparison of termination checking and inference.

The task of selecting a norm is the first step in many termination analysers. Once a norm is selected size and rigidity analyses are performed with respect to the selected norm. This step is crucial for proving termination as the analyser may succeed to obtain a termination proof using one norm and fail using others. Usually, a termination analyser provides a predefined set of semi-linear norms from which the user can select a norm as well as tools for defining a norm tailored for the specific data structures used in a program.

TerminWeb supports also *type-based* norms, derived automatically from declared (or inferred) types, as well as proofs based on the combination of several norms.

References

1. F. Benoy and A. King. Inferring argument size relationships with CLP(R). In J. Gallagher, editor, *Logic Program Synthesis and Transformation*, volume 1207 of *Lecture Notes in Computer Science*, pages 204–223. Springer, 1997.
2. MicHael Codish and Cohavit Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.
3. Patrick Cousot and Nicholas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, January 1978.
4. M. Gabbrielli and R. Giacobazzi. Goal independency and call patterns in the analysis of logic programs. In *Proceedings of the Ninth ACM Symposium on Applied Computing*, pages 394–399. ACM Press, 1994.
5. A. King and L. Lu. A Backward Analysis for Constraint Logic Programs. *Theory and Practice of Logic Programming*, page 32, July 2002.
6. Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. *ACM SIGPLAN Notices*, 36(3):81–92, 2001.
7. N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 63–77, Leuven, Belgium, 1997. The MIT Press.
8. Cohavit Taboch, Samir Genaim, and Michael Codish. Terminweb: A semantic based termination analyser for logic programs, 2002. `http://www.cs.bgu.ac.il/~mcodish/TerminWeb`.

Hasta-La-Vista: Termination Analyser for Logic Programs

Alexander Serebrenik, Danny De Schreye

Department of Computer Science, K.U. Leuven
Celestijnenlaan 200A, B-3001, Heverlee, Belgium
E-mail: {Alexander.Serebrenik,Danny.DeSchreye}@cs.kuleuven.ac.be

Verifying termination is often considered as one of the most important aspects of program verification. Logic languages, allowing us to program declaratively, increase the danger of non-termination. Therefore, termination analysis received considerable attention in logic programming (see e.g. [7, 8, 10, 16]). Unfortunately, the majority of existing termination analysers, such as TermiLog [15], TerminWeb [7], and cTI [16] are restricted to pure logic programs and thus, leave many interesting real-world examples out of consideration. Therefore, in order to abridge the gap between programming practice and existing termination analysers real-world programming techniques should be considered.

In this paper we present Hasta-La-Vista—a powerful tool for analysing termination of logic programs with integer computations. Hasta-La-Vista extends the constraints-based approach of Decorte *et al.* [9] by integrating the inference algorithm of [19]. Moreover, as explained in [19], in the integer case our approach is not limited to proving termination, but can also infer termination, i.e., find the set of queries terminating for a given program.

System architecture. Conceptually, Hasta-La-Vista consists of three main parts: transformation, constraints generation and constraints solving. As a preliminary step, given a program and a set of atomic queries, type analysis of Janssens and Bruynooghe [14] computes the call set. We opted for a very simple type inference technique that provides us only with information whether some argument is integer or not.

Based on the results of the type analysis the system decides whether termination of the given program can be dependent on the integer computation. In this case, the *adorning* transformation is applied [19]. The aim of the transformation is to discover bounded integer arguments and to make the bounds explicit. Intuitively, if a variable x is known to be bounded from above by n, then $n - x$ is always positive and thus, can be used as a basis for a definition of a *level-mapping* (a function from the set of atoms to the naturals). In order to prove termination we have to show that the level-mapping decreases while traversing a clause. This requirement can be translated into a set of constraints. Finally, this set of constraints is solved and depending on the solution termination is

reported for all queries or for some queries or possibility of non-termination is suspected.

Experimental evaluation. Hasta-La-Vista has been applied to more than 90 examples. In 95% of the cases the analysis was either powerful enough to prove termination for terminating computations, or justly suspected possibility of non-termination. Results of the experimental evaluation are summarised in Table 1. Times were measured on Intel®Pentium®4 with 1.60GHz CPU and 260M memory, running 2.4.20-pre11 Linux. The core part of the implementation was done in SICStus Prolog [20], using its CLP(FD) [5] and CLP(Q) [12] libraries. Type inference of Janssens and Bruynooghe [14] was implemented in MasterProLog [13].

Table 1. Experimental evaluation

Reference	**Number of examples**	**Success rate**	**Maximal time**
Symbolical computations			
Apt and Pedreschi [2]	12	100%	0.05
De Schreye and Decorte [8]	7	85%	0.01
Plümer [18]	31	90%	0.09
Integer computations			
Apt [1], chapter 9	13	100%	0.08
Sterling and Shapiro [21], chapter 8	14	100%	0.02
Various	19	100%	0.27
Total	92	95%	0.24

Hasta-La-Vista turned out to be robust enough to prove termination of such examples as Euler's totient function [11], depth-limited depth-first search [3] and finding all prime numbers up to a given limit [6]. Programs denoted as "various" were collected both from different Prolog textbooks [3, 6, 17] and from Prolog programs collections [4, 11].

Conclusion. We have presented Hasta-La-Vista, a termination analyser for logic programs. To the best of our knowledge, this analyser is unique in being able to prove termination of programs depending on integer computations.

Acknowledgement. We are very grateful to Gerda Janssens for making her type analysis system available for us.

References

1. K. R. Apt. *From Logic Programming to Prolog.* Prentice-Hall International Series in Computer Science. Prentice Hall, 1997.

2. K. R. Apt and D. Pedreschi. Modular termination proofs for logic and pure Prolog programs. In G. Levi, editor, *Advances in Logic Programming Theory*, pages 183–229. Oxford University Press, 1994.
3. I. Bratko. *Prolog programming for Artificial Intelligence*. Addison-Wesley, 1986.
4. F. Bueno, M. J. García de la Banda, and M. V. Hermenegildo. Effectiveness of global analysis in strict independence-based automatic parallelization. In M. Bruynooghe, editor, *Logic Programming, Proceedings of the 1994 International Symposium*, pages 320–336. MIT Press, 1994.
5. M. Carlsson, G. Ottoson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. H. Hartel, and H. Kuchen, editors, *Programming Languages: Implementations, Logics, and Programs, 9th International Symposium, PLILP'97, Including a Special Track on Declarative Programming Languages in Education, Southampton, UK, September 3-5, 1997, Proceedings*, volume 1292 of *Lecture Notes in Computer Science*, pages 191–206. Springer Verlag, 1997.
6. W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer Verlag, 1981.
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, May/July 1994.
9. S. Decorte, D. De Schreye, and H. Vandecasteele. Constraint-based termination analysis of logic programs. *ACM TOPLAS*, 21(6):1137–1195, November 1999.
10. N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing*, 12(1-2):117–156, 2001.
11. W. Hett. P-99: Ninety-nine Prolog problems. Available at `http://www.hta-bi.bfh.ch/~hew/informatik3/prolog/p-99/`, July 2001.
12. C. Holzbaur. OFAI CLP(Q,R) Manual. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna, 1995.
13. IT Masters. MasterProLog Programming Environment. Available at `http://www.itmasters.com/`, 2000.
14. G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2&3):205–258, July 1992.
15. N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. *TermiLog*: A system for checking termination of queries to logic programs. In O. Grumberg, editor, *Computer Aided Verification, 9th International Conference*, volume 1254 of *Lecture Notes in Computer Science*, pages 63–77. Springer Verlag, June 1997.
16. F. Mesnard and U. Neumerkel. Applying static analysis techniques for inferring termination conditions of logic programs. In P. Cousot, editor, *Static Analysis, 8th International Symposium, SAS 2001*, volume 2126 of *Lecture Notes in Computer Science*, pages 93–110. Springer Verlag, 2001.
17. R. A. O'Keefe. *The Craft of Prolog*. MIT Press, 1990.
18. L. Plümer. *Termination Proofs for Logic Programs*, volume 446 of *Lecture Notes in Computer Science*. Springer Verlag, 1990.
19. A. Serebrenik and D. De Schreye. Inference of termination conditions for numerical loops in Prolog. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conferencerence, Proceedings*, volume 2250 of *Lecture Notes in Computer Science*, pages 654–668. Springer Verlag, 2001.
20. SICS. *SICStus User Manual. Version 3.10.0*. Swedish Institute of Computer Science, 2002.
21. L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, 1994.

Tsukuba Termination Tool*

Nao Hirokawa and Aart Middeldorp

University of Tsukuba, Tsukuba 305-8573, Japan

We present a tool for automatically proving termination of first-order rewrite systems. The tool is based on the dependency pair method of Arts and Giesl [1]. It incorporates several new ideas that make the method more efficient. The tool produces high-quality output and has a convenient web interface. If T$_{\text{T}}$T succeeds in proving termination, it outputs a proof script which explains in considerable detail how termination was proved. This script is available in both HTML and LaTeX format. In the latter, the approximated dependency graph is visualized using the *dot* tool of the Graphviz toolkit. T$_{\text{T}}$T is written in Objective Caml. We tested the various options of T$_{\text{T}}$T on numerous examples. The results, as well as a comparison with other tools that implement the dependency pair method and some implementation details, can be found in [2, 3].

We describe some of the features of the tool (T$_{\text{T}}$T in the sequel) by means of its web interface, displayed in Fig. 1.

TRS The user inputs a TRS by typing the rules into the upper right text area or by uploading a file via the browse button. The exact input format is obtained by clicking the TRS link.

Comment and Bibtex Anything typed into the upper right text area will appear as a footnote in the generated LaTeX code. This is useful to identify TRSs. LaTeX `\cite` commands may be included. In order for this to work correctly, a corresponding bibtex entry should be supplied. This can be done by typing the entry into the appropriate text area or by uploading an appropriate bibtex file via the browse button.

Base Order The current version of T$_{\text{T}}$T supports the following three base orders: LPO with strict precedence, LPO with quasi-precedence, and KBO with strict precedence.

Dependency Pairs T$_{\text{T}}$T supports the basic features of the dependency pair technique (argument filtering, dependency graph, cycle analysis) described below. Advanced features like narrowing, rewriting, and instantiation are not yet available. Also innermost termination analysis is not yet implemented.

* `http://www.score.is.tsukuba.ac.jp/ttt/`

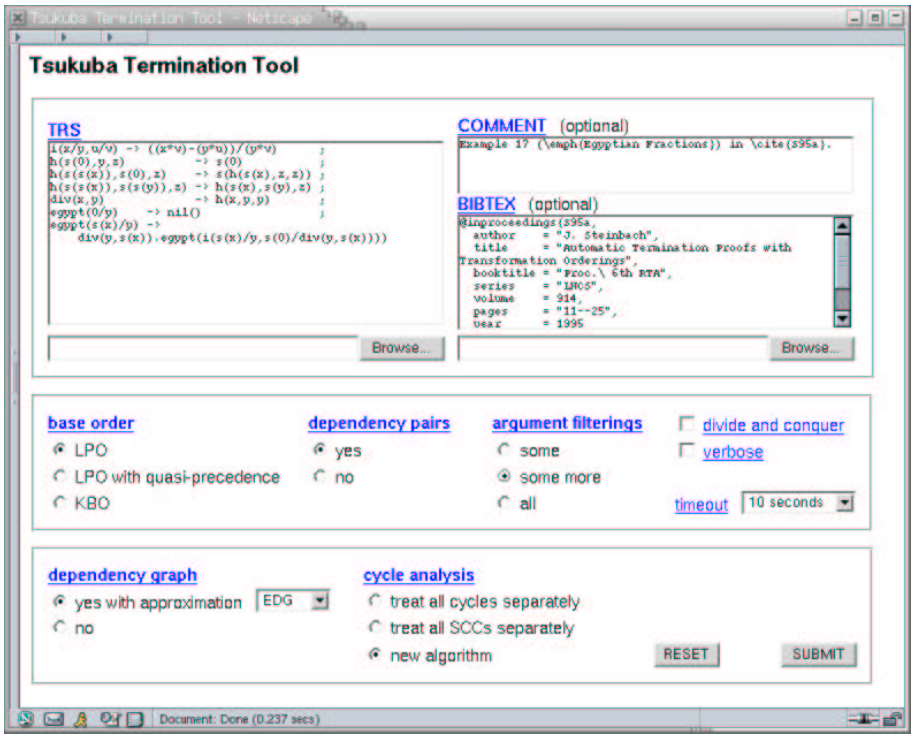


Fig. 1. A screen shot of the web interface of TTT.

Argument Filtering A single function symbol f of arity n gives rise to $2^n + n$ different argument filterings (AFs): $f(x_1, \ldots, x_n) \to f(x_{i_1}, \ldots, x_{i_m})$ for all $1 \leqslant i_1 < \cdots < i_m \leqslant n$ and $f(x_1, \ldots, x_n) \to x_i$ for all $1 \leqslant i \leqslant n$. A moment's thought reveals that even for relatively small signatures, the number of possible AFs is huge. TTT supports two simple heuristics to reduce this number. The *some* option considers for a function symbol f of arity n only the 'full' AF $f(x_1, \ldots, x_n) \to f(x_1, \ldots, x_n)$ and the n 'collapsing' AFs $f(x_1, \ldots, x_n) \to x_i$ $(1 \leqslant i \leqslant n)$. The *some more* option considers in addition the argument filtering $f(x_1, \ldots, x_n) \to f$ (when $n > 0$).

Dependency Graph The dependency graph determines the ordering constraints that have to be solved in order to guarantee termination. Since the dependency graph is in general not computable, a decidable approximation has to be adopted. The current version of TTT supplies two such approximations: EDG is the original estimation of Arts and Giesl; EDG$*$ is an improved version of EDG described in [4, latter half of Section 6].

Cycle Analysis Once an approximation of the dependency graph has been computed, some kind of cycle analysis is required to generate the actual ordering

constraints. T$_\text{T}$T offers three different methods:

1. The method described in the literature is to treat cycles in the approximated dependency graph separately. For every cycle $\mathcal{C}$, the dependency pairs in $\mathcal{C}$ and the rewrite rules of the given TRS must be weakly decreasing and at least one dependency pair in $\mathcal{C}$ must be strictly decreasing (with respect to some AF and base order).
2. Another method is to treat all strongly connected components (SCCs) separately. For every SCC $\mathcal{S}$, the dependency pairs in $\mathcal{S}$ must be strictly decreasing and the rewrite rules of the given TRS must be weakly decreasing. Treating SCCs rather than cycles separately improves the efficiency at the expense of reduced termination proving power.
3. The third method available in T$_\text{T}$T combines the termination proving power of the cycle method with the efficiency of the SCC method. It is described in [2].

Divide and Conquer The default option to find a suitable AF that enables a group of ordering constraints to be solved by the selected base order is *enumeration*, which can be very inefficient, especially for larger TRSs where the number of suitable AFs is small. Setting the *divide and conquer* option computes all suitable AFs for each constraint separately and subsequently merges them to obtain the solutions of the full set of constraints. This can (greatly) reduce the execution time at the expense of an increased memory consumption. The divide and conquer option is described in detail in [2]. At the moment of writing it is only available in combination with LPO.

Verbose Setting the verbose option generates more proof details. In combination with the divide and conquer option described above, the total number of AFs that enable the successive ordering constraints to be solved are displayed during the termination proving process.

Timeout Every combination of options results in a finite search space for finding termination proofs. However, since it can take days to fully explore the search space, (the web version of) T$_\text{T}$T puts a strong upper bound on the permitted execution time.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *TCS*, 236:133–178, 2000.
2. N. Hirokawa and A. Middeldorp. Automating the depencency pair method. In *Proc. 19th CADE*, LNAI, 2003. To appear.
3. N. Hirokawa and A. Middeldorp. Tsukuba termination tool. In *Proc. 14th RTA*, LNCS, 2003. To appear.
4. A. Middeldorp. Approximations for strategies and termination. In *Proc. 2nd WRS*, volume 70(6) of *ENTCS*, 2002.

Termptation

Cristina Borralleras[1] and Albert Rubio[2]

[1] Universitat de Vic, Spain
Email: cristina.borralleras@uvic.es
[2] Universitat Politècnica de Catalunya, Barcelona, SPAIN
Email: rubio@lsi.upc.es

TERMPTATION (TERMination Proof Techniques automATION) is a fully automated system for proving termination of term rewrite systems (TRS) which is based on the *monotonic semantic path ordering* method (MSPO) [BFR00] and its translation to an ordering constraint solving problem [BR03], called the MSPO-constraint method (see [Bor03] for a detailed description). It is available at `//www.lsi.upc.es/~albert/`.

The system has been implemented in Prolog and has three main steps. First it tries to ease the termination proof of the TRS by applying modularity results: disjoint unions, constructor-sharing unions and, in some cases, hierarchical unions of sets of rules. Once this splitting is made, for every subsystem we apply the MSPO-constraint method to obtain the a disjunction of ordering constraints. Finally the system tries to find a satisfiable constraint in the disjunction.

1 Options of the system

TERMPTATION has four parameters to be set:

– *The prover*.
The user can choose among five provers. Prover 5 is an implementation of the RPO. Provers 1 – 4 are implementations of the MSPO-constraint method which differ in

- the way the constraints are generated: by groups in provers 1 and 3, which chooses in a clever way the constraints to be tried first, and one by one in provers 2 and 4;
- whether modularity for hierarchical unions are applied or not. Provers 1 and 2 do not apply such modularity results while provers 3 and 4 do. Note that, if this modularity reuslt are applied then only a subset of the constraints generated by the MSPO-constraint method can be used.

In general, provers 2 and 4 work better for TRS's with large size rules (since in that case generating the constraints by groups can be very expensive in time and space), and provers 3 and 4 are recomended for TRS's with a large number of rules. The default option is prover 1 which treats the constraints by groups and it does not apply modularity result for hierarchical unions.

– *Standard or innermost rewriting*.
The system can be used for proving termination of rewriting or termination of innermost rewriting. Note that for innermost rewriting more powerful constraint solving techniques can be applied.

– *Special treatment of arguments*.
  By default the system applies the definition of the MSPO using the semantic path ordering (SPO) with a multiset comparison of arguments. If a special treatment of arguments is chosen then the system uses the lexicographic semantic path ordering (SPO-lex) with a lexicographic comparison of the arguments.
– *Deeper analysis*.
  If this option is activated then the system uses more powerful techniques to analyze each constraint that is considered. Therefore, constraints that are considered unsolvable without the deeper analysis can become solvable if this option is activated. Due to this, if the system fails in both cases then, of course, the system with the deeper analysis takes more time to produce the answer.
  On the other hand, if the system succeeds in both cases, since they may found different solutions, it is not clear which one will be faster.

2 The Output

The output provides part of the explanation about the proof of termination found by the system.

By now, except if we have used prover 5 (which is only RPO), in many cases part of the output can only be followed by users knowing about MSPO and its translation to constraints and with a deep knowledge about the applied constraint handling methods (some of them are similar to the ones used by the dependency pair method [AG00]). The provided information is intended to allow expert users to reconstruct the termination proof by hand.

3 Some implementation details

Our method basically tries to prove the satisfiability of one of the constaints generated. Since there can be many of those constraints, the system tries to consider first the ones that look simpler to be solved.

The techniques that are used for solving the obtained constraints are based on the analysis of the dependency graph [AG00] and some original techniques to check the potential cycles. After analayzing the graph and its potential cycles, the resulting ordering constraints are finally solved using term interpretations and the recursive path ordering (RPO).

3.1 Generating term interpretations and RPO

The system considers two different kinds of interpretations, projections and selections of arguments (so called argument filtering interpretations). The set of possible interpretations is finite, but it is in general very large. Our process consists of proving or discarding possible interpretations for each symbol. If all the interpretations for a symbol are discarded then the process fails and backtracks.

Some heuristics are used to choose which symbols are considered first for deciding its interpretation. These decisions are taken be analyzing the ordering restrictions that

have to be fulfiled. After any intermediate decision in the process, the system discards all remaining interpretations that are incompatible with the constraint.

Additionally, while producing the interpretations we keep track of all forced decisions on the precedence and status of the function symbols for the RPO. For instance, if in a particular state the interpretation of $g(x)$ is either g or $g(x)$ and we have a condition $a > g(b)$ in the constraint then the precedence should include $a \succ_F g$. If a latter decision on the interpretation forces $g \succeq_F a$ in the precedence, the process fails and backtracks.

Finally, the automatic generation of the RPO is implemented following the Davis-Putnam style for solving SAT [DLL62]. In general, the system takes first those decisions for which there are less alternatives to consider. After every decision, the system applies (propagates) all forced decisions about the precedence and the status for the function symbols and simplifies the remaining set of ordering conditions with respect to such decisions.

References

[AG00] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.

[BFR00] C. Borralleras, M. Ferreira, and A. Rubio. Complete monotonic semantic path orderings. *Proc. of the 17th International Conference on Automated Deduction*, LNAI 1831:346–364, Pittsburgh, USA, 2000. Springer-Verlag.

[Bor03] C. Borralleras. Ordering-based methods for proving termination automatically. PhD thesis, Universitat Politècnica de Catalunya, Dept. LSI., 2003. Available at `//www.lsi.upc.es/~albert/cristinaphd.ps`.

[BR03] C. Borralleras and A. Rubio. Proving Termination by the Reduction Constraint Framework. *Proc. of the 6th International Workshop on Termination*, Valencia, Spain, 2003.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. In *Communications of the ACM*, 5(7):394–397, July 1962.

The SCT Analyser

An analysis tool based on size-change termination

Arne John Glenstrup

DIKU, University of Copenhagen
panic@diku.dk
http://www.diku.dk/topps/Research.html#Products

Abstract. The Size-Change Termination Analyser is a prototype tool for identifying potentially non-terminating loops, or for guaranteeing termination, in a given first-order functional program. The size-change termination principle is general, allowing subject programs to be written using mutual and general recursion, yet the analyser is able to detect termination in many programs. The analyser is implemented with a publicly available web interface for demonstration purposes.

1 Background

The Size-Change Termination (SCT) Analyser is a prototype tool for identifying all potentially non-terminating loops in a given program, as part of the program's verification. It has been jointly developed by Neil D. Jones, Amir Ben-Amram and Chin Soon Lee at TOPPS, DIKU [1]. The web implementation is due to Carl C. Frederiksen [2].

2 Size-change termination

In an size-change terminating program, all infinite call sequences must contain some argument values that are never increased from one call to the next, and are decreased infinitely often. We assume function arguments range over a well-founded domain, so for a size-change terminating program infinite call sequences are impossible, implying termination under normal evaluation for any input.

Although this seems like a rather simple approach, as the value of tests in conditional expressions are not considered, it can detect termination of many programs, including Ackermann's function [2]. The method is general: it does not restrict subject programs to primitive recursion, does not rely on lexicographical ordering of argument values, and handles mutual recursion without any special treatment. The set of size-change terminating functions is large: it has been shown to be identical to Péter's multiple recursive functions [3, 4].

3 The analyser tool

The SCT Analyser takes as input a subject program written in the first-order functional language shown in Fig. 1, and displays as output some function call loops that might not terminate. If no such loops are detected, the program is guaranteed to terminate.

The analyser works in 4 steps:

1. First the size of the return values of functions, relative to the sizes of their input arguments, are approximated.
2. Based on these approximations, a size-change graph (SCG) is generated for each call site.
3. The closure of the set of SCGs under composition is computed.
4. The closure set is checked: each idempotent SCG in this set must have an in situ decreasing parameter $x \xrightarrow{\downarrow} x$.

Any SCG G in Step 4 without an in situ decreasing parameter represents one or more *critical multipaths*, that is, the list of call sites for the SCGs that were composed in Step 3 to yield G.

A web based user interface for the SCT Analyser is publicly available at the DIKU TOPPS pages for demonstration purposes. The user can enter a program or select an example program, and when the program has been analysed, the results are displayed in the browser as shown in Fig. 2. Output diagnostics include SCGs generated for individual call sites, critical multipaths, and the SCG set closure.

4 Further reading

The size-change termination principle was originally published at POPL [1] and is discussed in depth in Lee's PhD Dissertation [5]. Details on the SCT Analyser implementation are described in Frederiksen's Master's Thesis [2].

$$
\begin{array}{rcl}
\mathit{Prog} & ::= & \mathit{Def}_1 \ldots \mathit{Def}_n \\
\mathit{Def} & ::= & \mathit{fn}(x_1, \ldots, x_n)\ \texttt{=}\ \mathit{Expr}^{\mathit{fn}} \\
\mathit{Expr} & ::= & x \\
& | & \mathit{con} \\
& | & \mathit{con}(\mathit{Expr}_1, \ldots, \mathit{Expr}_n) \\
& | & \mathit{des}(\mathit{Expr}) \\
& | & \texttt{if}\ \mathit{Expr}_1\ \texttt{then}\ \mathit{Expr}_2\ \texttt{else}\ \mathit{Expr}_3 \\
& | & \texttt{let}\ x_1\ \texttt{=}\ \mathit{Expr}_1 \ldots x_n\ \texttt{=}\ \mathit{Expr}_n\ \texttt{in}\ \mathit{Expr}_0 \\
& | & \mathit{fn}(x_1, \ldots, x_n) \\
& | & \mathit{op}(x_1, \ldots, x_n) \\
x & := & \text{identifier beginning with lower case} \\
\mathit{fn} & := & \text{identifier beginning with lower case, not in } \mathit{op} \\
\mathit{con} & := & \text{capitalized identifier} \\
\mathit{des} & := & \{\texttt{1st}, \texttt{2nd}, \texttt{3rd}, \ldots\} \\
\mathit{op} & := & \text{primitive operator } \{\texttt{eq}, \texttt{equal}, \ldots\}
\end{array}
$$

Fig. 1. First-order functional language syntax treated by the analyser

References

1. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: ACM Symposium on Principles of Programming Languages. Volume 28., New York, ACM Press (2001) 81–92

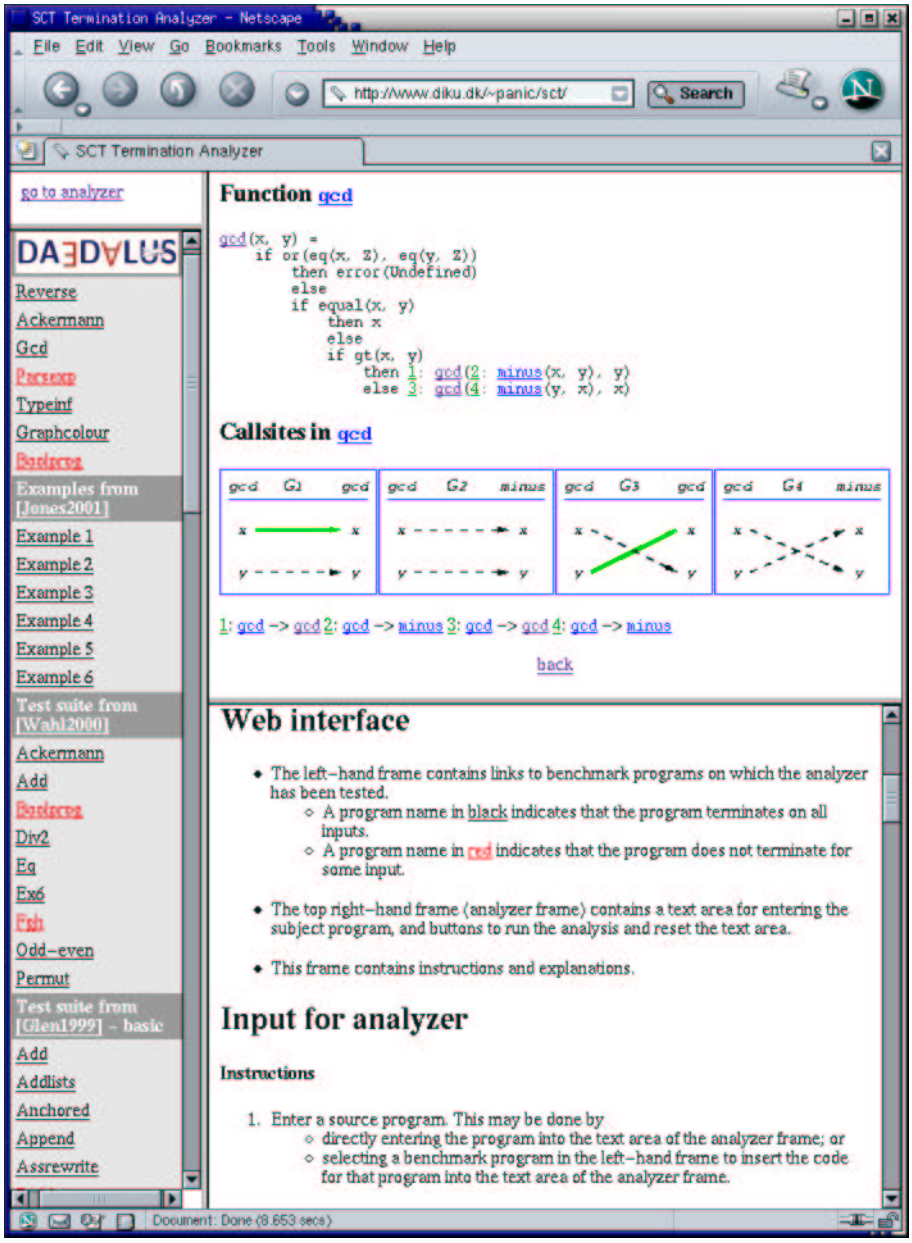


Fig. 2. Screenshot of the SCT Analyser web interface

2. Frederiksen, C.C.: Automatic runtime analysis for first order functional programs. Master's thesis, DIKU, University of Copenhagen, Denmark (2002)
3. Ben-Amram, A.M.: General size-change termination and lexicographic descent. In Mogensen, T., Schmidt, D., Sudborough, I.H., eds.: The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones. Volume 2566 of Lecture Notes in Computer Science. Springer-Verlag, Berlin (2002) 3–17
4. Péter, R.: Rekursive Funktionen (Recursive Functions). Académiai Kiadó, Budapest (1951 (1976)) (Academic Press, New York).
5. Lee, C.S.: Program Termination Analysis and Termination of Offline Partial Evaluation. PhD thesis, University of Western Australia (2002)

AProVE: A System for Proving Termination

Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke

LuFG Informatik II, RWTH Aachen, Ahornstr. 55, 52074 Aachen, Germany
{giesl|thiemann}@informatik.rwth-aachen.de
{nowonder|spf}@i2.informatik.rwth-aachen.de

1 Introduction

The system AProVE (Automated Program Verification Environment) can be used for automated termination and innermost termination proofs of (conditional) term rewrite systems (TRSs). AProVE currently offers the techniques of *recursive path orders* possibly with status [3] (Sect. 2), *dependency pairs* including recent refinements such as *narrowing*, *rewriting*, and *instantiation* of dependency pairs [1, 4, 5] (Sect. 3), and the *size-change principle*, also in combination with dependency pairs [7, 9] (Sect. 4). The tool is written in Java and proofs can be performed both in a fully automated or in an interactive mode via a graphical user interface (Sect. 5)

2 Direct Termination Proofs

In a direct termination proof, the system tries to find a reduction order such that all rules of a TRS are decreasing. Currently, the following *path orders* are implemented in AProVE: the embedding order (`EMB`), the lexicographic path order (`LPO`, [6]), the `LPO` with status which compares subterms lexicographically according to arbitrary permutations (`LPOS`), the recursive path order which compares subterms as multisets (`RPO`, [3]), and the `RPO` with status which is a combination of `LPOS` and `RPO` (`RPOS`).

Path orders may be parameterized by a precedence on function symbols and a status which determines how the arguments of function symbols are compared. To explore the search space for these parameters, the system leaves the precedence and the status as unspecified (or "*minimal*") as possible. The user can decide whether to explore the search space in a depth-first or a breadth-first search (where in the latter case, *all* possibilities for a minimal precedence and status are computed which satisfy the current constraints). Moreover, the user can configure the path orders by deciding whether different function symbols may be equivalent according to the precedence used in the path order ("`non-strict precedence`"). It is also possible to restrict potential equivalences to certain pairs of function symbols. When attempting termination proofs with path orders in AProVE, the precedence found by the system is displayed as a graph (in case of success) and in case of failure in the breadth-first search, the system indicates the problematic constraint.

3 Termination Proofs With Dependency Pairs

The *dependency pair* approach [1, 4, 5] increases the power of automated termination analysis significantly, since it permits the application of simplification orders for non-simply terminating TRSs. In AProVE, the user can select whether to use the dependency pair approach for termination or for innermost termination proofs. The system can also check whether a TRS is non-overlapping (then innermost termination already implies termination). Essentially, the dependency pair approach generates a set of constraints from the TRS and searches for a well-founded order satisfying them. For that purpose, the user can select any of the base orders from Sect. 2.

However, while all these orders are *strongly* monotonic, the dependency pair approach only requires *weak* monotonicity. For that reason, before searching for a suitable

order, some of the arguments of the function symbols in the constraints can be eliminated using an *argument filtering* [1]. In AProVE, several techniques are implemented to search for suitable argument filterings. They rely on the idea to process the constraints one by one to reduce the number of possible argument filterings which satisfy the constraints regarded so far.

(Innermost) termination proofs with dependency pairs can be performed in a modular way by constructing an estimated (innermost) dependency graph and by regarding its cycles separately [1, 5]. For each cycle, only one dependency pair must be strictly decreasing, whereas all others only have to be weakly decreasing. AProVE always starts with examining maximal cycles (SCCs), since afterwards, only cycles of those dependency pairs have to be regarded which were not already strictly decreasing in the maximal cycle. To inspect estimated (innermost) dependency graphs, they can be displayed in a special "`Graph`"-window.

To increase the power of the dependency pair technique, in [1, 4] three different transformation techniques were suggested which transform a dependency pair into several new pairs: *narrowing*, *rewriting*, and *instantiation*. These transformations are often crucial for the success of the proof, but in general, they may be applicable infinitely often. AProVE automatically applies these transformations in "safe" cases where their application is guaranteed to terminate. To permit an application of these transformations in other cases as well, the user has to specify a limit for each transformation which determines how often this transformation may be applied to each dependency pair in "unsafe" cases. Then, whenever a proof attempt fails, the current dependency pairs are transformed and the proof is re-attempted again.

In addition to the fully automated mode, (innermost) termination proofs with dependency pairs can also be performed in an interactive mode. Here, the user can specify which narrowing, rewriting, and instantiation steps should be performed and for any cycle or SCC, the user can determine (parts of) the argument filtering, the base order, and the dependency pair which should be strictly decreasing. Moreover, one can immediately see the constraints resulting from such selections, such that interactive termination proofs are supported in a very comfortable way. This mode is intended for the development of new heuristics as well as for the machine-assisted proof of particularly challenging examples.

4 Termination Proofs with the Size-Change Principle

A new *size-change principle* for termination of functional programs was presented in [7] and this principle was extended to TRSs in [9]. In AProVE, the technique of [9, Thm. 11] for termination of TRSs is implemented, where we use the embedding order as underlying base order.[1] AProVE displays all size-change graphs as well as all maximal multigraphs (in case of success) or one critical maximal multigraph without a decreasing edge $i \overset{\succeq}{\rightarrow} i$ (in case of failure).

AProVE also contains the new approach of [9, Thm. 18] which combines the size-change principle with dependency pairs in order to prove innermost termination. This combined approach has the advantage that it often succeeds with much simpler argument filterings and base orders than the pure dependency pair approach. For each SCC $\mathcal{P}$ of the (estimated) innermost dependency graph, let $\mathcal{C}_\mathcal{P}$ be the constructors in $\mathcal{P}$ and let $\mathcal{D}_\mathcal{P}$ be a subset of the defined symbols in $\mathcal{P}$. Then the system builds the size-change graphs and the maximal multigraphs resulting from $\mathcal{P}$ using an argument filtering and the embedding order on $\mathcal{C}_\mathcal{P} \cup \mathcal{D}_\mathcal{P}$. Again, all these multigraphs must have an edge $i \overset{\succ}{\rightarrow} i$ and in case of success, the system displays them all. Next, the argument filtering must be extended such that all usable rules for the symbols $\mathcal{D}_\mathcal{P}$ are weakly decreasing w.r.t.

[1] As shown in [9], only very restricted base orders are sound in connection with the size-change principle. In addition to the results in [9], it is sound to use the full embedding order, where $f(\ldots, x_i, \ldots) \succ x_i$ also holds for defined function symbols f.

the selected base order. Here, the *usable rules* for a symbol f are the f-rules together with the usable rules for all defined symbols occurring in right-hand sides of f-rules. For reasons of efficiency, the user can give a limit on the maximal size of $\mathcal{D}_\mathcal{P}$ and one can restrict the number of symbols which may be argument-filtered.

In case of failure for some SCC, the dependency pairs are transformed by narrowing, rewriting, or instantiation and the proof attempt is re-started. If the limits for the transformations are reached, then the pure dependency pair method is tried, if the user has selected the "`hybrid`" algorithm. In this way, the combined dependency pair and size-change method can be used as a very fast technique which is checked first for every SCC. Only if this method fails, the ordinary dependency pair approach is used on this SCC. For example, when choosing `LPO` with "`non-strict precedence`" as underlying base order, 103 of the 110 examples in the collection of [2] can be proved innermost terminating fully automatically. Most of these proofs take less than a second and the longest proof takes about 20 seconds. The remaining 7 examples in [2] only fail because of the underlying reduction pair (they would require polynomial or Knuth-Bendix orders).

5 Running AProVE

In an AProVE-session, one can load files containing (possibly conditional) TRSs. Conditional TRSs are transformed into unconditional TRSs according to the technique of [4, 8] to prove their (innermost) quasi-decreasingness. When performing termination proofs, a "`system log`" can be inspected to examine all (possibly failed) proof attempts. The results of the termination proof are displayed in `html`-format and both the results and the log information may be saved to a file. Any termination proof attempt may of course be interrupted by a "`Stop`"-button. Instead of running the system on only one term rewrite system, it is also possible to run it on collections and directories of examples in a `batch mode`. In this case, apart from the information on the termination proofs of the separate examples, the "`result`" also contains statistics on the success and the runtime for the examples in the collection. For more details on AProVE and to download the system, the reader is referred to the AProVE web-site `http://www-i2.informatik.rwth-aachen.de/AProVE`.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
2. T. Arts and J. Giesl. A collection of examples for termination of term rewriting using dependency pairs. Technical Report AIB-2001-09[2], RWTH Aachen, Germany, 2001.
3. N. Dershowitz. Termination of rewriting. *J. Symbolic Computation*, 3:69–116, 1987.
4. J. Giesl and T. Arts. Verification of Erlang processes by dependency pairs. *Applicable Algebra in Engineering, Communication and Computing*, 12(1,2):39–72, 2001.
5. J. Giesl, T. Arts, and E. Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *Journal of Symbolic Computation*, 34(1):21–58, 2002.
6. S. Kamin and J. J. Lévy. Two generalizations of the recursive path ordering. Unpublished Manuscript, University of Illinois, IL, USA, 1980.
7. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Proc. POPL '01*, pages 81–92, 2001.
8. E. Ohlebusch. Termination of logic programs: Transformational approaches revisited. *Appl. Algebra in Engineering, Communication and Computing*, 12(1,2):73–116, 2001.
9. R. Thiemann and J. Giesl. Size-change termination for term rewriting. In *Proc. 14th RTA*, LNCS, 2003. To appear. Extended version appeared as Technical Report AIB-2003-02[2], RWTH Aachen, Germany, 2003.

[2] Available from `http://aib.informatik.rwth-aachen.de`.

Proving Termination of Rewriting with C*i*ME

Evelyne Contejean, Claude Marché, Benjamin Monate and Xavier Urbain

Laboratoire de Recherche en Informatique (LRI) CNRS UMR 8623
Bât. 490, Université Paris-Sud, Centre d'Orsay
91405 Orsay Cedex, France
E-mail: {contejea,marche,monate,urbain}@lri.fr

1 Introduction

The various and successful applications of rewriting has brought the need for automated manipulation of TRSs, thus of rewriting tools. Amongst those is the C*i*ME system [2].

C*i*ME is a toolbox which may be used for dealing with matching and unification of strings or terms modulo equational theories (A, AC, ...), tree automata, parameterized string rewriting [4], etc. Its capabilities w.r.t. TRSs include confluence checking, completion; it is also able to find (AC-)termination proofs with full automation, thanks to its termination experts and its efficient constraint solver over finite domains.

2 Proving Termination

A termination proof search in C*i*ME begins with computation of termination constraints. These may be checked by ACRPO [5] or translated into Diophantine constraints in order to obtain polynomial interpretations. In that last case, the finite domain constraint solver of C*i*ME tries to find a solution. Our policy is to provide a tool for proving termination of the TRS one meets *in practice*. Hence, we focus on modular/incremental proofs, a case for which the search for termination preserved under non-deterministic collapse is a particularly important issue.

Basic Features. Several criteria may be used in C*i*ME: The standard one (all rules strictly decrease) but also *dependency pairs* criteria [1], with or without marking of symbols, and with or without dependency graphs. Our extensions of dependency pairs to the AC case are also implemented [3].

We use two kinds of cycle analysis for the dependency graph refinement: one which treats all strongly connected components (and which is, thus, very efficient) and one which treats all strongly connected parts of the graph; the latter being more powerful but of higher complexity than the former.

The search for polynomial interpretations may be parameterized by the kind of polynomial to restrict to (linear, simple ou simple-mixed) and by the bound of their coefficients. In any case, AC-compatible interpretations will be used for AC-symbols.

Restricting to linear polynomials leads to fewer and easier (smaller) constraints than restricting to more complex polynomials, but at the cost of some termination power. Similarly, restricting to very small coefficients leads to faster constraint solving but might be not enough to find a suitable ordering.

Thus, in order to deal with TRS with numerous rules that are common in practice, and so as to make constraints as weak as possible, C*i*ME makes use of powerful modular and incremental criteria.

Modularity The algorithms at work in C*i*ME involve powerful methods so as to discover proofs of $\mathcal{CE}$-termination in an incremental and modular fashion [6,7]. This *divide*

and conquer approach significantly weakens constraints over orderings and decreases their number.

TRSs may be defined in C*i*ME as hierarchies of rewriting modules, we denote those as Hierarchical TRS (HTRS for short), and termination proofs are performed incrementally/modularly on modules constituting those extensions. In particular $\mathcal{CE}$-termination of a module R_i is proven *only once*, and never again in a termination proof of any extension of R_i. The function for searching an incremental/modular termination proof of a HTRS is `h_termination`. It tries to find a termination proof of the rewrite system given as argument by providing, for each sub-hierarchy (w.r.t. topological sorting), a suitable ordering.

Optimisations such as symbol marking and dependency graphs may be used as they affect both C*i*ME incremental/modular and classical termination experts.

In order to boost the efficiency of the termination expert, it may prove useful to restrict to nonexpensive criteria on most of the incremental proof.

For instance, when searching for a proof using polynomial interpretations, such tuning may be done with help of function `h_termination_with` which takes as first argument a list of pairs *criterion, bound*. They denote the kind of polynomials and the bounds to be tried successively on a module when a search fails.

Finally, a HTRS may be considered as a hierarchy of minimal modules. HTRS are then split up in minimal modules for termination proofs, which is particularly useful when dealing with a huge bunch of rules in a single HTRS.

3 Example

The following TRS describes natural numbers in binary notation and multisets and computes the sum of all numbers in a multiset. The termination proof using C*i*ME is done as follows. Firstly we enter variables, the signature and the TRS into the C*i*ME system:

```
CiME> let X = vars "x y z l b";
CiME> let F = signature "
    #, empty   : constant ; 0,1 : postfix unary ;
    singl, sum : unary ;
    U : AC ;  + : infix binary ;";
CiME> let R = HTRS {} F X "
  (#)0 -> #;      # + x -> x;      x + # -> x;
  (x)0 + (y)0 -> (x+y)0;  (x)0 + (y)1 -> (x+y)1;
  (x)1 + (y)0 -> (x+y)1;  (x)1 + (y)1 -> (x+y+(#)1)0;
  (x + y) + z -> x + (y + z);
  empty U b -> b;         sum(empty) -> (#)0;
  sum(singl(x)) -> x;     sum(x U y) -> sum(x) + sum(y);";
```

(Note the AC operator `U`.) Then we may choose minimal decomposition of `R` and an incremental/modular proof using linear polynomials with coefficients in $[0, 1]$ or simple polynomials with coefficients in $[0, 2]$ if no suitable linear interpretation is found.

```
CiME> termcrit "minimal";
CiME> h_termination_with {("linear",1);("simple",2)} R;
```

In less than a second, the hierarchy is splitted up in 8 modules (6 of which yielding *no constraint*) and we obtain interpretations for the remaining 2, namely for module

$$\begin{cases} \#+x \to x \qquad x+\# \to x \qquad (x+y)+z \to x+(y+z) \\ (x)0+(y)0 \to (x+y)0 \quad (x)0+(y)1 \to (x+y)1 \\ (x)1+(y)0 \to (x+y)1 \quad (x)1+(y)1 \to ((x+y)+(\#)1)0 \end{cases}$$

the tool proposes:

```
 checking each of the 1 strongly connected components :
 checking component 1 (disjunction of 1 constraints)
[#] = 0;
[0](X0) = X0 + 1;
[1](X0) = 2*X0 + 2;
[+](X0,X1) = X1*X0 + X1 + 2*X0 + 1;
['+'](X0,X1) = X1*X0 + 2*X0;
```

where `'+'` is the marked copy of `+`, and for module

$$\begin{cases} sum(singl(x)) \to x \qquad sum(empty) \to (\#)0 \\ sum(x \cup y) \to sum(x)+sum(y) \end{cases}$$

```
checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[#] = 0;
[0](X0) = 0;
[1](X0) = 0;
[+](X0,X1) = X1 + X0;
[empty] = 0;
[singl](X0) = X0;
[U](X0,X1) = X1 + X0 + 1;
[sum](X0) = X0;
['sum'](X0) = X0;
Termination proof found.
```

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. Marché, B. Monate, and X. Urbain. CiME version 2, 2000. Available at `http://cime.lri.fr/`.
3. K. Kusakari, C. Marché, and X. Urbain. Termination of associative-commutative rewriting using dependency pairs criteria. Research Report 1304, LRI, 2002.
4. B. Monate. *Propriétés uniformes de familles de systèmes de réécriture de mots paramétrées par des entiers*. Thèse de doctorat, Université Paris-Sud, Orsay, France, Jan. 2002.
5. A. Rubio. A fully syntactic AC-RPO. In P. Narendran and M. Rusinowitch, editors, *10th International Conference on Rewriting Techniques and Applications*, volume 1631 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag.
6. X. Urbain. Automated Incremental Termination Proofs for Hierarchically defined Terms Rewriting Systems. In R. Goré, A. Leitsch, and T. Nipkow, editors, *International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 485–498, Siena, Italy, june 2001. Springer-Verlag.
7. X. Urbain. Modular and incremental automated termination proofs. To appear in the *Journal of Automated Reasoning*, 2003.

The TALP Tool for Termination Analysis of Logic Programs

Enno Ohlebusch[1], Claus Claves[2], and Claude Marché[3]

[1] Faculty of Technology, University of Bielefeld
P.O. Box 10 01 31, 33501 Bielefeld, Germany
Email: enno@techfak.uni-bielefeld.de
[2] Bremer Str. 39, 33613 Bielefeld, Germany
claus.claves@bertelsmann.de
[3] LRI (UMR 8623 du CNRS) & INRIA Futurs
Bât. 490, Université de Paris-Sud, Centre d'Orsay
91405 Orsay Cedex, France
Email: marche@lri.fr

1 Introduction

In the last decade, the automatic termination analysis of logic programs has been receiving increasing attention. Among other methods, techniques have been proposed that transform a well-moded logic program into a term rewriting system (TRS) so that termination of the TRS implies termination of the logic program under Prolog's selection rule. In [8] it has been shown that the two-stage transformation obtained by combining the transformations of [6] into deterministic conditional TRSs (CTRSs) with a further transformation into TRSs [3] yields the transformation proposed in [2], and that these three transformations are equally powerful. In most cases simplification orderings are not sufficient to prove termination of the TRSs obtained by the two-stage transformation. However, if one uses the dependency pair method [1] in combination with polynomial interpretations instead, then most of the examples described in the literature can automatically be proven terminating. Based on these observations, we have implemented a tool for proving termination of logic programs automatically. This tool consists of a front-end which implements the two-stage transformation and a back-end, the C*i*ME system [4], for proving termination of the generated TRS.

As in [5], we have tested the TALP system on well-known examples (the benchmarks are collected in [7]). A Web interface for TALP is available at http://bibiserv.techfak.uni-bielefeld.de/talp/. Overall, our results are comparable to those reported in [5] but there are examples for which TALP succeeds and other tools don't (the example in Sect. 3 for instance) and vice versa. During our experiments we made the following observations. For the class of TRSs generated from logic programs, a termination proof using the standard Manna-Ness criterion is usually not possible. On the other hand, the dependency pairs criterion is quite powerful, using moreover very simple polynomial interpretations: linear polynomial interpretation with coefficients in the interval $[0;2]$. The restriction to linear polynomial interpretations seems to be a very good heuristic because whenever searching for a linear interpretation fails, then searching for a more general one, like simple-mixed, does not succeed either.

Moreover, for programs of a larger size, with many predicates, being able to perform an *incremental* proof [9, 10] appears to be very important.

2 Transformation of Programs into TRSs

TALP takes a Prolog program and a query as input and proceeds in four steps:

1. The Prolog program is translated into a logic program P. In this process, clauses with `if-then`-structures, disjunctions, or negated atoms are translated into new clauses. For instance, the clause $A \leftarrow B, \texttt{not}\ C, D$ is replaced with the clauses $A \leftarrow B, C, \texttt{fail}$ and $A \leftarrow B, D$. Cuts are ignored.
2. The query determines which of the arguments in its predicates are used as input and output, respectively. According to this information, the tool tries to generate a *moding* for the logic program such that the program is *well-moded*. If this step is successfully completed, the logic program will be transformed into a TRS as follows.
3. Every atom $A = p(t_1, \ldots, t_n)$ with input positions $i_1, \ldots, i_k$ and output positions $i_{k+1}, \ldots, i_n$ associates with a rewrite rule

   $$\rho(A) = p_{in}(t_{i_1}, \ldots, t_{i_k}) \to p_{out}(t_{i_{k+1}}, \ldots, t_{i_n})$$

   and every program clause $C = A \leftarrow B_1, \ldots, B_m$ is transformed into a conditional rewrite rule $\rho(C) = \rho(A) \Leftarrow \rho(B_1), \ldots, \rho(B_m)$. The CTRS $R_P = \{\, \rho(C) \mid C \in P \,\}$ obtained in this way is deterministic because the logic program P is well-moded.
4. Every rule $l \to r \Leftarrow c \in R_P$ with n conditions in c is transformed into $n+1$ unconditional rewrite rules by operator U defined inductively by :

   $$\begin{aligned} U(l \to r) &= \{l \to r\} \\ U(l \to r \Leftarrow s \to t, c) &= \{l \to u(s, x)\} \cup U(u(t, x) \to r \Leftarrow c) \end{aligned}$$

   where u is a fresh symbol and $x = Var(l) \cap (Var(t) \cup Var(c) \cup Var(r))$

3 Example

If TALP gets the following Prolog program with query $\texttt{flat}(in, out)$ as input

```
flat(niltree, nil).
flat(tree(X, niltree, T), cons(X, L)) :- flat(T, L).
flat(tree(X, tree(Y, T1, T2), T3), L) :-
  flat(tree(Y, T1, tree(X, T2, T3)), L).
```

then the first transformation yields the CTRS

$$\begin{aligned} \texttt{flat}_{in}(\texttt{niltree}) &\to \texttt{flat}_{out}(\texttt{nil}) \\ \texttt{flat}_{in}(\texttt{tree}(x, \texttt{niltree}, t)) &\to \texttt{flat}_{out}(\texttt{cons}(x, l)) \Leftarrow \\ &\qquad \texttt{flat}_{in}(t) \to \texttt{flat}_{out}(l) \\ \texttt{flat}_{in}(\texttt{tree}(x, \texttt{tree}(y, t_1, t_2), t_3)) &\to \texttt{flat}_{out}(l) \Leftarrow \\ &\qquad \texttt{flat}_{in}(\texttt{tree}(y, t_1, \texttt{tree}(x, t_2, t_3))) \to \texttt{flat}_{out}(l) \end{aligned}$$

and the second transformation yields the unconditional TRS

$$\begin{aligned}
\mathtt{flat}_{in}(\mathtt{niltree}) &\to \mathtt{flat}_{out}(\mathtt{nil})\\
\mathtt{flat}_{in}(\mathtt{tree}(x,\mathtt{niltree},t)) &\to u_1(\mathtt{flat}_{in}(t),x)\\
u_1(\mathtt{flat}_{out}(l),x) &\to \mathtt{flat}_{out}(\mathtt{cons}(x,l))\\
\mathtt{flat}_{in}(\mathtt{tree}(x,\mathtt{tree}(y,t_1,t_2),t_3)) &\to u_2(\mathtt{flat}_{in}(\mathtt{tree}(y,t_1,\mathtt{tree}(x,t_2,t_3))))\\
u_2(\mathtt{flat}_{out}(l)) &\to \mathtt{flat}_{out}(l)
\end{aligned}$$

Subsequently C*i*ME is asked to find a linear polynomial interpretation with coefficients in the interval $[0;2]$. It generates the following interpretation

$$\begin{array}{lll}
[\![\mathtt{nil}]\!] = 0 & [\![\mathtt{flat}_{out}]\!](x_0) = 0 & [\![u_1]\!](x_0,x_1) = 0\\
[\![\mathtt{niltree}]\!] = 0 & [\![u_2]\!](x_0) = 0 & [\![\mathtt{tree}]\!](x_0,x_1,x_2) = x_2+2x_1+1\\
[\![\mathtt{flat}_{in}]\!](x_0) = 0 & [\![\mathtt{cons}]\!](x_0,x_1) = 0 & [\![\mathtt{FLAT}_{in}]\!](x_0) = x_0
\end{array}$$

and the induced polynomial ordering satisfies all constraints obtained from the cycles in the estimated dependency graph.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
2. T. Arts and H. Zantema. Termination of logic programs via labelled term rewrite systems. In *Proceedings of Computing Science in the Netherlands*, pages 22–34, 1995.
3. M. Chtourou and M. Rusinowitch. Méthode transformationnelle pour la preuve de terminaison des programmes logiques. Unpublished manuscript, 1993.
4. E. Contejean, C. Marché, B. Monate, and X. Urbain. CiME version 2, 2000. Available at `http://cime.lri.fr/`.
5. S. Decorte, D. D. Schreye, and H. Vandecasteele. Constraint-based termination analysis of logic programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(6):1137–1195, 1999.
6. H. Ganzinger and U. Waldmann. Termination proofs of well-moded logic programs via conditional rewrite systems. In *3rd International Workshop on Conditional Term Rewriting Systems*, volume 656 of *Lecture Notes in Computer Science*, pages 113–127, Berlin, 1993. Springer-Verlag.
7. N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs (with detailed experimental results). Technical report, Hebrew University, Jerusalem, 1997.
8. E. Ohlebusch. Transforming conditional rewrite systems with extra variables into unconditional systems. In *6th International Conference on Logic for Programming and Automated Reasoning*, volume 1705 of *Lecture Notes in Artificial Intelligence*, pages 111–130, Berlin, 1999.
9. X. Urbain. Automated incremental termination proofs for hierarchically defined term rewriting systems. In R. Goré, A. Leitsch, and T. Nipkow, editors, *International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 485–498, Siena, Italy, june 2001. Springer-Verlag.
10. X. Urbain. Modular and incremental automated termination proofs. *Journal of Automated Reasoning*, 2003. to appear.

CARIBOO : A Multi-Strategy Termination Proof Tool Based on Induction

Olivier Fissore, Isabelle Gnaedig, Hélène Kirchner
LORIA-INRIA & LORIA-CNRS
B.P. 239 F-54506 Vandoeuvre-lès-Nancy Cedex
e-mail : fissore@loria.fr, gnaedig@loria.fr, Helene.Kirchner@loria.fr

1 A termination proof tool for rule-based programs

CARIBOO is a termination proof tool for rule-based programming languages, where a program is a rewrite system and query evaluation consists in rewriting a ground expression [3]. It applies to languages such as ASF+SDF, Maude, Cafe-OBJ, or ELAN.

By contrast with most of the existing tools, which prove in general termination of standard rewriting (rewriting without strategy) on the free term algebra, our proof tool, named CARIBOO (for **C**omputing **A**bst**R**action for **I**nduction **B**ased termination pr**OO**fs), allows proving termination under specific reduction strategies, which becomes of special interest when the computations diverge for standard rewriting. It deals in particular with :

- the innermost strategy, specially useful when the rule-based formalism expresses functional programs, and central in the evaluation process of ELAN,
- local strategies on operators, provided in OBJ-like languages, and allowing to control evaluation strategies in a very fine local way,
- the outermost strategy, useful to avoid evaluations known to be non terminating for the standard strategy, to make strategy computations shorter, and used for interpreters and compilers using call by name.

2 Proving termination by explicit induction

The proof technique of CARIBOO is backed by a generic inductive process, declinable for the different considered strategies [4, 2]. For proving that a term t terminates, we proceed by explicit induction on the termination property, on ground terms with a noetherian ordering $\succ$, assuming that for any t' such that $t \succ t'$, t' terminates (according to the given strategy). We first prove that a basic set of minimal elements for $\succ$ terminates. We require the subterm property for $\succ$, so the set of minimal elements is a subset of the set of constants of the signature. We then simulate the rewriting (according to the given strategy) derivation tree starting from a ground term t which is an instance of a term $g(x_1,\ldots,x_m)$, for all defined symbols g, and variables $x_1,\ldots,x_m$. Proving termination on ground terms amounts to prove that these rewriting (according to the given strategy) derivation trees have only finite branches.

Each derivation tree is simulated by a proof tree starting from $g(x_1,\ldots,x_m)$, and developed by alternatively using two main concepts, namely narrowing and abstraction, whose definition depends on the considered rewriting strategy. More precisely, narrowing schematizes all rewriting possibilities of the terms in the derivations. The abstraction process simulates the normalization of subterms in the derivations, according to the strategy. It consists in replacing these subterms by special variables, denoting any of their normal forms. This abstraction step is performed on subterms that can be assumed terminating by induction hypothesis. Our procedure terminates whith success when termination has been inductively established for each proof tree.

3 A constraint-based proof process

The induction ordering is not given a priori but is determined by ordering constraints set along the proof. Thanks to the power of induction, ordering constraints are often simpler than for other constraint-based termination proof rewrite methods, like the rule-orientation based method using simplification orderings: they are often satisfied by the subterm ordering or the embedding ordering, integrated in CARIBOO. Otherwise, they can be delegated to automatic external solvers like C*i*ME2 [1].

The narrowing process, well-known to easily diverge, is restricted for the innermost and local strategies, by abstraction constraints, expressing which terms abstraction variables represent the normal form of. Abstraction constraints are used only for proofs whose narrowing process diverges, and are also automatically solved in most cases. For automatically solved constraints, our proof process is completely automatic.

4 A flexible technique allowing integrating other approaches

The proof method of CARIBOO allows the combination with auxiliary termination proofs with a different technique: when the induction hypothesis cannot be applied on a term u during the proof, it is often possible to prove termination (according to the given strategy) of any ground instance of u by any free other way, like testing its ground reducibility or studying rules involved in its reduction.

This flexibility of our method, allowing to integrate other termination proof approaches in the main inductive proof process in a natural way, gives us a formal framework for different termination tools to efficiently cooperate, unlike other methods that, when failing, require the user to find another proof technique by himself.

5 A user-friendly ELAN reflexive tool

The proof tool consists of two main parts :

- the proof procedure, written in ELAN, generating the proof trees and writing information (constraints and proof trees) both on the standard output and input files : our abstraction and narrowing-based proof mechanism is formalized with inference rules, backed by the same principle for the different studied rewriting strategies, and a strategy to apply them, dependent on the rewriting strategy. As ELAN was precisely designed for prototyping rule and strategy-based algorithms, the proof procedure is directly encoded through ELAN rules and an ELAN strategy to apply them.

- a graphical user interface (GUI), written in Java, making it possible for the user to follow each step of the proof process : which defined symbols have already been handled and, for each of them, the proof tree together with the detail of each state.

CARIBOO is reflexive in the sense that its rule-based proof process can be applied to prove termination of ELAN programs.

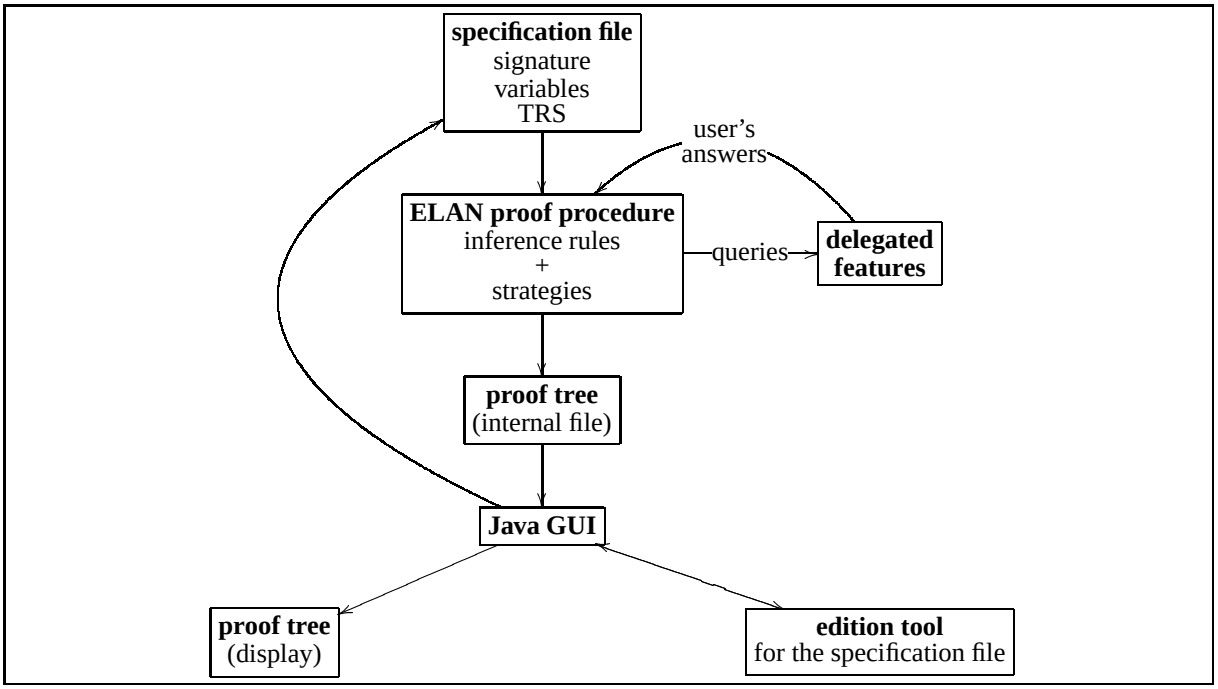
6 How CARIBOO works

The ELAN proof procedure expects the user to give the TRS which he wants to prove termination of, i.e. the rewrite rules of the system along with the variables and the signature of the algebra.
An edition tool has been written in Java to help the user enter these data, which are then transformed into the ELAN specification used by the main program.

During the process, the ELAN proof procedure builds the proof tree, and writes the detail of each state into a file. This information is then picked up by the Java GUI that displays in a graphical way the evolution of the proof process.

To handle the delegated features, i.e. to prove termination of a term by an external criterion, or for abstraction constraint solving, the proof procedure directly communicates with the user through the window in which it has been launched.

Then, once the proof process is achieved, the proof tree has been generated by the ELAN proof procedure for each defined symbol and displayed by the Java GUI. We can get the detail of each state of these trees. If none of the proof trees contains a failure state, then the TRS given as input is terminating w.r.t. the given strategy. The trace can be saved in text or LaTeX format.



CARIBOO will also soon offer a termination proof procedure for full ELAN strategies, transforming ELAN strategies into simpler ones, and applying sufficient conditions for termination on the simplified forms.

References

[1] Benjamin Monate Evelyne Contejean, Claude Marché and Xavier Urbain. C*i*ME version 2, 2000. Preliminary version available at http://cime.lri.fr/.

[2] O. Fissore, I. Gnaedig, and H. Kirchner. Termination of rewriting with local strategies. In M. P. Bonacina and B. Gramlich, editors, *Selected papers of the 4th International Workshop on Strategies in Automated Deduction*, volume 58 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[3] O. Fissore, I. Gnaedig, and H. Kirchner. CARIBOO : An induction based proof tool for termination with strategies. In *Proceedings of the Fourth International Conference on Principles and Practice of Declarative Programming (PPDP)*, Pittsburgh, USA, October 2002. ACM Press.

[4] O. Fissore, I. Gnaedig, and H. Kirchner. Outermost ground termination. In *Proceedings of the Fourth International Workshop on Rewriting Logic and Its Applications*, Pisa, Italy, September 2002. Electronic Notes in Computer Science. To appear.