The 2007 Federated Conference on Rewriting, Deduction and Programming

Paris, France

June 25 – 29, 2007

WST'07

The 9th International Workshop on Termination

June 29th, 2007

Proceedings

Editors:

Dieter Hofbauer and Alexander Serebrenik

Preface

This volume contains the papers presented at the *9th International Workshop on Termination (WST'07)*, held on June 29, 2007 in Paris, France, as part of the *Federated Conference on Rewriting, Deduction, and Programming (RDP'07).*

The workshop traditionally brings together, in an informal setting, researchers interested in all aspects of termination, whether this interest be practical or theoretical, primary or derived. The workshop also provides a ground for cross-fertilisation of ideas from term rewriting and from the different programming language communities. WST 2007 continues the sequence of successful workshops held in St. Andrews (1993), La Bresse (1995), Ede (1997), Dagstuhl (1999), Utrecht (2001), Valencia (2003), Aachen (2004), and Seattle (2006).

There were 21 submissions by researchers from twelve countries, including six submissions by authors from different countries. Each submission was reviewed by at least 3 programme committee members. The committee decided to accept all the submissions for presentation at the workshop.

In addition to these submissions the current volume includes a report on this year's competition of termination provers. For detailed information on the competition results the interested reader is referred to

`http://www.lri.fr/~marche/termination-competition/2007/`

The workshop organisers would like to thank, first of all, the authors of submitted papers for their interest in WST. We also thank the programme committee members and the external reviewers for their outstanding work during the reviewing process.

We are grateful to Claude Marché, Johannes Waldmann and Hans Zantema for their work in the termination competition committee, and in particular to Claude Marché for running the competition.

Organising the workshop would have been impossible without the support of RDP 2007 organisers. Our special thanks go to Ralf Treinen for taking care of numerous issues, including publishing this volume, updating the web-site and making the local arrangements. We would also like to thank the institutional sponsors of RDP'07 without whom it would not have been possible to organise that event: the Conservatoire des Arts et Métiers (CNAM), the Centre National de la Recherche Scientifique (CNRS), the École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise (ENSIEE), the GDR Informatique Mathématique, the Institut National de Recherche en Informatique et Automatique (INRIA) unit Futurs, and the Région Île de France.

Last but not least, we thank the authors of the EasyChair conference management system which made the practical organisation of the reviewing process considerably easier.

May 2007

Dieter Hofbauer
Alexander Serebrenik

Workshop Organization

Programme Chairs

Dieter Hofbauer
Alexander Serebrenik

Programme Committee

Daniel De Schreye
Nachum Dershowitz
Samir Genaim
Jürgen Giesl
Isabelle Gnaedig
Aart Middeldorp
Étienne Payet

External Reviewers

Michael Codish
Mircea-Dan Hernest
Adam Koprowski
Fred Mesnard
Georg Moser
Manh Thang Nguyen
Paolo Pilozzi
Fausto Spoto
Christian Sternagel
Harald Zankl

Table of Contents

Arctic Termination

Johannes Waldmann[1]

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

Abstract. We extend the matrix method for proving termination of string rewriting: we use matrices of the arctic semi-ring. (Its domain are the natural numbers, extended by minus infinity, arctic multiplication is standard addition, and arctic addition is the standard maximum-operation.) This method will be used by Matchbox in the 2007 Termination Competition. It produces some short termination proofs for hard or previously unsolved problems. We prove that the arctic termination method contains the quasi-periodic method as a special case.

1 Arctic Matrix Interpretations

We adapt the matrix method for proving termination of string rewriting [7] to matrices over a different semi-ring.

(This present section is extracted from a paper on the general theory of matrix interpretations over ordered semi-rings that has been submitted elsewhere [8]. Section 2 is new.)

The arctic semi-ring $\mathbb{A}$, also known as the (max,plus) algebra [5], has as carrier the set $\mathbb{N}$ of naturals, extended with $-\infty$. Arctic addition is the maximum operation, with neutral element $-\infty$, and arctic multiplication is standard addition, with neutral element 0.

$\mathbb{A}$ is ordered by the standard relation $<$ on $\mathbb{N}$, extended by making $-\infty$ smallest. This ordering satisfies weak monotonicity for max: if $x \leq y$ then $\max(x, z) \leq \max(y, z)$, and strict monotonicity for plus except at $-\infty$: if $-\infty < z$, then $x < y \Rightarrow x + z < y + z$. The max operation is not strict in this sense, but we call it *half strict* because of $x_1 < x_2 \wedge y_1 < y_2$ implies $\max(x_1, y_1) < \max(x_2, y_2)$.

Arctic polynomial interpretations have been used by Amadio [1], while arctic matrices for termination seem to be new.

A Σ-interpretation of dimension d is a mapping $[\cdot] : \Sigma \to \mathbb{A}^{d\times d}$ (from letters to matrices with entries in the semi-ring). We call it *admissible* if each matrix $[c]$ has upper left entry $[c]_{1,1} > -\infty$.

An interpretation can be extended from letters to words by multiplication, where arctic matrix multiplication is defined as usual: $(A \cdot B)_{i,k} = \max\{A_{i,j} + B_{j,k} \mid 1 \leq j \leq d\}$. An interpretation is *weakly compatible* with a rewriting rule $l \to r$ if $[l] \geq [r]$ where $\geq$ means point-wise $\geq$ for matrix entries at corresponding positions. An interpretation is *strictly compatible* with a rewriting rule $l \to r$ if $[l] > [r]$ where we say $A > B$ for matrices A, B if for each position (i, j) we have $A_{i,j} > B_{i,j} \vee B_{i,j} = -\infty$. That is, strict inequality except possibly at $-\infty$.

Theorem 1. *If there is an admissible interpretation that is strictly compatible with a rewriting system R and weakly compatible with S then R is terminating relative to S.*

Example 1. For the rewriting system $R = \{a^2 \to aba\}$, we take the interpretation on the left, and obtain the inequality on the right, proving termination:

$$[a] = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}, \quad [b] = \begin{pmatrix} 0 & -\infty \\ -\infty & -\infty \end{pmatrix}; \quad [a^2] = \begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix} > \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} = [aba].$$

Since for $k \geq 1$ we have $[b^k] = [b]$, this proves $\mathrm{SN}(R/\{b \to b^2\})$ as well. □

As described in [4], matrix implementations can be combined with the Dependency Pairs method [2]. This also works for arctic matrices. For interpretations of the DP symbols we take matrices of shape $\mathbb{A}^{1\times d}$ (row vectors), where each such vector is called admissable. Then the analogue of Theorem 1 holds, and this gives a powerful method that solves some hard problems.

Example 2. For $R = \{a^2 \to bc, b^2 \to ac, c^2 \to ab\}$ ("Zantema's Other Problem") the nontrivial part of $\mathrm{DP}(R)$ is $\{Aa \to Bc, Bb \to Ac\}$. Take the interpretation

$$[a] = \begin{pmatrix} 0 & 3 \\ 2 & 1 \end{pmatrix}, [b] = \begin{pmatrix} 3 & 2 \\ 1 & -\infty \end{pmatrix}, [c] = \begin{pmatrix} 0 & 1 \\ 3 & 2 \end{pmatrix}, [A] = [B] = (0 \ \ -\infty).$$

Then we have these weak compatibilities

$$[a^2] = \begin{pmatrix} 5 & 4 \\ 3 & 5 \end{pmatrix} \geq \begin{pmatrix} 5 & 4 \\ 1 & 2 \end{pmatrix} = [bc], \qquad [b^2] = \begin{pmatrix} 6 & 5 \\ 4 & 3 \end{pmatrix} = [ac]$$

$$[c^2] = \begin{pmatrix} 4 & 3 \\ 5 & 4 \end{pmatrix} \geq \begin{pmatrix} 4 & 2 \\ 5 & 4 \end{pmatrix} = [ab], \qquad [Aa] = (0 \ 3) \geq (0 \ 1) = [Bc]$$

and the strict compatibility $[Bb] = (3 \ 2) > (0 \ 1) = [Ac]$. This allows to remove one rule and the rest is trivial (counting symbols). This looks like currently the shortest termination proof for this problem. Previous "automatic" proofs [6] used an integer matrix interpretation of size 5, resp. 4 (after DP transformation). □

2 Relation to Quasi-Periodic Interpretations

A quasi-period function $f : \mathbb{N} \to \mathbb{N}$ of period p (and slope 1) satisfies $\forall x : f(x+p) = f(x)+p$. See [9], from where we cite the following proof of termination of $S = \{0000 \to 1011, 1001 \to 0000\}$ (SRS/Gebhardt/18):

After removing the length-decreasing dependency pairs it remains to prove $\mathrm{SN}(R_{\mathrm{top}}/S)$ for $R = \{0_\#000 \to 1_\#011,\ 1_\#001 \to 0_\#000\}$. We use the following quasi-periodic interpretation with period 5:

$$\begin{array}{lllll}
[0](0) = 1, & [0](1) = 2, & [0](2) = 3, & [0](3) = 4, & [0](4) = 5, \\
[1](0) = 1, & [1](1) = 1, & [1](2) = 1, & [1](3) = 6, & [1](4) = 6, \\
[0_\#](0) = 2, & [0_\#](1) = 7, & [0_\#](2) = 7, & [0_\#](3) = 7, & [0_\#](4) = 7, \\
[1_\#](0) = 4, & [1_\#](1) = 4, & [1_\#](2) = 4, & [1_\#](3) = 8, & [1_\#](4) = 8.
\end{array}$$

We create an equivalent arctic matrix interpretation. The idea is to use integer division (by the period) where the remainder determines the position of the quotient in the matrix. The construction is presented in two steps.

In the first step, from the given quasi-periodic interpretation $[\cdot]$ with period p, we construct an arctic matrix interpretation $[\cdot]'$ of dimension $p \times p$ with

$$0 \leq x, y < p : [c](x) = p \cdot k + y \;\Rightarrow\; [c]'_{y,x} = k.$$

In our example we get (writing "$-$" for "$-\infty$", and starting the index ranges by 0 instead of 1 because it is more convenient)

$$[0]' = \begin{pmatrix} - & - & - & - & 1 \\ 0 & - & - & - & - \\ - & 0 & - & - & - \\ - & - & \boxed{0} & - & - \\ - & - & - & 0 & - \end{pmatrix}, \quad [1]' = \begin{pmatrix} - & - & - & - & - \\ 0 & 0 & 0 & 1 & 1 \\ - & - & - & - & - \\ - & - & - & - & - \\ - & - & - & - & - \end{pmatrix}, \quad [0_\#]' = \begin{pmatrix} - & - & - & - & - \\ - & - & - & - & - \\ 0 & 1 & 1 & 1 & 1 \\ - & - & - & - & - \\ - & - & - & - & - \end{pmatrix}, \quad [1_\#]' = \begin{pmatrix} - & - & - & - & - \\ - & - & - & - & - \\ - & - & - & - & - \\ - & - & - & 1 & 1 \\ 0 & 0 & 0 & - & - \end{pmatrix}$$

E.g. the (boxed) entry $[0]'_{3,2} = 0$ is because $[0](2) = 5 \cdot 0 + 3$.

Then we have $\forall w \in \Sigma^* : [w](x) = p \cdot k + y \iff [w]'_{y,x} = k$ where on the right hand side we have a matrix product in the arctic semi-ring.

In the second step we fill the "minus infinity" entries of those matrices. Each column of $[c]'$ has shape

$$(-, \ldots, -, \boxed{e}, -, \ldots, -)^T,$$

and we produce from that in interpretation $[c]''$ with the column

$$(e, \ldots, e, \boxed{e}, e-1, \ldots, e-1)^T.$$

The places above entry e are filled with e, and the places below entry e are filled with $e-1$ (if $e = 0$, then take $-\infty$). In the example, we get

$$[0]'' = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ - & 0 & 0 & 0 & 0 \\ - & - & 0 & 0 & 0 \\ - & - & - & 0 & 0 \end{pmatrix}, \quad [1]'' = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ - & - & - & 0 & 0 \\ - & - & - & 0 & 0 \\ - & - & - & 0 & 0 \end{pmatrix}, \quad [0_\#]'' = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ - & 0 & 0 & 0 & 0 \\ - & 0 & 0 & 0 & 0 \end{pmatrix}, \quad [1_\#]'' = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Note that each row is monotonically increasing, and each column is monotonically decreasing with at most one step. This property is invariant under multiplication. For each w we have that if $[w]'_{y,x} = k$, then $z < y \Rightarrow [w]''_{z,x} = k$, and $z > y \Rightarrow [w]''_{z,x} = k - 1$.

It follows that if the quasi-periodic interpretation $[\cdot]$ is weakly compatible with rule $l \to r$, then for the arctic matrices we have $[l]'' \geq [r]''$, and for a strictly comptible interpretation, we get a strict inequality.

3 Discussion

We have implemented the arctic matrix termination method as part of the 2007 version of Matchbox. The matrices are found as solutions of an integer constraint system that is translated to a SAT problem, which is solved by minisat [3].

Here we report only on results for the string termination problems submitted in 2006 by Endrullis and Gebhardt because they appeared to be very hard. With the DP transformation and arctic matrix interpretations (without any further additions or transformations) we can solve

– the problems Endrullis/$\{1 \dots 7, 9\}$. Note: only problem 9 could be solved in the 2006 competition (by Jambox and MultumNonMulta), meanwhile 1, 2, 5, 6 can be solved by quasi-periodic interpretations (QPI, Torpa). Problem 8 remains open.
– the problems Gebhardt/$\{4, 7, 9, 11, 14, 16, 17, 18\}$. Of these, 2 and 16 were solved in 2006, and all except 14 can now be done by QPI. The remaining open problems are 10, 13, 19, 20 (the others are looping).

While the arctic matrix method subsumes QPI, we do not imply that an arctic implementation automatically outperformes a QPI implementation. Rather, quasi-periodic functions give a more efficient representation for arctic matrices of a certain shape, but they may miss arctic proofs of different shapes.

References

1. Roberto M. Amadio. Max-plus quasi-interpretations. In Martin Hofmann, editor, *TLCA*, volume 2701 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2003.
2. Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000.
3. Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.
4. Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of term rewriting. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 574–588. Springer, 2006.
5. Stephane Gaubert and Max Plus. Methods and applications of (max, +) linear algebra. In Rüdiger Reischuk and Michel Morvan, editors, *STACS*, volume 1200 of *Lecture Notes in Computer Science*, pages 261–282. Springer, 1997.
6. Dieter Hofbauer and Johannes Waldmann. Termination of $\{aa \to bc, bb \to ac, cc \to ab\}$. *Inf. Process. Lett.*, 98(4):156–158, 2006.
7. Dieter Hofbauer and Johannes Waldmann. Termination of string rewriting with matrix interpretations. In Frank Pfenning, editor, *RTA*, volume 4098 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2006.
8. Johannes Waldmann. Weighted automata for proving termination of string rewriting. submitted, 2006.
9. Johannes Waldmann and Hans Zantema. Termination by quasi-periodic interpretations. accepted for RTA, 2007.

Matrix Evolutions

Andreas Gebhardt[1], Dieter Hofbauer[2], and Johannes Waldmann[1]

[1] Hochschule für Technik, Wirtschaft und Kultur (FH) Leipzig
Fb IMN, PF 30 11 66, D-04251 Leipzig, Germany.
[2] Berufsakademie Nordhessen, Eichlerstr. 25, D-34537 Bad Wildungen, Germany.

Abstract. We introduce matrix interpretations with non-integer entries for termination proofs of string rewriting, and we propose to use evolutionary algorithms to find such interpretations.

1 Introduction

The matrix method [2] proves termination of string rewriting via compatible monotone interpretations into matrices with nonnegative integer entries. The compatibility conditions amount to a system of inequalities between polynomials in the matrix entries of the interpretation. It is a difficult problem to solve such a constraint system. Current implementations Matchbox and Jambox transform the integer constraint problem to a boolean constraint problem and use an external SAT solver.

In this note, we report on experiments with another method, evolutionary algorithms [4]. A set of interpretations is maintained (the *population*) and in each step, some new candidates are generated. The "fittest" among them are added to the population, and others are removed. This approach has been used by Hofbauer in the program Multum-Non-Multa and by Zantema (private communication). (Multum-Non-Multa uses additional ideas not explained here.)

For integer matrix interpretations, even a single "local" modification (decrease or increase one matrix entry by one) can have drastic effects, resembling a mutation. Therefore we suggest to use *non-integer* matrix entries for interpretations. This should allow for a better handling of localized variations. We have to ensure that the domain of the matrix interpretations is still well-founded (w.r.t. the component-wise ordering), so we cannot allow arbitrary non-integer valued matrices.

2 Integer Matrix Interpretations

The matrix method [2] interpretes letters by matrices from the set

$$E = \{A \in \mathbb{N}^{n\times n} \mid A_{1,1} \geq 1, A_{n,n} \geq 1\}.$$

(All matrices have square shape $n \times n$. We often leave n implicit. The choice of E corresponds to one particular version of the general method.) An interpretation

$[\cdot] : \Sigma \to E$ is extended to $[\cdot] : \Sigma^* \to E$ multiplicatively by $[a_1 \dots a_k] = [a_1] \cdot \dots \cdot [a_k]$. Note that E is closed w.r.t. multiplication: $E^* \subseteq E$. An interpretation is called *(weakly) compatible* with a rule $l \to r$ if $[l] \geq [r]$ where $\geq$ denotes the pointwise ordering. Equivalently, $[l] - [r] \in N$ where N (read: non-negative) denotes $\mathbb{N}^{n \times n}$. The interpretation is called *strictly compatible* with $l \to r$ if $[l] > [r]$ where we define $A > B$ as $A \geq B \wedge A_{1,n} > B_{1,n}$. Equivalently, $[l] - [r] \in P$ (read: positive) where

$$P = \{A \in \mathbb{N}^{n \times n} \mid A_{1,n} > 0\}.$$

Then $>$ is well-founded on $\mathbb{N}^{n \times n}$ and the set of admissible differences P is stable under multiplication: $E \cdot P \cdot E \subseteq P$. This implies the theorem: if an E-interpretation is compatible with a set of rules S and strictly compatible with a set of rules R, then R is terminating relative to S.

3 Non-Integer Matrix Interpretations

We propose to keep the idea of E, N, P but now entries can be real numbers. We define sets

$$\begin{aligned} N' &= \{A \in \mathbb{R}^{n \times n} \mid \forall i, j : A_{i,j} \geq 0\}, \\ E' &= \{A \in N' \mid A_{1,1} \geq 1 \wedge A_{n,n} \geq 1\}, \\ P' &= \{A \in N' \mid A_{1,n} > 0\}. \end{aligned}$$

Note that their restriction to $\mathbb{N}^{n \times n}$ corresponds to their respective original definition. Note also that over $\mathbb{N}$, the conditions $x \geq 1$ and $x > 0$ are equivalent, while over $\mathbb{R}$, they are not. We still say that an interpretation $[\cdot] : \Sigma \to N'$ is strictly (weakly, resp.) compatible with a rule $l \to r$ if $[l] - [r] \in P'$ (N', resp.).

Theorem 1. *For any finite string rewriting system R, and any string rewriting system S: if there is an E'-interpretation that is strictly compatible with R and weakly compatible with S, then R is terminating relative to S.*

Again, E' is closed, and P' is stable w.r.t. multiplication: $E'^* \subseteq E'$ and $E' \cdot P' \cdot E' \subseteq P'$.

But there seems to be a problem: the ordering $>$, defined as before, is not well-founded on N', since $>$ is dense on $\mathbb{R}$. The solution is to use a modified ordering $>_d$, depending on a parameter $d > 0$, as follows:

$$A >_d B \text{ iff } A \geq B \wedge A_{1,n} - B_{1,n} \geq d$$

This is well-founded on N'. An equivalent characterization is given by

$$P'_d = \{A \in N' \mid \wedge A_{1,n} \geq d\}$$

and $A >_d B$ iff $A - B \in P'_d$. Then for each $d > 0$, the set P'_d is stable under multiplication, $E' \cdot P'_d \cdot E' \subseteq P_d$, since the top left and bottom right entries in E' are ≥ 1, while > 0 would not be enough.

From the given rewriting system R and a strictly compatible interpretation $[\cdot]$, compute the minimum over the top right entries of the difference matrices

$$d = \min\{([l] - [r])_{1,n} \mid (l \to r) \in R\}.$$

(Note that indeed $d > 0$ since we required R to be finite.) Then $u \to_R v$ implies $[u] - [v] \in P'_d$, and this proves Theorem 1.

Example 1. Our implementation of the evolutionary approach described below found the following non-integer matrix interpretation that proves termination of the rewriting system $R = \{l \to r\} = \{babbab \to ababbba\}$ (SRS/Zantema/z057).

$$[a] = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad [b] = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & \frac{2}{3} & 0 & 0 \\ 0 & \frac{1}{3} & \frac{4}{3} & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad [l] - [r] = \begin{pmatrix} 0 & \frac{98}{3^4} & \frac{5}{3^5} & \frac{1}{3^3} \\ 0 & \frac{4}{3^5} & \frac{116}{3^7} & \frac{40}{3^5} \\ 0 & \frac{2}{3^5} & \frac{8}{3^5} & \frac{208}{3^5} \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

It is interesting to note that there also is a strictly compatible integer matrix interpretation of dimension 5, found in a few seconds by current solvers. After several CPU hours we still could not find a compatible integer matrix proof of dimension 4. (The system R has an easy alternative termination proof because it is RFC-matchbounded by 1, but this is not the point here.)

We note that exact numerical computation (e.g. using exact representations of rational or algebraic numbers) seems to be crucial when computing non-integer matrices for termination. The reasoning is as follows.

Assume we use approximate rationals (e.g. by using the `double` type or similar). To be certain of the inequalities, we would have to add some margin before making comparisons. So we always over-estimate some values. If the over-estimated matrices still fulfil all inequalities, then this implies that the underlying (but hidden) exact values fulfil the inequalities strictly. This means that there is some interpretation that allows to remove all rules at once. This is contrary to what one would expect: for a rewriting system with several rules, the usual termination proof by matrix interpretations (cf. recent Termination Competition) proceeds in several stages, removing only a few rules each time, using the concept of relative termination.

4 Implemention

The above example was found by an evolutionary algorithm [4] that maintains a population, which is a set of individuals, ranked by their fitness.

In our implementation, the (un-)fitness of an individual (i.e. a matrix interpretation $[\cdot]$) is computed by adding penalties for unwanted properties. For each rule $l \to r$ and each negative entry e in the $[l]-[r]$ matrix, the penalty is e^2. If the sum of these penalties is zero, then the interpretation is weakly compatible with all rules. If a weakly compatible interpretation has a positive entry at position $(1, n)$ in some matrix $[l] - [r]$, then it is strictly compatible with that rule, and the algorithm stops successfully, because that rule can be removed. So, as long as there is no positive entry in these positions, there is a fixed penalty $c_1 > 0$. This way, absence of penalties fully characterizes a successful individual.

In each step of the algorithm, a new candidate individual is computed from one (mutation) or two parents (crossover), and possibly enters the population. We use mutations (changing matrix entries) of two kinds: "drastic" and "local" changes. A drastic change resets an entry to 0, or to some random value. A local change modifies an entry by one unit (e.g. 1/3 in Example 1). In fact we execute a sequence of random local changes as long as the fitness increases, and sometimes even allow a decrease. This corresponds to "simulated annealing". (We also tried several crossover methods. None is a clear winner.)

5 Discussion

With these operations and some minor adjustments our implementation is able to prove termination for 39 of 127 of Zantema's examples from the Termination Problem Database. This is merely a proof of concept.

Besides questions of efficient implementation, it is interesting to know what is the relative "power" of matrix interpretations over the naturals, the rationals, and the reals. Obviously, each matrix proof over $\mathbb{N}$ can be seen as a proof over $\mathbb{Q}$ and $\mathbb{R}$ as well. What about the other direction: is there a rewriting system with a strictly compatible $\mathbb{R}$ matrix interpretation, but which has no compatible interpretation over $\mathbb{Q}$ or $\mathbb{N}$? We can also refine the question by considering dimensions: is it possible that there is a $\mathbb{Q}$ matrix interpretation smaller than the smallest $\mathbb{N}$ matrix interpretation, cf. Example 1.

For polynomial interpretations for term rewriting, Lucas [3] recently proved some "gap" theorems. It is hoped that these methods carry over.

In [1] it was shown how to use integer matrix interpretations for proving (relative) (top) termination of term rewriting. It is obvious that non-integer matrix interpretations carry over to this approach.

References

1. Jörg Endrullis, Johannes Waldmann, and Hans Zantema. *Matrix interpretations for proving termination of term rewriting.* In Ulrich Furbach and Natarajan Shankar (Eds.), IJCAR 2006, LNCS 4130, pp. 574–588. Springer-Verlag, 2006.
2. Dieter Hofbauer and Johannes Waldmann. *Termination of String Rewriting Systems with Matrix Interpretations.* In Frank Pfenning (Ed.): RTA 2006, LNCS 4098, pp. 328–342, 2006. Springer-Verlag, 2006
3. Salvador Lucas. *On the relative power of polynomials with real, rational, and integer coefficients in proofs of termination of rewriting.* Appl. Algebra Eng. Commun. Comput. 17(1): 49-73 (2006)
4. Karsten Weicker. *Evolutionäre Algorithmen.* B. G. Teubner, 2002

Certification of Matrix Interpretations in Coq

Adam Koprowski and Hans Zantema

Eindhoven University of Technology
Department of Computer Science
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
{A.Koprowski,H.Zantema}@tue.nl

Abstract. We describe how to certify the matrix interpretation method for proving termination of term rewriting in the theorem prover Coq. Certification requires both the formalization of the underlying theory of monotone algebras and their instantiation to vectors and matrices as they are used in the matrix method; as well as a mechanism to deal with concrete examples, based on those theoretical results.

1 Introduction

The *matrix interpretation* method [2] is an instantiation of the well-known approach for proving termination by well-founded orderings, which turned out to be powerful in practice. *CoLoR* [1] is a Coq library on rewriting and termination made with the goal of certification of termination proofs in mind. This goal is accomplished by *Rainbow* which consists of the description of termination proofs in XML format and a mechanism to transform proofs in this format into Coq scripts that certify their correctness (using theoretical results from CoLoR).

We succeeded in certifying the matrix interpretation method in Coq. We did that by formalizing the theory of monotone algebras and then instantiating it to the algebra of vectors used in the matrix interpretation method. Then we constructed a framework within Coq to check that all the conditions of the respective theorems are fulfilled in the instantiation to concrete examples. Finally we extended Rainbow with a proof format for the matrix interpretation method and with the ability to translate proofs in this format to certified Coq proofs.

2 Monotone algebras

Here we summarize the monotone algebra theory as presented in [2]. There is one difference: in contrast to [2] we do not consider many-sortedness. It is not essential for certification as every proof in the many-sorted setting can be trivially translated to one-sorted setting. The reason for this more complex setup in [2] is that it allows for an optimization in the search for termination proofs using matrix interpretations.

The monotone algebra approach works for all non-empty sets A; when using the matrix method A always consists of the set of vectors over $\mathbf{N}$.

Definition 1. *An operation $[f] : A \times \cdots \times A \to A$ is* monotone *with respect to a binary relation $\to$ on A if for all $a_i, b_i \in A$ for $i = 1, \dots, n$ with $a_i \to b_i$ for some i and $a_j = b_j$ for all $j \neq i$ we have $[f](a_1, \dots, a_n) \to [f](b_1, \dots, b_n)$.*

A weakly monotone Σ-algebra $(A, [\cdot], >, \gtrsim)$ *is a Σ-algebra $(A, [\cdot])$ equipped with two binary relations $>$, $\gtrsim$ on A such that*

- *$>$ is well-founded;*
- *$> \cdot \gtrsim \subseteq >$;*
- *for every $f \in \Sigma$ the operation $[f]$ is monotone with respect to $\gtrsim$.*

An extended monotone Σ-algebra $(A, [\cdot], >, \gtrsim)$ *is a weakly monotone Σ-algebra $(A, [\cdot], >, \gtrsim)$ in which moreover for every $f \in \Sigma$ the operation $[f]$ is monotone with respect to $>$.*

We write $\mathsf{SN}(\to_{\mathcal{R}}/\to_{\mathcal{S}})$ for termination of $\mathcal{R}$ relative to $\mathcal{S}$, meaning that every infinite $\mathcal{R} \cup \mathcal{S}$ reduction consists of only finitely many $\mathcal{R}$-steps. We write $\mathsf{SN}(\to_{\mathcal{R}\text{top}}/\to_{\mathcal{S}})$ if $\mathcal{R}$-steps are only allowed at the root position. Such relative top termination plays a crucial role in the dependency pair setting as $\mathsf{SN}(\to_{\mathcal{R}})$ is equivalent to $\mathsf{SN}(\to_{\mathsf{DP}(\mathcal{R})_{\text{top}}}/\to_{\mathcal{R}})$. Up to presentation details the following theorem is the one-sorted version of the main theorem for the matrix interpretations from [2, Theorem 2].

Theorem 2. *Let R, R', S, S' be TRSs over a signature Σ.*

1. *Let $(A, [\cdot], >, \gtrsim)$ be an extended monotone Σ-algebra such that $[\ell, \alpha] \gtrsim [r, \alpha]$ for every rule $\ell \to r$ in $\mathcal{R} \cup \mathcal{S}$ and $[\ell, \alpha] > [r, \alpha]$ for every rule $\ell \to r$ in $R' \cup S'$, for every $\alpha : \mathcal{X} \to A$.*
   Then $\mathsf{SN}(\to_{\mathcal{R}}/\to_{\mathcal{S}})$ implies $\mathsf{SN}(\to_{\mathcal{R}} \cup \to_{\mathcal{R}'} / \to_{\mathcal{S}} \cup \to_{\mathcal{S}'})$.
2. *Let $(A, [\cdot], >, \gtrsim)$ be a weakly monotone Σ-algebra such that $[\ell, \alpha] \gtrsim [r, \alpha]$ for every rule $\ell \to r$ in $\mathcal{R} \cup \mathcal{S}$ and $[\ell, \alpha] > [r, \alpha]$ for every rule $\ell \to r$ in $\mathcal{R}'$, for every $\alpha : \mathcal{X} \to A$.*
   Then $\mathsf{SN}(\to_{\mathcal{R}\text{top}}/\to_{\mathcal{S}})$ implies $\mathsf{SN}((\to_{\mathcal{R}} \cup \to_{\mathcal{R}'})_{\text{top}}/\to_{\mathcal{S}})$.

We obviously have $\mathsf{SN}(\to_{\mathcal{R}}/\to_{\mathcal{S}})$ if and only if the relation $\to_{\mathcal{S}}{}^* \cdot \to_{\mathcal{R}}$ is well-founded. Our formalized proof of Theorem 2 is constructive (hence slightly differs from the proof in [2]) and is based on the following lemma.

Lemma 3. *Let $\to_{\mathcal{R}}, \to_{\mathcal{S}}, \to_{\mathcal{R}}{}', \to_{\mathcal{S}}{}'$ be binary relations for which $\to_{\mathcal{S}}{}^* \cdot \to_{\mathcal{R}}$ and $(\to_{\mathcal{R}} \cup \to_{\mathcal{S}})^* \cdot (\to_{\mathcal{R}}{}' \cup \to_{\mathcal{S}}{}')$ are well-founded. Then $(\to_{\mathcal{S}} \cup \to_{\mathcal{S}}{}')^* \cdot (\to_{\mathcal{R}} \cup \to_{\mathcal{R}}{}')$ is well-founded.*

When thinking in terms of infinite sequences this lemma can easily be proven by truncating the initial part of such, supposedly, infinite sequences and observing that the remaining part must be finite. Working in the constructive setting of Coq forces us to use a slightly different kind of reasoning where the focus is on providing a relation that is decreasing along the sequence and performing induction with respect to it. Our experience shows, however, that for typical properties on well-foundedness, including this lemma, the ingredients of reasoning with infinite sequences can be recognized in such constructive proofs.

3 Matrix interpretations

In this section we show how monotone algebras are instantiated for the matrix interpretations with a fixed dimension d.

For the interpretation $[f]$ of a symbol $f \in \Sigma$ of arity n we choose n matrices $F_1, F_2, \ldots, F_n$ over $\mathbf{N}$, each of size $d \times d$, such that the upper left elements $(F_i)_{1,1}$ are positive for all $i = 1, 2, \ldots, n$, and a vector $\boldsymbol{f} \in \mathbf{N}^d$. Now we define $[f](\boldsymbol{v_1}, \ldots, \boldsymbol{v_n}) = F_1\boldsymbol{v_1} + \cdots + F_n\boldsymbol{v_n} + \boldsymbol{f}$ for all $\boldsymbol{v_1}, \ldots, \boldsymbol{v_n} \in A$.

So we fix a monotone algebra with $A = \mathbf{N}^d$, interpretations $[\cdot]$ defined as above and we use the following orders on algebra elements:

$$\begin{aligned}
(u_1, \ldots, u_d) \gtrsim (v_1, \ldots, v_d) &\iff \forall i : u_i \geq_{\mathbf{N}} v_i \\
(u_1, \ldots, u_d) > (v_1, \ldots, v_d) &\iff (u_1, \ldots, u_d) \gtrsim (v_1, \ldots, v_d) \wedge u_1 >_{\mathbf{N}} v_1
\end{aligned}$$

Let $x_1, \ldots, x_k$ be the variables occurring in ℓ, r. Then due to the linear shape of the functions $[f]$ we can compute matrices $L_1, \ldots, L_k, R_1, \ldots, R_k$ and vectors $\boldsymbol{l}, \boldsymbol{r}$ such that

$$\begin{aligned}
[\ell, \alpha] &= L_1\boldsymbol{x_1} + \cdots + L_k\boldsymbol{x_k} + \boldsymbol{l} \\
[r, \alpha] &= R_1\boldsymbol{x_1} + \cdots + R_k\boldsymbol{x_k} + \boldsymbol{r}
\end{aligned}$$

where $\alpha(x_i) = \boldsymbol{x_i}$ for $i = 1, \ldots, k$.

For matrices $B, C \in \mathbf{N}^{d \times d}$ write

$$B \succcurlyeq C \iff \forall i, j : (B)_{i,j} \geq (C)_{i,j}.$$

The following lemma provides a decision procedure for orders $>$ and $\gtrsim$ lifted to terms as required by the conditions of Theorem 2.

Lemma 4. *Let ℓ, r be terms and let matrices $L_1, \ldots, L_k, R_1, \ldots, R_k$ and vectors $\boldsymbol{l}, \boldsymbol{r}$ be defined as above. Then:*

- $\forall \alpha : \mathcal{X} \to A, [\ell, \alpha] \gtrsim [r, \alpha] \iff \boldsymbol{l} \gtrsim \boldsymbol{r} \wedge \forall i : L_i \succcurlyeq R_i$, *and*
- $\forall \alpha : \mathcal{X} \to A, [\ell, \alpha] > [r, \alpha] \iff \boldsymbol{l} > \boldsymbol{r} \wedge \forall i : L_i \succcurlyeq R_i$.

4 Coq formalization

In this section we briefly outline the Coq formalization. It consists of four parts that we will discuss in turn: the library on matrices, the theory of monotone algebras, the theory of the matrix interpretations and the extension of Rainbow to handle proofs using the matrix interpretation technique.

We developed a simple library on matrices as a functor, which takes a semiring of coefficients as an argument and produces a structure of matrices of arbitrary dimension with such coefficients along with basic operations (matrix construction, addition, multiplication, transposition etc.) and some basic properties (commutativity of addition, associativity of multiplication etc.). Matrices are represented as vectors of vectors which enables us to reuse the rich library on vectors that is part of CoLoR.

The theory of monotone algebras is developed as another functor that takes a carrier set A, a signature Σ with interpretations for all $f \in \Sigma$, two relations $>$ and $\gtrsim$ (which are typically orders but this is not required) and all properties as presented in Section 2. To be able to deal with concrete examples we need one additional ingredient: a decision procedure for liftings of $>$ and $\gtrsim$ to terms, as we must be able to check whether a given rule is (weakly) oriented. In fact instead of requiring a decision procedure for such relations we require it for some subsets of them, which are then used to prove termination. This is because in some instances the relations in question are undecidable (for example for non-linear polynomial interpretations) and in this way we are able to use some heuristics in such cases. For the matrix interpretations the intended relations are decidable, due to Lemma 4, but this allows us not to prove completeness of this characteristic (so we only had to prove the 'if' parts of this lemma). The module generated by this functor contains all the results (including Theorem 2) and Coq tactics needed to prove termination of concrete examples.

Finally the matrix interpretations are built as yet another functor that expects as argument a structure containing: a signature Σ, a dimension d for vectors and matrices and the matrix interpretations for all $f \in \Sigma$. It instantiates monotone algebras to matrix interpretations of dimension d. The main difficulty is providing a decision procedure for the intended relations, which accounts for proving the 'if' part of Lemma 4.

Finally we extended the Rainbow XML proof format with a format for the matrix interpretations proofs and with the ability to translate such proofs to actual Coq proofs. We added support for this format to the termination prover TPA [3] and with its help obtained certified termination proofs for 237 TRSs from TPDB v. 3.2. By expressing the CoLoR implementation of polynomial interpretations, by Sébastien Hinderer, in the setting of monotone algebras and by evaluating (on the same problem set) matrix interpretations combined with (non-linear) polynomial interpretations, 275 proofs were obtained. The average time for verification of the correctness of a proof by Coq (excluding proof search by TPA) was 5 seconds.

References

1. F. Blanqui, S. Coupet-Grimal, W. Delobel, S. Hinderer and A. Koprowski, CoLoR, a Coq Library on Rewriting and termination. In *WST '06.*
2. J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. In *IJCAR '06*, volume 4130 of *LNCS*, pages 574–588.
3. A. Koprowski TPA: Termination Proved Automatically. In *RTA '06*, volume 4098 of *LNCS*, pages 257–266.

Termination of TRSs with Bounded Increase[⋆]

Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp

LuFG Informatik 2, RWTH Aachen, Germany,
{giesl,thiemann,swiderski,psk}@informatik.rwth-aachen.de

1 Dependency Pairs and Integer Polynomial Orders

All termination tools for TRSs fail on examples like the following one, where an argument is *increased* in recursive calls repeatedly until it reaches a bound.

$$\begin{array}{rlrl} \mathsf{minus}(x,y) &\to \mathsf{cond}(\mathsf{gt}(x,y),x,y) & \mathsf{gt}(\mathsf{0},v) &\to \mathsf{false}\\ \mathsf{cond}(\mathsf{false},x,y) &\to \mathsf{0} & \mathsf{gt}(\mathsf{s}(u),\mathsf{0}) &\to \mathsf{true}\\ \mathsf{cond}(\mathsf{true},x,y) &\to \mathsf{s}(\mathsf{minus}(x,\mathsf{s}(y))) & \mathsf{gt}(\mathsf{s}(u),\mathsf{s}(v)) &\to \mathsf{gt}(u,v) \end{array}$$

We present a method to prove innermost termination of such TRSs by polynomial interpretations on *integers* and the *dependency pair (DP) framework* [1, 2]. We have the following DPs, where MINUS is the tuple symbol for minus, etc.

$$\begin{array}{rlrrlr} \mathsf{MINUS}(x,y) &\to \mathsf{COND}(\mathsf{gt}(x,y),x,y) & (1) & \mathsf{COND}(\mathsf{true},x,y) &\to \mathsf{MINUS}(x,\mathsf{s}(y)) & (3)\\ \mathsf{MINUS}(x,y) &\to \mathsf{GT}(x,y) & (2) & \mathsf{GT}(\mathsf{s}(u),\mathsf{s}(v)) &\to \mathsf{GT}(u,v) & (4) \end{array}$$

DPs are often used together with *polynomial interpretations* $\mathcal{P}ol$ which map every n-ary function symbol f to a polynomial $f_{\mathcal{P}ol}$ over n variables $x_1, ..., x_n$. Here, we map tuple symbols to polynomials with *integer* coefficients and all other function symbols to polynomials with *natural* coefficients. This mapping is extended to terms by $[x]_{\mathcal{P}ol} = x$ for all variables x and by $[f(t_1, \ldots, t_n)]_{\mathcal{P}ol} = f_{\mathcal{P}ol}([t_1]_{\mathcal{P}ol}, \ldots, [t_n]_{\mathcal{P}ol})$. Now $s \succ_{\mathcal{P}ol} t$ (resp. $s \succsim_{\mathcal{P}ol} t$) iff $[s]_{\mathcal{P}ol} > [t]_{\mathcal{P}ol}$ (resp. $[s]_{\mathcal{P}ol} \geq [t]_{\mathcal{P}ol}$) holds for all instantiations of the variables with *natural* numbers.

To prove innermost termination, the set of DPs $\mathcal{P}$ is simplified repeatedly by a *reduction pair processor* (RPP). If the processor finally removes all DPs, then innermost termination is proved. Our new RPP transforms a set of DPs into *two* new sets. The first set results from removing all strictly decreasing DPs. The second set results from removing all DPs from $\mathcal{P}$ that are *bounded from below.*

Definition 1 (RPP). *Let* $\mathcal{P}_{bound} = \{s \to t \in \mathcal{P} \mid s \succsim_{\mathcal{P}ol} \mathsf{c}\}$ *for a fresh constant* c.

$$Proc(\mathcal{P}) = \begin{cases} \{\, \mathcal{P} \setminus \succ_{\mathcal{P}ol},\ \mathcal{P} \setminus \mathcal{P}_{bound} \,\} & \text{if } \mathcal{P} \subseteq \succ_{\mathcal{P}ol} \cup \succsim_{\mathcal{P}ol} \text{ and } \mathcal{U}(\mathcal{P}) \subseteq \succsim_{\mathcal{P}ol}\\ \{\, \mathcal{P} \,\} & \text{otherwise} \end{cases}$$

Here, $\mathcal{U}(\mathcal{P})$ are the *usable rules* [1, 2] whose definition has to be adapted by taking into account which positions are ignored by the quasi-order $\succsim_{\mathcal{P}ol}$ and on which positions $\succsim_{\mathcal{P}ol}$ is monotonically increasing or decreasing (see [4] for details).

Let $\mathsf{GT}_{\mathcal{P}ol_1} = x_1$, $\mathsf{MINUS}_{\mathcal{P}ol_1} = x_1{+}1$, $\mathsf{COND}_{\mathcal{P}ol_1} = x_2{+}1$, $\mathsf{s}_{\mathcal{P}ol_1} = x_1{+}1$, and $f_{\mathcal{P}ol_1} = 0$ for all other symbols f. Then all DPs in $\mathcal{P} = \{(1), ..., (4)\}$ are bounded

⋆ Supported by the Deutsche Forschungsgemeinschaft DFG under grant GI 274/5-1 and by the DFG Research Training Group 1298 (*AlgoSyn*).

(i.e., $\mathcal{P} \setminus \mathcal{P}_{bound} = \varnothing$) and weakly decreasing. Since (2) and (4) are even strictly decreasing, they can be removed. Hence, the RPP results in $\mathcal{P} \setminus \succ_{\mathcal{P}ol_1} = \{(1), (3)\}$.

For these remaining DPs, we want to use $\mathcal{P}ol_2$ where $\mathsf{MINUS}_{\mathcal{P}ol_2} = x_1 - x_2$, $\mathsf{COND}_{\mathcal{P}ol_2} = x_2 - x_3$, $\mathsf{s}_{\mathcal{P}ol_2} = x_1 + 1$, and $f_{\mathcal{P}ol_2} = 0$ for all other symbols f. Now (3) would be strictly decreasing but none of the two DPs would be bounded (i.e., $\mathsf{MINUS}(x, y) \not\succsim_{\mathcal{P}ol_2} \mathsf{c}$ and $\mathsf{COND}(\mathsf{true}, x, y) \not\succsim_{\mathcal{P}ol_2} \mathsf{c}$). Thus, the RPP would return $\{(1)\}$ and $\{(1), (3)\}$, i.e., it would not simplify the problem.

2 Conditions for Bounded Increase

The solution is to consider *conditions* when requiring inequalities. For example, to include the DP (1) in $\mathcal{P}_{bound}$, we do not have to demand $\mathsf{MINUS}(x, y) \succsim_{\mathcal{P}ol_2} \mathsf{c}$ for *all* instantiations of x and y. It suffices to require this inequality only for instantiations σ where (1)'s instantiated right-hand side $\mathsf{COND}(\mathsf{gt}(x, y), x, y)\sigma$ reduces to an instantiated left-hand side $u\sigma$ for some (variable renamed) DP $u \to v$. By considering all possibilities $u \to v \in \{(1), (3)\}$, we get the following two constraints. If both are valid,[1] then we can include (1) in $\mathcal{P}_{bound}$.

$$\mathsf{COND}(\mathsf{gt}(x, y), x, y) = \mathsf{MINUS}(x', y') \quad \Rightarrow \quad \mathsf{MINUS}(x, y) \succsim_{\mathcal{P}ol_2} \mathsf{c} \tag{5}$$
$$\mathsf{COND}(\mathsf{gt}(x, y), x, y) = \mathsf{COND}(\mathsf{true}, x', y') \quad \Rightarrow \quad \mathsf{MINUS}(x, y) \succsim_{\mathcal{P}ol_2} \mathsf{c} \tag{6}$$

We now introduce a calculus to simplify conditional constraints. For example, (5) is trivially valid, since its condition is unsatisfiable. The reason is that COND is a *tuple symbol* and hence a *constructor*. Therefore, COND-terms can only be reduced to COND-terms.

I. **Constructor and Different Function Symbol**

$$\frac{f(p_1, ..., p_n) = g(q_1, ..., q_m) \wedge \varphi \;\Rightarrow\; \psi}{\mathit{TRUE}} \quad \text{if } f \text{ is a constructor and } f \neq g$$

Rule (II) handles conditions like $\mathsf{COND}(\mathsf{gt}(x, y), x, y) = \mathsf{COND}(\mathsf{true}, x', y')$ where both terms start with the constructor COND. So (6) is transformed to

$$\mathsf{gt}(x, y) = \mathsf{true} \wedge x = x' \wedge y = y' \quad \Rightarrow \quad \mathsf{MINUS}(x, y) \succsim_{\mathcal{P}ol_2} \mathsf{c} \tag{7}$$

II. **Same Constructors on Both Sides**

$$\frac{f(p_1, ..., p_n) = f(q_1, ..., q_n) \wedge \varphi \;\Rightarrow\; \psi}{p_1 = q_1 \wedge \ldots \wedge p_n = q_n \wedge \varphi \;\Rightarrow\; \psi} \quad \text{if } f \text{ is a constructor}$$

Rule (III) removes conditions of the form "$x = q$" or "$q = x$" by applying the substitution $[x/q]$ to the constraint. So (7) is transformed to

$$\mathsf{gt}(x, y) = \mathsf{true} \quad \Rightarrow \quad \mathsf{MINUS}(x, y) \succsim_{\mathcal{P}ol_2} \mathsf{c} \tag{8}$$

III. **Variable in Equation**

$$\frac{x = q \wedge \varphi \;\Rightarrow\; \psi}{\varphi\,\sigma \;\Rightarrow\; \psi\,\sigma} \quad \text{if } x \in \mathcal{V} \text{ and } \sigma = [x/q] \qquad \frac{q = x \wedge \varphi \;\Rightarrow\; \psi}{\varphi\,\sigma \;\Rightarrow\; \psi\,\sigma} \quad \text{if } x \in \mathcal{V}, q \text{ has no defined symbols, } \sigma = [x/q]$$

[1] We refer to [4] for the exact semantics of such conditional constraints.

Of course, one can omit any conjunct from the premise of an implication.

IV. **Delete Conditions**

$$\frac{\varphi \;\Rightarrow\; \psi}{\varphi' \;\Rightarrow\; \psi} \quad \text{if } \varphi' \subseteq \varphi \;\; \text{(where we regard a conjunction as a set of formulas)}$$

So one can also transform (7) to (8) by Rule (IV). (8) means that $\mathsf{MINUS}(x, y)\sigma \succsim_{\mathcal{P}ol_2} \mathsf{c}$ holds whenever $\mathsf{gt}(x,y)\sigma$ is innermost terminating and $\mathsf{gt}(x,y)\sigma \xrightarrow{\mathrm{i}}{}^{*}_{\mathcal{R}}$ true holds for a normal substitution σ. To simplify (8) further, the next inference rule performs an *induction* on the length of $\mathsf{gt}(x,y)\sigma$'s reduction. As $\mathsf{gt}(x,y)$ and true do not unify, at least one reduction step is needed. To detect all possibilities for the first reduction step, we consider all *narrowings* of the term $\mathsf{gt}(x,y)$:

$$\mathsf{gt}(x,y) \rightsquigarrow_{[x/\mathsf{0},y/v]} \mathsf{false}, \quad \mathsf{gt}(x,y) \rightsquigarrow_{[x/\mathsf{s}(u),y/\mathsf{0}]} \mathsf{true}, \quad \mathsf{gt}(x,y) \rightsquigarrow_{[x/\mathsf{s}(u),y/\mathsf{s}(v)]} \mathsf{gt}(u,v)$$

Thus, we could replace (8) by the following three new constraints where we always apply the respective narrowing substitution to the whole constraint:

$$\mathsf{false} = \mathsf{true} \;\Rightarrow\; \mathsf{MINUS}(\mathsf{0}, v) \succsim_{\mathcal{P}ol_2} \mathsf{c} \tag{9}$$
$$\mathsf{true} = \mathsf{true} \;\Rightarrow\; \mathsf{MINUS}(\mathsf{s}(u), \mathsf{0}) \succsim_{\mathcal{P}ol_2} \mathsf{c} \tag{10}$$
$$\mathsf{gt}(u,v) = \mathsf{true} \;\Rightarrow\; \mathsf{MINUS}(\mathsf{s}(u), \mathsf{s}(v)) \succsim_{\mathcal{P}ol_2} \mathsf{c} \tag{11}$$

So to transform a constraint $f(x_1,\ldots,x_n) = q \wedge \varphi \;\Rightarrow\; \psi$, we consider all rules $f(\ell_1,\ldots,\ell_n) \to r$. Then the constraint could be replaced by the new constraints

$$r = q\sigma \wedge \varphi\sigma \;\Rightarrow\; \psi\sigma, \qquad \text{where } \sigma = [x_1/\ell_1,\ldots,x_n/\ell_n]. \tag{12}$$

But we perform a better transformation. If r contains a recursive call $f(r_1,...,r_n)$, then $f(r_1,...,r_n)\sigma$'s reduction is shorter than the reduction of $f(x_1,...,x_n)\sigma$. So for $\mu = [x_1/r_1,...,x_n/r_n]$ we assume

$$\forall y_1,\ldots,y_m \;\; f(r_1,\ldots,r_n) = q\mu \wedge \varphi\mu \;\Rightarrow\; \psi\mu \tag{13}$$

as *induction hypothesis* when requiring (12). Here, $y_1,\ldots,y_m$ are all occurring variables except those in r. Of course, we may assume that variables in rewrite rules (i.e., in r) are disjoint from variables in constraints (i.e., in q, φ, and ψ). So instead of (12), it suffices to demand (13) $\Rightarrow$ (12), or equivalently

$$r = q\sigma \wedge \varphi\sigma \wedge (13) \;\Rightarrow\; \psi\sigma. \tag{14}$$

V. **Induction (Defined Symbol with Pairwise Different Variables)**

$$\frac{f(x_1,...,x_n) = q \quad \wedge\; \varphi \qquad \Rightarrow\; \psi}{\displaystyle\bigwedge_{f(\ell_1,...,\ell_n)\to r\in\mathcal{R}} (\; r = q\,\sigma \wedge \varphi\,\sigma \wedge \varphi' \;\Rightarrow\; \psi\,\sigma\;)} \quad \text{if } f \text{ is a defined symbol and } f(x_1,...,x_n) \text{ does not unify with } q$$

Here, $x_1,\ldots,x_n$ are pairwise different variables, $\sigma = [x_1/\ell_1,...,x_n/\ell_n]$, and $\varphi' =$ (13) if r contains $f(r_1,...,r_n)$, there is no defined symbol in any r_i, and $y_1,...,y_m$ are all occurring variables except $\mathcal{V}(r)$. Otherwise, we define $\varphi' = \mathit{TRUE}$.

So Rule (V) transforms (8) into (9), (10), and (15). Here, (15) results from the narrowing step $\mathsf{gt}(x,y) \rightsquigarrow_{[x/\mathsf{s}(u),y/\mathsf{s}(v)]} \mathsf{gt}(u,v)$, i.e., we have $\sigma = [x/\mathsf{s}(u), y/\mathsf{s}(v)]$, $r_1 = u$, $r_2 = v$, and $\mu = [x/u, y/v]$. There are no variables $y_1,\ldots,y_m$.

$$\begin{array}{l}\mathsf{gt}(u,v) = \mathsf{true} \\ \wedge\; (\mathsf{gt}(u,v) = \mathsf{true} \;\Rightarrow\; \mathsf{MINUS}(u,v) \succsim_{\mathcal{P}ol_2} \mathsf{c}) \;\Rightarrow\; \mathsf{MINUS}(\mathsf{s}(u),\mathsf{s}(v)) \succsim_{\mathcal{P}ol_2} \mathsf{c}\end{array} \tag{15}$$

To simplify (15) further, now we can "apply" the induction hypothesis, since its condition $\mathsf{gt}(u,v) = \mathsf{true}$ is guaranteed to hold. So we can transform (15) to

$$\mathsf{gt}(u,v) = \mathsf{true} \;\wedge\; \mathsf{MINUS}(u,v) \succsim_{\mathcal{P}ol_2} \mathsf{c} \;\Rightarrow\; \mathsf{MINUS}(\mathsf{s}(u),\mathsf{s}(v)) \succsim_{\mathcal{P}ol_2} \mathsf{c}.$$

VI. **Simplify Condition**

$$\frac{\varphi \wedge (\forall y_1,\ldots,y_m \;\; \varphi' \Rightarrow \psi' \;) \;\Rightarrow\; \psi}{\varphi \wedge \qquad\qquad\qquad \psi'\,\sigma \;\Rightarrow\; \psi}$$

if $DOM(\sigma) \subseteq \{y_1,\ldots,y_m\}$, there is no defined symbol in any $\sigma(y_i)$, and $\varphi'\,\sigma \subseteq \varphi$

To simplify the remaining constraints (9), (10), (15), note that (9) can be eliminated by Rule (I) as its condition $\mathsf{false} = \mathsf{true}$ is unsatisfiable. Moreover, Rule (II) can delete the trivial condition $\mathsf{true} = \mathsf{true}$ of (10). For (15), with Rule (IV) one can always omit conditions like $\mathsf{gt}(u,v) = \mathsf{true}$ from constraints. In this way, all conditions with equalities $p = q$ are removed in the end.

So to finish the termination proof of our example, we can include the DP (1) in $\mathcal{P}_{bound}$ if the constraints $\mathsf{MINUS}(\mathsf{s}(u),\mathsf{0}) \succsim_{\mathcal{P}ol_2} \mathsf{c}$ and $\mathsf{MINUS}(u,v) \succsim_{\mathcal{P}ol_2} \mathsf{c} \Rightarrow \mathsf{MINUS}(\mathsf{s}(u),\mathsf{s}(v)) \succsim_{\mathcal{P}ol_2} \mathsf{c}$ are satisfied. Of course, these constraints obviously hold if we choose $\mathsf{c}_{\mathcal{P}ol_2} \leq 1$. Then the DP (3) is strictly decreasing and (1) is bounded from below and thus, the RPP transforms the remaining set of DPs $\{(1),(3)\}$ into $\{(1)\}$ and $\{(3)\}$. These resulting sets of DP are easy to handle by another application of the RPP and thus, innermost termination is proved.

3 Conclusion

We have extended the reduction pair processor of the DP method in order to handle TRSs that terminate because of bounded increase. To be able to measure the *increase* of arguments, we used *integer* polynomial interpretations. Moreover, to exploit the *bounds* given by conditions, we developed a calculus based on induction which simplifies the constraints needed for the reduction pair processor.

We implemented the new reduction pair processor in our termination prover AProVE [3]. To demonstrate the power of our method, the full version of this paper [4] contains a collection of typical TRSs with bounded increase where all existing tools failed up to now. But with the results from this paper, AProVE can prove innermost termination for all of them. See `http://aprove.informatik.rwth-aachen.de/eval/Increasing/` to experiment with our implementation.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
2. J. Giesl, R. Thiemann, P. Schneider-Kamp. The DP framework: Combining Techniques for Automated Termination Proofs. *Proc. LPAR'04*, LNAI 3452, pages 301-331, 2005.
3. J. Giesl, P. Schneider-Kamp, R. Thiemann. AProVE 1.2: Automatic termination proofs in the DP framework. In *Proc. IJCAR '06*, LNAI 4130, pages 281–286, 2006.
4. J. Giesl, R. Thiemann, S. Swiderski, and P. Schneider-Kamp. Proving termination by bounded increase. In *Proc. CADE '07*, LNAI, 2007. To appear. Extended version is available as Technical Report AIB 2007-03, RWTH Aachen, at `http://aib.informatik.rwth-aachen.de`.

Root Stabilisation Using Dependency Pairs

Jörg Endrullis[1] and Jeroen Ketema[2]

[1] Department of Computer Science, Vrije Universiteit Amsterdam
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
joerg@few.vu.nl

[2] Research Institute of Electrical Communication, Tohoku University
2-1-1 Katahira, Aoba-ku, Sendai 980-8577, Japan
jketema@nue.riec.tohoku.ac.jp

1 Introduction

A dependency pair problem [1] is elegantly formulated as a relative termination problem [2]: For a TRS $\mathcal{R} = (\Sigma, R)$, does $\to_{\mathcal{DP}(R)}$ terminate relative to $\to_R$ when the steps from $\to_{\mathcal{DP}(R)}$ occur as *root steps* and the steps from $\to_R$ occur as *non-root steps*? In other words, is $\to_{\mathcal{DP}(\mathcal{R})}$ *root stabilising relative to* $\to_{\mathcal{R}}$?

The current paper is concerned with the application of a number of methods from the dependency pair approach — reduction pairs, dependency graphs, and labelling of defined symbols — in the more general setting of relative root stabilisation. This is interesting for at least three reasons:

- Enhancement of known techniques for proving relative root stabilisation, e.g. matrix interpretations [2].
- Identification of those methods from the dependency pair approach which may have wider application.
- Strong normalisation [3] of finite terms in infinitary TRSs with finite sets of rules, which is root stabilisation in case of left-linear systems due to compression, i.e. every reduction is equivalent to one of length at most ω [4].

2 Preliminaries

We assume familiarity with rewriting [4] and dependency pairs [1]. Below, Σ denotes a signature, V denotes a countable set of variables, and $\mathcal{T}er(\Sigma, V)$ denotes the set of terms over Σ and V; $\mathcal{R} = (\Sigma, R)$ and $\mathcal{S} = (\Sigma, S)$ denote arbitrary TRSs. A *reduction pair* is a pair $(\gtrsim, \succ)$ with $\gtrsim$ a quasi-rewrite order and $\succ$ a stable, not necessarily monotonic, well-founded order compatible with $\gtrsim$.

A binary relation $\to_1$ is called *terminating relative to* $\to_2$ iff $\mathsf{SN}(\to_1/\to_2)$, i.e. $\mathsf{SN}(\to_2^* \cdot \to_1 \cdot \to_2^*)$. A step $\to$ is a *root step* $\hat{\to}$ if it contracts a redex at the root, otherwise it is a *non-root step* $\check{\to}$. Obviously, $\to \; = \; \hat{\to} \uplus \check{\to}$. A TRS $\mathcal{R}$ is *root stabilising relative to* $\mathcal{S}$ iff $\mathsf{SN}(\hat{\to}_R/\check{\to}_S)$; $\mathcal{R}$ is *root stabilising* iff it is root stabilising relative to itself, i.e. no reductions with infinitely many root steps occur. A term is *root stable* if no root steps occur in any reduction starting from the term.

Remark 2.1. In a root stabilising TRS, a term may admit infinite reductions and it may require an arbitrary, finite number of root steps to obtain a root stable term. For example, the TRS with the rules $\{a \to g(a),\ f(h(x)) \to f(x),\ g(a) \to h(b),\ g(h(x)) \to h(h(x))\}$ is root stabilising and admits an infinite reduction starting from $f(a)$:

$$f(a) \to f(g(a)) \to f(g^2(a)) \to \cdots \to f(g^n(a)) \to \cdots .$$

For each $n \in \mathbb{N}$, $f(g^n(a))$ is reducible to $f(h^n(b))$ and n root steps are required to reduce $f(h^n(b))$ to the root stable term $f(b)$.

3 Reduction Pairs

We relate relative root stabilisation and reduction pairs. We have the following theorem, which is close to the main theorem of the dependency pair approach [1]:

Theorem 3.1. *It holds that* $\mathsf{SN}(\overset{\wedge}{\to}_R/\overset{\vee}{\to}_S)$ *iff there exists a reduction pair* $(\succsim, \succ)$ *such that* $S \subseteq \succsim$ *and* $R \subseteq \succ$*.*

Note that $\mathcal{R}$ is root stabilising iff there exists a reduction pair $(\succsim, \succ)$ such that $R \subseteq \succsim \cap \succ$. As usual, we like to weaken proof obligations step-by-step:

Theorem 3.2. *Let* $(\succsim, \succ)$ *be a reduction pair with* $S \subseteq \succsim$ *and* $R \subseteq \succsim \cup \succ$*. It holds that* $\mathsf{SN}(\overset{\wedge}{\to}_R/\overset{\vee}{\to}_S)$ *iff* $\mathsf{SN}(\overset{\wedge}{\to}_{R-\succ}/\overset{\vee}{\to}_S)$*.*

Example 3.3. Consider $R = \{f(g(x), y) \to f(x, f(g(x), y))\}$ [5, Example 4.4], we show root stabilisation, i.e. $\mathsf{SN}(\overset{\wedge}{\to}_R/\overset{\vee}{\to}_R)$. Using the polynomial interpretation $|f(x_1, x_2)| = |x_1|$ and $|g(x)| = 1 + |x|$ over the natural numbers, we obtain $|f(g(x), y)| = 1 + |x| > |x| = |f(x, f(g(x), y))|$. Thereby, $R \subseteq \succsim \cap \succ$ and, hence, R is root stabilising.

Given that argument filterings are defined as usual [1, 6] and that π is an arbitrary argument filtering, write the following:

$$\pi(R) = \{\pi(\ell) \to \pi(r) \mid \ell \to r \in R,\ \pi(\ell) \neq \pi(r)\}.$$

Theorem 3.4. *Let* π *be an argument filtering and* $(\succsim, \succ)$ *a reduction pair with* $R' = \{\ell \to r \in R \mid \pi(\ell) \succ \pi(r)\}$ *and* $\pi(S) \cup \pi(R - R') \subseteq \succsim$*. It holds that* $\mathsf{SN}(\overset{\wedge}{\to}_R/\overset{\vee}{\to}_S)$ *iff* $\mathsf{SN}(\overset{\wedge}{\to}_{R-R'}/\overset{\vee}{\to}_S)$*.*

The proofs of all the above theorems are standard [1, 2].

4 Dependency Graphs

The 'dependency pairs' of a root stabilisation problem $\mathsf{SN}(\overset{\wedge}{\to}_R/\overset{\vee}{\to}_S)$ are the rules of $\mathcal{R}$. The dependency graph provides information on the order in which $\overset{\wedge}{\to}_R$ steps can occur within a $\overset{\wedge}{\to}_R \cup \overset{\vee}{\to}_S$-reduction.

Definition 4.1. *The* dependency graph $\mathcal{DG}(\hat{\to}_R/\check{\to}_S)$ *is a graph that has as its nodes the rewrite rules of $\mathcal{R}$ and there is an edge from $s \to_R t$ to $u \to_R v$ iff there are substitutions σ and τ with $\sigma(t) \check{\to}_S^* \tau(u)$. A cycle $\mathcal{C}$ is a nonempty set of nodes such that there is a nonempty path from n to m for all $n, m \in \mathcal{C}$.*

An estimated dependency graph *of $\mathcal{R}$ and $\mathcal{S}$ is a graph that has as its nodes the rewrite rules of $\mathcal{R}$ and that has $\mathcal{DG}(\hat{\to}_R/\check{\to}_S)$ as a subgraph.*

The dependency graph is uncomputable in general. In case of dependency pair problems $\mathsf{SN}(\hat{\to}_{\mathcal{DP}(\mathcal{R})}/\check{\to}_R)$, the above definition coincides with the one from the dependency pair approach.

Example 4.2. Denote the set of rules from Remark 2.1 by R. The dependency graph $\mathcal{DG}(\hat{\to}_R/\check{\to}_R)$ is:

$$\begin{tikzcd}
 & f(h(x)) \to f(x) \arrow[loop above] & \\
a \to g(a) \arrow[dr] & & g(h(x)) \to h(h(x)) \\
 & g(a) \to h(b) &
\end{tikzcd}$$

Definition 4.3. *Let $\mathcal{C} \subseteq R$. An infinite $\hat{\to}_{\mathcal{C}} \cup \check{\to}_S$-reduction is called* $\mathcal{C}$-minimal *if all rules from $\mathcal{C}$ are applied infinitely often.*

Clearly the existence of a $\mathcal{C}$-minimal rewrite sequence implies that $\mathcal{C}$ is a cycle in the (estimated) dependency graph. Therefore, we have the following:

Theorem 4.4. *It holds that $\mathsf{SN}(\hat{\to}_R/\check{\to}_S)$ iff there is no cycle $\mathcal{C}$ in the (estimated) dependency graph for which there exists an $\mathcal{C}$-minimal rewrite sequence.*

In practice, considering *strongly connected components* [7] is preferred, as the number of cycles can grow exponentially with the number of nodes. A strongly connected component is a maximal cycle (with respect to inclusion).

Theorem 4.5. *It holds that $\mathsf{SN}(\hat{\to}_{\mathcal{R}}/\check{\to}_{\mathcal{S}})$ iff for each strongly connected component $\mathcal{C}$ in the (estimated) dependency graph $\mathsf{SN}(\hat{\to}_{\mathcal{C}}/\check{\to}_{\mathcal{S}})$.*

We use Theorem 4.5 together with dependency graph approximations [1, 7] to split proof obligations into simpler subtasks. These subtasks are root stabilisation problems to which we can apply Theorem 3.2 or 3.4 and Theorem 4.5, recursively.

Example 4.6. We revisit Example 4.2 and show $\mathsf{SN}(\hat{\to}_R/\check{\to}_R)$. The dependency graph contains only one strongly connected component $\mathcal{C} = \{f(h(x)) \to f(x)\}$. Using Theorem 4.5 the proof obligation reduces to $\mathsf{SN}(\hat{\to}_{\mathcal{C}}/\check{\to}_R)$. We choose an interpretation over the ordinal numbers[3] from $\omega \cdot 2$: $|a| = \omega$, $|b| = 0$, $|f(x)| = |x|$, $|g| = I_g(|x|)$ and $|h(x)| = |x| + 1$, where $I_g(\alpha) = \alpha + 1$ for $\alpha \neq \omega$ and $I_g(\omega) = \omega$. Thereby, $\mathcal{C} \subseteq \succ$ and $\mathcal{R} \subseteq \succsim$ and we have $\mathsf{SN}(\hat{\to}_{\mathcal{C}}/\check{\to}_R)$ by Theorem 3.2.

[3] An interpretation over $\mathbb{N}$ is not feasible, because $f(a)$ allows for an arbitrary number of root steps (see Remark 2.1).

5 Labelling Root Symbols

We define a *labelling of root symbols* for $\mathsf{SN}(\hat{\to}_R/\check{\to}_S)$. The *defined symbols* of $\mathcal{R}$ are those from $\mathcal{D} = \{root(\ell) \mid \ell \to r \in R\}$. For every $f \in \mathcal{D}$, let $f^\sharp$ be a fresh function symbol of same arity as f. If $t = f(t_1, \dots, t_n)$, then $t^\sharp$ stands for $f^\sharp(t_1, \dots, t_n)$.

Definition 5.1. *The* root labelling $\hat{\mathcal{L}}(R)$ *is defined as follows:*

$$\hat{\mathcal{L}}(R) = \bigcup_{\ell \to r \in R} \begin{cases} \{\ell^\sharp \to r^\sharp\} & \textit{if } root(r) \in \mathcal{D} \\ \{\tau(\ell)^\sharp \to \tau(r)^\sharp \mid f \in \mathcal{D},\, \tau = \sigma_{\ell \to x, f}\} & \textit{if } r = x \in V \end{cases}$$

where $\sigma_{\ell \to x, f} = \{x \mapsto f(x_1, \dots, x_n)\}$ *with* $x_1, \dots, x_n$ *pairwise different variables that do not occur in* ℓ.

Observe the close analogy with $\mathcal{DP}(R)$ from the dependency pairs approach. The definition has a case distinction, as rules might be collapsing. A similar case distinction occurs in [8], which deals with context sensitive dependency pairs.

Example 5.2. Consider $R = \{f(x) \to g(f(x)),\ g(f(x)) \to x\}$, we have:

$$\hat{\mathcal{L}}(R) = \{f^\sharp(x) \to g^\sharp(f(x)),\ g^\sharp(f(f(y))) \to f^\sharp(y),\ g^\sharp(f(g(y))) \to g^\sharp(y)\}\ .$$

We have the following theorem:

Theorem 5.3. *It holds that* $\mathsf{SN}(\hat{\to}_R/\check{\to}_S)$ *iff* $\mathsf{SN}(\hat{\to}_{\hat{\mathcal{L}}(R)}/\check{\to}_S)$.

The labelling separates the defined symbols of $\mathcal{R}$ from the symbols of $\mathcal{S}$. Hence, different interpretations may be assigned to the symbols. Moreover, all rules with a non-defined symbol at the root of the right-hand side are thrown away. This increases chances that root-stabilisation may be proved.

6 Subterm and Usable Rules Criteria

The subterm criterion of [9] and the usable rules criterion of [1] are no longer valid in the case of root stabilisation.

The subterm criterion states that for any cycle $\mathcal{C}$ in the dependency graph, if there exists a simple projection $\pi : \mathcal{D}_\mathcal{C} \to \mathbb{N}$ for the defined symbols in $\mathcal{C}$ such that $\pi(u) \trianglerighteq \pi(v)$ for all $u \to v \in \mathcal{C}$ and $\pi(u) \rhd \pi(v)$ for at least one $u \to v \in \mathcal{C}$, then there is no $\mathcal{C}$-minimal rewrite sequence.

Consider the root stabilisation problem $\mathsf{SN}(\hat{\to}_{\hat{\mathcal{L}}(R)}/\check{\to}_R)$ with:

$$R = \{a \to g(a),\ f(g(x)) \to f(x)\} \qquad \hat{\mathcal{L}}(R) = \{f^\sharp(g(x)) \to f^\sharp(x)\}$$

The dependency graph obviously has a cycle. Given the simple projection $\pi(f^\sharp) = 1$, we obtain $\pi(f^\sharp(g(x))) = g(x) \rhd x = \pi(f^\sharp(x))$. Hence, the subterm criterion is satisfied. However, we have a $\mathcal{C}$-minimal rewrite sequence:

$$f^\sharp(g(a)) \to f^\sharp(a) \to f^\sharp(g(a)) \to f^\sharp(a) \to f^\sharp(g(a)) \to \cdots\ .$$

The above example also shows that the usable rules criterion is not valid. The rule $a \to g(a)$ would not be considered as usable, but the remaining TRS is root stabilising.

7 Experimental Results

We implemented the above methods as part of Jambox. To analyse their effectiveness, we considered part of the Termination Problem Database 2006 (TPDB): the 103 non-terminating and 76 unknown problems from the TRS category of the Termination Competition 2006 [10], i.e. 179 TRSs in total.

Theorem 3.2 in combination with matrix interpretations [2] allowed to show root stabilisation of 42 of the TRSs.[4] Thereby, 5 proofs depended on matrix interpretations of dimensions 2 or 3; for the remaining 37 problems linear polynomial interpretations sufficed. Preprocessing the problems by labelling the root symbols (Theorem 5.3) increased the score to 46. Finally, using dependency graph approximations [7,11] and Theorem 4.5 often reduced the hardness of the problem, allowing for smaller matrix dimensions and allowing for 47 TRSs to be shown root stabilising. Of the 47 TRSs, 38 are known to be non-terminating and 9 remained unknown in the Termination Competition 2006. The generated proofs are available via: `http://infinity.few.vu.nl/wst07/`.

References

1. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. TCS **236** (2000) 133–178
2. Endrullis, J., Waldmann, J., Zantema, H.: Matrix interpretations for proving termination of term rewriting. In: IJCAR'06. Volume 4130 of LNAI. (2006) 574–588
3. Klop, J.W., de Vrijer, R.: Infinitary normalization. In Artëmov, S.N., Barringer, H., d'Avila Garcez, A.S., Lamb, L.C., Woods, J., eds.: We Will Show Them: Essays in Honour of Dov Gabbay. Volume 2. College Publications (2005) 169–192
4. Terese, ed.: Term Rewriting Systems. Cambridge University Press (2003)
5. Lucas, S.: Termination of context-sensitive rewriting by rewriting. In: ICALP'96. Volume 1099 of LNCS. (1996) 122–133
6. Kusakari, K., Nakamura, M., Toyama, Y.: Argument filtering transformation. In: PPDP'99. Volume 1702 of LNCS. (2004) 47–61
7. Hirokawa, N., Middeldorp, A.: Automating the dependency pair method. I&C **199** (2005) 172–199
8. Alarcón, B., Gutiérrez, R., Lucas, S.: Context-sensitive dependency pairs. In: FSTTCS'06. Volume 4337 of LNCS. (2006) 298–309
9. Hirokawa, N., Middeldorp, A.: Dependency pairs revisited. In: RTA 2004. Volume 3091 of LNCS. (2004) 249–268
10. Termination Competition: (`www.lri.fr/~marche/termination-competition/`)
11. Giesl, J., Thiemann, R., Schneider-Kamp, P.: Proving and disproving termination of higher-order functions. In: FroCoS 2005. Volume 3717 of LNCS. (2005) 216–231

[4] We did not employ Theorem 3.4, as choosing a zero matrix for an argument has the same effect as filtering on that argument.

Improving Termination Proofs of Context-Sensitive Rewriting using Dependency Pairs [*]

Beatriz Alarcón, Raúl Gutiérrez, and Salvador Lucas

DSIC, Universidad Politécnica de Valencia, Spain
{balarcon , rgutierrez , slucas}@dsic.upv.es

1 Introduction

Termination is one of the most interesting problems when dealing with *context-sensitive rewriting* (*CSR*) [Luc02]. With *CSR* we can *achieve* a terminating behavior with non-terminating Term Rewriting Systems (TRSs) by pruning (all) infinite rewrite sequences. Proving termination of *CSR* has been recently recognized as an interesting problem with several applications in the fields of term rewriting and programming languages (see [Luc06]). Several methods have been developed for proving termination of *CSR*. In particular, a number of transformations which permit to treat termination of *CSR* as a standard termination problem have been described (see [GM04,Luc06] for recent surveys). However, up to [AGL06], the *dependency pairs method* [AG00], one of the most powerful techniques for proving termination of rewriting, has not been investigated in connection with proofs of termination of *CSR*. In this paper we summarize the features and main contributions of this context-sensitive dependency pairs approach.

2 Context-Sensitive Dependency Pairs

The basic intuitions from the standard approach are valid for *CSR*, although some important differences arise. Basically, the subterms in the right-hand sides of the rules which are considered to build the CS-dependency pairs must be μ-*replacing* terms (where μ is a replacement map $\mu : \mathcal{F} \to \mathcal{P}(\mathbb{N})$ which indicates the arguments $\mu(f)$ of each symbol $f \in \mathcal{F}$ where rewritings are allowed). However, this is not sufficient to obtain a correct approximation.

Example 1. [AGL06, Example 2] Consider the following TRS $\mathcal{R}$:

```
a -> c(f(a))
f(c(X)) -> X
```

[*] This work has been partially supported by the EU (FEDER) and the Spanish MEC, under grants TIN 2004-7943-C04-02 and HA 2006-2007, and the Generalitat Valenciana under grant GV06/285. Beatriz Alarcón was partially supported by the Spanish MEC under FPU grant AP2005-3399.

together with $\mu(\mathtt{c}) = \varnothing$ and $\mu(\mathtt{f}) = \{1\}$. There is no μ-replacing subterm s in the right-hand sides of the rules which is rooted by a defined symbol. Thus, there is no 'standard' dependency pair. We could wrongly conclude that $\mathcal{R}$ is μ-terminating, which is not true:

$$\mathtt{f}(\underline{\mathtt{a}}) \hookrightarrow_\mu \underline{\mathtt{f(c(f(a)))}} \hookrightarrow_\mu \mathtt{f}(\underline{\mathtt{a}}) \hookrightarrow_\mu \cdots$$

As shown in [AGL06], the problem comes when a rule $l \to r$ has *migrating* variables which are those μ-replacing variables in r which are *not* μ-replacing in l. Then, we must add the following *collapsing* dependency pair `F(c(X)) -> X` which would not be allowed in Arts and Giesl's approach [AG00] because the right-hand side is a variable.

Definition 1. [AGL06] *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a TRS, $\mu \in M_\mathcal{R}$ and $\rhd_\mu$ denotes the strict μ-*replacing *subterm relation. We define $\mathsf{DP}(\mathcal{R}, \mu) = \mathsf{DP}_\mathcal{F}(\mathcal{R}, \mu) \cup \mathsf{DP}_\mathcal{X}(\mathcal{R}, \mu)$ to be the set of* context-sensitive *dependency pairs (CS-DPs) where:*

$$\mathsf{DP}_\mathcal{F}(\mathcal{R}, \mu) = \{l^\sharp \to s^\sharp \mid l \to r \in R, r \unrhd_\mu s, root(s) \in \mathcal{D}, l \ntriangleright_\mu s\}$$

and $\mathsf{DP}_\mathcal{X}(\mathcal{R}, \mu) = \{l^\sharp \to x \mid l \to r \in R, x \in \mathcal{V}ar^\mu(r) - \mathcal{V}ar^\mu(l)\}$. We extend $\mu \in M_\mathcal{F}$ into $\mu^\sharp \in M_{\mathcal{F}^\sharp}$ by $\mu^\sharp(f) = \mu(f)$ if $f \in \mathcal{F}$, and $\mu^\sharp(f^\sharp) = \mu(f)$ if $f \in \mathcal{D}$.

In order to capture termination of *CSR* with the CS-DPs in Definition 1 we introduce an appropriate notion of chain of CS-DPs.

Definition 2 (Chain of CS-DPs [AGL06]**).** *Let $(\mathcal{R}, \mu)$ be a CS-TRS. Given $\mathcal{P} \subseteq \mathsf{DP}(\mathcal{R}, \mu)$, an $(\mathcal{R}, \mathcal{P}, \mu^\sharp)$-chain is a finite or infinite sequence of pairs $u_i \to v_i \in \mathcal{P}$, for $i \geq 1$ such that there is a substitution σ satisfying both:*

1. *$\sigma(v_i) \hookrightarrow^*_{\mathcal{R},\mu^\sharp} \sigma(u_{i+1})$, if $u_i \to v_i \in \mathsf{DP}_\mathcal{F}(\mathcal{R}, \mu)$, and*
2. *if $u_i \to v_i = u_i \to x_i \in \mathsf{DP}_\mathcal{X}(\mathcal{R}, \mu)$, then there is $s_i \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ such that $\sigma(x_i) \unrhd_\mu s_i$ and $s_i^\sharp \hookrightarrow^*_{\mathcal{R},\mu^\sharp} \sigma(u_{i+1})$ (where $\unrhd_\mu$ is the μ-replacing subterm relation on $\mathcal{T}(\mathcal{F}, \mathcal{X})$).*

The first item in Definition 2 is close to the usual definition. The second item permits to deal with migrating variables which lead to the introduction of collapsing CS-dependency pairs. The following result shows that these definitions characterize termination of *CSR*.

Theorem 1 (Characterization of μ-termination [AGL06]**).** *Let $\mathcal{R}$ be a TRS and $\mu \in M_\mathcal{R}$. $\mathcal{R}$ is μ-terminating iff there is no infinite $(\mathcal{R}, \mathsf{DP}(\mathcal{R}, \mu), \mu^\sharp)$-chain.*

3 Checking termination of *CSR* with the CS-DPs

As in the standard approach, the analysis of infinite chains of CS-DPs can be made by looking at (the cycles $\mathfrak{C}$ of) the *context-sensitive dependency graph*(CS-DG) associated to the TRS $\mathcal{R}$. The nodes of the CS-DG are the dependency pairs in $\mathsf{DP}(\mathcal{R},\mu)$. The main difference with standard rewriting arises in the treatment of collapsing CS-DPs. In our first approach there was an arc from a dependency pair $u \to v \in \mathsf{DP}_{\mathcal{X}}(\mathcal{R},\mu)$ to *each* dependency pair $u' \to v' \in \mathsf{DP}(\mathcal{R},\mu)$ [AGL06]. In [AGL07], we have investigated how to improve this. The following notion turns to be essential to obtain a simpler graph which leads to a powerful CS-DP approach for proving termination of *CSR*.

Definition 3 (Hidden symbol [AGL07]). *Let $\mathcal{R} = (\mathcal{F}, R)$ be a TRS and $\mu \in M_{\mathcal{R}}$. We say that $f \in \mathcal{F}$ is a* hidden symbol *if it occurs at a non-μ-replacing position of a right-hand side of a rule. Let $\mathcal{H}(\mathcal{R},\mu)$ be the set of all hidden symbols in $(\mathcal{R},\mu)$.*

Definition 4 (Context-Sensitive Dependency Graph [AGL07]). *Let $\mathcal{R}$ be a TRS and $\mu \in M_{\mathcal{R}}$. The context-sensitive dependency graph consists of the set $\mathsf{DP}(\mathcal{R},\mu)$ of context-sensitive dependency pairs and arcs which connect them as follows:*

1. *There is an arc from $u \to v \in \mathsf{DP}_{\mathcal{F}}(\mathcal{R},\mu)$ to $u' \to v' \in \mathsf{DP}(\mathcal{R},\mu)$ if there are subtitutions σ and θ such that $\sigma(v) \hookrightarrow^*_{\mathcal{R},\mu^\sharp} \theta(u')$.*
2. *There is an arc from $u \to v \in \mathsf{DP}_{\mathcal{X}}(\mathcal{R},\mu)$ to $u' \to v' \in \mathsf{DP}(\mathcal{R},\mu)$ if $u' = f^\sharp(u_1,...,u_k)$ and $f \in \mathcal{H}(\mathcal{R},\mu)$.*

The definition of the estimated CS-DG is similar to the standard one (see [AGL06]). We also use narrowing of CS-DPs to refine the estimated CS-DG, improving therefore the proofs of μ-termination [AGL07]. The absence of infinite chains is checked by finding (possibly different) *μ-reduction pairs* $(\gtrsim, \sqsupset)$ for each cycle $\mathfrak{C}$ in the (estimated or 'narrowed') graph, where $\gtrsim$ is a stable and μ-monotonic quasi-ordering which is compatible with the well-founded and stable ordering $\sqsupset$, i.e., $\gtrsim \circ \sqsupset \subseteq \sqsupset$ or $\sqsupset \circ \gtrsim \subseteq \sqsupset$, thus relaxing the monotonicity requirements [AGL06]. We have also adapted the subterm criterion [HM07] to deal with *CSR* [AGL06].

4 Experiments

We have implemented these techniques as part of MU-TERM [AGIL07,Luc04]. In order to evaluate them, we have considered the examples in the Context-Sensitive Rewriting subcategory of the 2006 Termination Competition[1] which collects 90 examples of CS-TRSs. We have also used AProVE for proving termination of the examples. AProVE [GST06] is currently the most powerful tool for proving termination of TRSs and implements most existing results and techniques regarding DPs and related techniques. AProVE is able to prove termination of *CSR* by using *transformations*. Such transformations convert a context-sensitive rewriting termination problem into a rewriting termination problem, exploiting the huge amount of techniques for termination of TRSs. Further details about our benchmarks with MU-TERM and AProVE can be found here:

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

`http://www.dsic.upv.es/~rgutierrez/muterm/wst07/benchmarks.html`

The following table summarizes them:

	MU-TERM	AProVE
YES score	65	56
YES average time	1.54 sec.	4.74 sec.

According to these results, we can say that the current techniques developed for the CS-DP approach greatly improve on the use of transformations for proving termination of *CSR*.

References

[AG00] T. Arts and J. Giesl. Termination of Term Rewriting Using Dependency Pairs *Theoretical Computer Science*, 236:133-178, 2000.

[AGIL07] B. Alarcón, R. Gutiérrez, J. Iborra, and S. Lucas. Proving Termination of Context-Sensitive Rewriting with MU-TERM. *Electronic Notes in Theoretical Computer Science, to appear*, 2007.

[AGL06] B. Alarcón, R. Gutiérrez, and S. Lucas. Context-Sensitive Dependency Pairs. *Proc. of FSTTCS'06*, LNCS 4337:297-308, Springer-Verlag, Berlin, 2006.

[AGL07] B. Alarcón, R. Gutiérrez, and S. Lucas Improving the context-sensitive dependency graph. Electronic Notes in Theoretical Computer Science, to appear, 2007.

[GST06] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. *Proc. of IJCAR'06*, LNAI 4130:281-286, Springer-Verlag, Berlin, 2006.

[GM04] J. Giesl and A. Middeldorp. Transformation techniques for context-sensitive rewrite systems. *Journal of Functional Programming*, 14(4): 379-427, 2004.

[HM07] N. Hirokawa and A. Middeldorp. Tyrolean termination tool: Techniques and features. *Information and Computation*, 205:474-511, 2007.

[Luc02] S. Lucas. Context-sensitive rewriting strategies. *Information and Computation*, 178(1):293-343, 2002.

[Luc04] S. Lucas. MU-TERM: A Tool for Proving Termination of Context-Sensitive Rewriting *Proc. of RTA'04*, LNCS 3091:200-209, Springer-Verlag, Berlin, 2004. Available at `http://www.dsic.upv.es/~slucas/csr/termination/muterm`.

[Luc06] S. Lucas. Proving termination of context-sensitive rewriting by transformation. *Information and Computation*, 204(12):1782-1846, 2006.

A new approach to termination analysis of Constraint Handling Rules

Dean Voets, Paolo Pilozzi[⋆], and Danny De Schreye

Department of Computer Science, K.U.Leuven, Belgium
{ Dean.Voets@student, Paolo.Pilozzi@cs, Danny.DeSchreye@cs }.kuleuven.be

1 Introduction

Constraint Handling Rules (CHR) is a concurrent, committed-choice constraint programming language (see [2]). It is a rule-based language, in which multisets of atomic constraints are rewritten using guarded rules. It has a simple syntax and declarative semantics, and is very suitable for implementing constraint solvers.

Although the language is strongly related to Logic Programming (LP) and to lesser extent also to Term-Rewrite Systems (TRS), termination analysis of CHR has received little attention. To the best of our knowledge, the only contribution so far is reported in [3]. This study is limited to CHR programs with only one type of rules: the so called 'simplification rules'. The work shows that, for this class of programs, termination analysis techniques developed for LP (see [1]) can be adapted to CHR.

In this paper, we present a new approach to termination analysis of CHR which is applicable to a much larger class of CHR programs. We propose a new termination condition and show its applicability to CHR programs with rules that are not only of the simplification type. We have successfully tested the condition on a benchmark of programs, using a prototype analyser.

2 CHR: syntax and semantics

A constraint is a predicate in first-order logic. We distinguish between built-in (predefined) constraints and CHR (user-defined) constraints. Built-in constraints are handled by an underlying constraint solver. CHR constraints are defined by a CHR program. A CHR program is a finite set of CHR rules. There are three kinds of rules. *Simplification* rules replace a conjunction of constraints by an equivalent conjunction of (simplified) constraints. *Propagation* rules only generate new (redundant) constraints. These rules take respectively the form

$$H_1, ..., H_n \begin{pmatrix} \Leftrightarrow \\ \Rightarrow \end{pmatrix} G_1, ..., G_k \mid B_1, ..., B_m.$$

where $H_1, ..., H_n$ is a conjunction of CHR constraints , $G_1, ..., G_k$ is a conjunction of (built-in) guards and $B_1, ..., B_m$ is a new conjunction of CHR constraints and built-in constraints. The language also offers *simpagation* rules, which represent a combination of simplification and propagation. We will not discuss this type of rule here any further, since it can be considered a redundant extension, aiming at more compact specifications.

[⋆] Supported by the Fund for Scientific Research - Flanders (Belgium) (F.W.O. - Vlaanderen): "Termination analysis, crossing paradigm borders".

Example 1 (Fibonacci).

$$\begin{array}{l} fib(N,M1), fib(N,M2) \Leftrightarrow M1 = M2 \mid fib(N,M1). \\ fib(0,M) \Rightarrow M = 1. \\ fib(1,M) \Rightarrow M = 1. \\ fib(N,M) \Rightarrow N > 1 \mid N1 \; is \; N-1, N2 \; is \; N-2, \\ \qquad\qquad fib(N1,M1), fib(N2,M2), M \; is \; M1+M2. \end{array}$$

The operational semantics of CHR programs is given by a state transition system, where a state refers to a conjunction of (CHR and built-in) constraints. As mentioned above, we refer to it as the *constraint store.* A *computation* is a sequence of state transitions. A *query* provides the initial state. The computation terminates when an inconsistent state is obtained (failing computation) or when no more computation steps are possible. The selection of rules from the program is done in a fair way. However, it is a committed choice once the guard is satisfied.

Denoting by H, G and B; the conjunctions $H_1, ..., H_n$,$G_1, ..., G_k$ and $B_1, ..., B_m$ respectively, the state transition system $\rightarrow$ is specified as:

Simplify: $H' \wedge D \rightarrow (H' = H) \wedge G \wedge B \wedge D$

Propagate: $H' \wedge D \rightarrow (H' = H) \wedge H' \wedge G \wedge B \wedge D$

where H' is a conjunction of constraints in the constraint store which matches with the head H of some rule in the program, such that this matching, together with the guard G and the built-ins in B, are satisfiable in the underlying constraint solver, given the constraints in the remainder of the store, D. Note that all propagation rules can be activated infinitely often on the same conjunction of constraints, since the constraints that match with the head of the rule are not removed from the store. To avoid this, the operational semantics additionally includes a *propagation history*, which keeps track of which rules were applied on which conjunctions of constraints.

Example 2 (Fibonacci continued). With a typical query like *fib*$(3, N)$, the last propagation rule adds two new *fib/2* constraints with lower first arguments to the store (but does not remove the constraint *fib*$(3, N)$), the first two propagation rules resolve base cases , while the simplification rule removes duplicates. The end state is the constraint store *fib*$(3, 3) \wedge$ *fib*$(2, 2) \wedge$ *fib*$(1, 1) \wedge$ *fib*$(0, 1)$.

3 Termination analysis and CHR

In [3], termination proofs for CHR make use of a *ranking* function that maps constraints to natural numbers. It is the direct counterpart of the *norm* and *level mapping* used in LP (see [1]). One difference, however, is that in LP it suffices to reason about the rank or level values of individual atoms. In CHR, multiple constraints are simultaneously replaced by a number of constraints, or - for propagation rules - give rise to the addition of constraints. Although this implies different termination conditions, [3] shows that concepts and ideas from LP termination analysis are adaptable to CHR *with simplification only.*

The extension to programs with propagation rules gives a totally new termination problem. In LP, TRS and in CHR with simplification rules only, a termination proof is based on a decreasing order associated to consecutive computation states. However, with each activation of a propagation rule in CHR, there is an explicit increase of the constraint store: new constraints are added and no existing constraints are removed. One would need to keep track of information regarding the propagation history *within* the computation states, in order to observe a decrease between consecutive states. Unfortunately, we expect that making the propagation history explicit in states could lead to rather messy and low-level termination conditions.

Instead of a termination argument based on a comparison of sizes of *consecutive computation states*, we formulate and verify conditions, imposed on the dynamic *process of adding constraints* to the store. We formulate conditions which guarantee that in the complete computation process, only a finite number of constraints are added to the store. Due to the use of the propagation history in CHR, this implies termination.

4 The ranking condition

In the following, P denotes a CHR program and I a query. constraints).

Definition 1 (Norm, level mapping). *A norm $\| \,.\, \|$ is a function from terms to $\mathbb{N}$. A level mapping $\mid . \mid$ is a function from CHR-constraints to $\mathbb{N}$.*

Definition 2 (Rigidity). *Let C be CHR constraint and $\mid . \mid$ a level mapping. C is rigid with respect to $\mid . \mid$ if for any substitution $\theta : \mid C\theta \mid = \mid C \mid$.*

Definition 3 (Call set). *Given a program P and a query I, the call set for P and I, $Call(P, I)$, is the set of all CHR constraints which enter the constraint store during a computation of P for I.*

Typically, in an automated analysis, we will use abstract interpretation to compute a safe (upper) approximation of the call set.

Definition 4 (Ranking condition for CHR). *A program P and a query I satisfy the Ranking condition for CHR, w.r.t. a level mapping $\mid . \mid$ if:*
1) All elements of $Call(P, I)$ are rigid with respect to $\mid . \mid$,
2) For each rule of P and for each conjunction of atoms H' in $Call(P, I)$ which matches with the head H, using matching substitution θ, and such that all built-in constraints in $G\theta$ and $B\theta$ can be satisfied with answer substitution θ':
- *For a simplification rule $H_1, ..., H_n \Leftrightarrow G \mid B_1, ..., B_m.$, with body-CHR constraints $B_k, ..., B_m$:*
  Let $m_h = max_{i=1,...,n} \mid H_i\theta \mid$ and $m_b = max_{j=k,...,m} \mid B_j\theta \mid$ then
  - *either $m_h > m_b$, or,*
  - *$m_h = m_b$ and, with q_h, q_b the number of constraints with rank m_h in $H_1\theta, ..., H_n\theta$ and $B_1\theta, ..., B_m\theta$ respectively: $q_h > q_b$*
- *For a propagation rule $H_1, ..., H_n \Rightarrow G \mid B_1, ..., B_m.$, with body-CHR constraints $B_k, ..., B_m$:*
  - *for all $i = 1, ..., n$ and $j = k, ..., m$: $\mid H_i\theta \mid > \mid B_j\theta \mid$.*

Theorem 1 (Sufficiency of the Ranking condition). *Let P be a CHR program and I a query. If there exists a level mapping $\mid . \mid$, such that P and I satisfy the Ranking condition w.r.t. $\mid . \mid$, then all computations for I in P terminate.*

The idea of the proof is that, with the Ranking condition, indeed only finitely many constraints can ever enter the store. With propagation rules, only smaller ranking constraints can enter the store. Due to the propagation history, this process dies out. With simplification rules, either bigger ranking constraints are replaced by smaller ranking ones, or the number of maximally ranked ones is reduced. These effects combined yield only finitely many constraints.

Example 3 (Fibonacci continued). Let $\mathit{fib}(n, M)$ be a query, with n a natural number and M a free variable. One can infer that $\{\mathit{fib}(n, m) \mid n, m \in \mathbb{N}\} \cup \{\mathit{fib}(n, M) \mid n \in \mathbb{N}\ \mathit{and}\ M\ a\ free\ variable\}$ is te call set. As a norm, we map each natural number to itself. The level mapping is defined on the call set as $\mid fib(n, X) \mid = \parallel n \parallel = n$. Clearly the call set is rigid w.r.t. $\mid . \mid$. For the first rule we have: $\mid fib(n, M1) \mid = \mid fib(n, M2) \mid = \mid fib(n, M1) \mid = n$. So, $m_h = m_b$ and $q_h > q_b$. The conditions for rules 2 and 3 are trivially satisfied, since they have no CHR constraints in their bodies. Finally, for rule 4, $\mid fib(n, M) \mid = n > \mid fib(n1, M1) \mid = n - 1$ and $\mid fib(n, M) \mid = n > \mid fib(n2, M2) \mid = n - 2$. Thus, the program terminates for these queries.

5 Experimental evaluation

We have implemented a prototype system using the proposed Ranking condition and tested it on a benchmark of CHR programs. The benchmark consists of a set of CHR programs based on LP termination problems, a set of CHR programs taken from [3], marked with a $'*'$, and some others taken from WebCHR (http://chr.informatik.uni-ulm.de/∼webchr/). The symbol $'+'$ denotes that the system proved termination, $'-'$ that it failed to do so. All programs terminate.

CHR, simplification only								General CHR	
ackermann	-	convert	+	linpoleq*	-	pathcons*	-	arccons*	-
average	-	diff	+	max	+	power	+	bool	+
binlog	+	factorial	+	mean	+	revlist	+	fibonacci	+
booland*	+	gcd	+	mergesort	+	som	+	integrity	+
boolcard*	+	genint	+	modulo	+	toyama	-	primes1	+
concat	-	joinlists	+	oddeven	+	weight	-	primes2	+

The success rate of the system is quite good, both for simplification only and for the general case. Note that in theory, the technique in [3] is more powerful on the simplification only cases. In order to be able to deal with propagation, our conditions on the simplification rules are stronger than those in [3]. However, it should be noted that [3] is a not available as an implemented system and that an implementation might actually not be able to support all examples of the theory (for instance if static analysis is used to infer some required information).

References

1. D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, 19-20:199–260, 1994.
2. T. Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
3. T. Frühwirth. Proving termination of constraint solver programs. In Krzysztof R. Apt, Antonis C. Kakas, Eric Monfroy, and Francesca Rossi, editors, *New Trends in Constraints*, volume 1865 of *Lecture Notes in Computer Science*, pages 298–317, Paphos, Cyprus, 2000. Springer-Verlag.

Proving Termination of CHR in Prolog: A Transformational Approach

Paolo Pilozzi[⋆], Tom Schrijvers[⋆⋆], and Danny De Schreye

Department of Computer Science, K.U.Leuven, Belgium
{Paolo.Pilozzi, Tom.Schrijvers, Danny.DeSchreye }@cs.kuleuven.be

1 Introduction

Constraint Handling Rules (CHR) is a concurrent, comitted-choice, logic programming language. It is constraint-based and has guarded rules that rewrite multisets of atomic formulas [1]. Its simple syntax and semantics make it well-suited for implementing custom constraint solvers. Despite the amount of work directed to CHR, not much has been done on termination analysis. To the best of our knowledge, there were no attempts made to automate termination proofs.

The first and, until now, only contribution to termination analysis of CHR is reported in [2] and shows termination of CHR programs under the theoretical semantics [1] of CHR. Termination is shown, using a *ranking function*, mapping sets of constraints to a well-founded order. A *ranking condition*, on the level of the rules, implies termination. Although termination conditions in CHR take a different form than in Logic Programs (LP) and Term-Rewrite Systems (TRS), [2] shows that achievements from the work on termination of LP and TRS are relevant and adaptable to the CHR context.

In this paper, we present a termination preserving transformation of CHR to Prolog. This allows the direct reuse of termination proof methods from LP and TRS for CHR, yielding the first fully automatic termination proving for CHR. We implemented the transformation and used existing termination tools for LP and TRS on a set of CHR programs to demonstrate the usefulness of our approach. In [3], we formalize the transformation and prove soundness w.r.t. termination.

2 Syntax and Semantics of CHR

We briefly introduce the necessary aspects of CHR. For a more elaborate introduction see [1].

[⋆] Supported by the Fund for Scientific Research - Flanders (Belgium) (F.W.O. - Vlaanderen): ”Termination analysis, crossing paradigm borders”.

[⋆⋆] Post-Doctoral Researcher of the Fund for Scientific Research - Flanders (Belgium) (F.W.O. - Vlaanderen).

Syntax. A *constraint* in CHR is a first-order predicate. We distinguish between *built-in constraints*, pre-defined and solved by the underlying constraint solver, and *CHR constraints*, user-defined and solved by a finite set of *CHR rules* (CHR program). Rules are of the form:

$$H_1 \ \backslash \ H_2 \ \Leftrightarrow \ G \ \mid \ B.$$

Both parts of the multihead, H_1 and H_2, are conjunctions of CHR constraints. The body, B, is a conjunction of built-in and CHR constraints. The optional guard, G, is a conjunction of built-in constraints. Empty conjuncts are denoted by the built-in constraint *true*.

Semantics. A *CHR program* defines a state transition system, where the *state* is defined as a conjunction of CHR and built-in constraints, called the *constraint store*. The *initial state* or *query* is an arbitrary conjunction of constraints. In a *final state* or *answer*, either the built-in constraints are inconsistent (*failed state*), or no more transitions are possible. The *transition relation*, $\longmapsto$, between states, given a constraint theory CT (built-ins) and a CHR program P, is defined as:

Transition Relation
$H_1' \wedge H_2' \wedge D \longmapsto (H_1 \wedge H_2 = H_1' \wedge H_2') \wedge G \wedge B \wedge H_1' \wedge D$
$if\ (H_1 \setminus H_2 \Leftrightarrow G \mid B)\ in\ P\ and\ CT \models D \rightarrow \exists \bar{x}((H_1 \wedge H_2 = H_1' \wedge H_2') \wedge G)$

A rule is applicable to CHR constraints, H', whenever these match (are instance of) the head atoms H of the rule, such that the guard G is satisfied, given the built-ins in the store. The matching is the effect of the existential quantification in $\exists \bar{x}(H = H')$, where $\bar{x}$ denotes the variables in the rule chosen from P. Rule application is non-deterministic and committed-choice.

Example. The program below implements merge-sort[1]. The query $mergesort(L)$ with L a list of length 2^n, yields a tree-representation of the order.

$true \setminus mergesort([]) \Leftrightarrow true.$
$true \setminus mergesort([L|List]) \Leftrightarrow root(0, L),\ mergesort(List).$
$true \setminus root(D, L1),\ root(D, L2) \Leftrightarrow less(L1, L2) \mid root(s(D), L1),\ edge(L1, L2).$

The first two rules decompose a list of elements, while adding new *root/2* constraints to the store. The constraint $root(D, L)$ represents a tree of depth D (initially 0) and root value L. The third rule performs the actual merge-sorting by joining two trees of equal depth. It replaces both trees by a new tree of increased depth, where the largest root becomes a child of the smallest.

3 A Transformational Approach to Termination Analysis

This section presents a termination preserving CHR to Prolog transformation.

[1] Based on an example from `http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/`

Transformation. The CHR constraint store is represented in Prolog as a list of constraints. This list is permuted before inspection to behave as a multiset. Each CHR rule is transformed into a *rule/2* Prolog clause. The *rule/2* predicate defines the transition relation between constraint lists.

A call to *rule/2* with a list representation of the store, unifies the permutation of the list with a list representation of the rule's head and a remainder R. If a unification exists such that all body atoms of the clause succeed, then the second term of *rule/2* binds to the resulting store, by adding to R the left heads and the introduced CHR constraints. I.e. a CHR rule of the form:

$$H_1, ..., H_i \setminus H_j, ..., H_n \Leftrightarrow G_1, ..., G_k \mid B_{BI_1}, ..., B_{BI_l}, B_{CHR_1}, ..., B_{CHR_m}.$$

becomes a Prolog clause:

$$\begin{aligned} &rule([H_1, ..., H_n|R], [H_1, ..., H_i, B_{CHR_1}, ..., B_{CHR_m}|R]) \texttt{ :- } \\ &\qquad\qquad G_1, ..., G_k, B_{BI_1}, ..., B_{BI_l}. \end{aligned}$$

Repeated rule application is captured by[2]:

$$\begin{aligned} &goal(S) \texttt{ :- } \\ &\quad permutation(S, PS), \\ &\quad rule(PS, NS), \\ &\quad goal(NS). \\ &goal(\_). \end{aligned}$$

Termination. Termination proofs of the transformed program take a general form, showing a decrease in level mapping: $|goal(S)| > |goal(NS)|$. This condition boils down to a condition on the norms of the arguments of all *rule/2* clauses: $\| PS \|_{sl} > \| NS \|_{sl}$. Here $\| . \|_{sl}$ denotes the norm[3] used on storelists.

Example. We revisit *mergesort* and transform each of the rules into their clausal form. We assume that *less/2* is predefined.

$$\begin{aligned} &rule([mergesort([])|R], R). \\ &rule([mergesort([L|List])|R], [root(0, L), mergesort(List)|R]). \\ &rule([root(D, L1), root(D, L2)|R], [root(s(D), L1), edge(L1, L2)|R]) \texttt{ :- } less(L1, L2). \end{aligned}$$

The following conditions imply termination of *mergesort*:

$$\begin{aligned} \| [mergesort([])|R] \|_{sl} &> \| R \|_{sl} \\ \| [mergesort([L|List])|R] \|_{sl} &> \| [root(0, L), mergesort(List)|R] \|_{sl} \\ \| [root(D, L1), root(D, L2)|R] \|_{sl} &> \| [root(s(D), L1), edge(L1, L2)|R] \|_{sl} \end{aligned}$$

These conditions are met by the following storelist norm, $\| \cdot \|_{sl}$:
$\| [E_1, ..., E_n] \|_{sl} = \sum_1^n \| E_i \|_c$. Here, $\| . \|_c$ are constraint norms:

$$\begin{aligned} \| mergesort(List) \|_c &= 1 + 2 \| List \|_l \\ \| root(D, L) \|_c &= 1 \\ \| edge(A, B) \|_c &= 0 \end{aligned}$$

and $\| \cdot \|_l$ is the usual list-length norm.

[2] We have factored out *permutation/2* to *goal/1*, as it is present in every rule clause.
[3] The norm should be invariant over permutation.

4 Evaluation

We have implemented the transformation[4] in K.U.Leuven CHR for SWI-Prolog. The implementation was used to transform two sets of CHR programs, one based on LP[5] benchmarks and another from [2] marked with a '*'. Polytool (PT) and AProVE (AP), based on respectively LP and TRS termination proof techniques, were used to prove termination of the transformed CHR programs:

	PT	AP		PT	AP		PT	AP		PT	AP
ackermann	-	-	convert	-	-	linpoleq*	-	+	pathcons*	-	+
average	+	+	diff	+	+	max	+	+	power	+	+
binlog	+	+	factorial	+	+	mean	+	+	revlist	+	+
booland*	+	+	gcd	+	+	mergesort	+	+	som	+	+
boolcard*	-	+	genint	+	+	modulo	+	+	toyama	+	+
concat	-	-	joinlists	+	+	oddeven	+	+	weight	-	-

Similar, but weaker results were obtained with the LP termination tools cTI and TerminWeb. The results show that the transformation conserves the termination argument, that analysis can be done using approaches from LP and TRS and that it can be automated in terms of existing termination tools.

5 Conclusion

We have introduced a straightforward program transformation that allows for termination proofs of CHR programs, by using existing tools for LP and TRS. We have implemented the transformation and obtained good results, which confirm the applicability of proof techniques of LP and TRS to CHR. Termination proofs take a general form as was suggested by [2] and can be automated using existing termination analysers for LP and TRS.

Future work will address CHR programs with rules that do not remove constraints from the store. These rules introduce trivial non-termination, prevented by a propagation history, which is not supported by our current transformation.

References

1. T. Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
2. T. Frühwirth. Proving termination of constraint solver programs. In Krzysztof R. Apt, Antonis C. Kakas, Eric Monfroy, and Francesca Rossi, editors, *New Trends in Constraints*, volume 1865 of *Lecture Notes in Computer Science*, pages 298–317, Paphos, Cyprus, 2000. Springer-Verlag.
3. P. Pilozzi, T. Schrijvers, and D. De Schreye. Proving termination of CHR in Prolog: A transformational approach. Report CW 487, K.U.Leuven, Department of Computer Science, April 2007.

[4] Available at `http://www.cs.kuleuven.be/~paolo/`

[5] `http://www.lri.fr/~marche/wst2006-competition/tpdb.html`

Ranking Functions for Size-Change Termination II
(extended abstract)

Amir M. Ben-Amram and Chin Soon Lee[⋆]

Abstract. The Size-Change Termination technique is based on a program abstraction for which termination is decidable. Termination is verified by a set of local termination proofs that account for *all cycles* in a control-flow graph. We present algorithms that construct a *global ranking function* for an SCT instance. Such functions serve as easy-to-check witnesses for termination, and are therefore interesting for purposes of program certification. Their particular form and complexity shed light on the theory of SCT termination proofs.
Our constructions are simpler and more transparent than previously known. They improve the upper bound on the size of the ranking expression from triply exponential to singly exponential. Another contribution is a set of lower-bound results, proving that our constructions are optimal in a certain sense. An interesting point that arises from our constructions is the usefulness of *multisets of data* in ranking expression construction.

1 SCT and Ranking Functions in a Nutshell

Let *Val* be a well-ordered set of data values. A control-flow graph (CFG) is a directed multigraph (F, C). The nodes are called flow-chart points or just flow-points. The set of arcs from $f \in F$ to $g \in F$ is C_{fg}. One of the nodes, f_0, is *initial* or starting point. All nodes are reachable from f_0. For each $f \in F$, we have a distinct set of *parameters* $\mathit{Par}(f)$, representing data pertinent to describing the program state at this point. For simplicity, all such sets have the same size n. Formally, the set of (abstract) program states is

$$St = \{(f, \sigma) \mid f \in F, \sigma : \mathit{Par}(f) \to \mathit{Val}\}\,.$$

For $f, g \in F$, a size-change graph (SCG) with source f and target g is a bipartite directed graph with source nodes corresponding to $\mathit{Par}(f)$ and target nodes to $\mathit{Par}(g)$. We write this fact as $G : f \to g$. Arcs of G represent constraints on transitions $(f, \sigma) \to (g, \sigma')$. In the ordinary SCT formulation, there are just two types of arcs: a *strict* arc $x \overset{\downarrow}{\to} y$ represents strict descent, i.e., $\sigma(x) > \sigma'(y)$. A *non-strict* arc $x \to y$ represents the constraint $\sigma(x) \geq \sigma'(y)$. We write $G \models (f, \sigma) \mapsto (g, \sigma')$ if all constraints are satisfied. An SCT instance, or abstract program, also known as *annotated control-flow graph* (ACG), is a CFG where every arc $c \in C_{fg}$ is annotated with a SCG $G_c : f \to g$.

Let $\mathcal{G}$ be an SCT instance (formally we view $\mathcal{G}$ as just the set of SCG's, implicitly specifying the CFG). A $\mathcal{G}$-*multipath* is a (finite or infinite) sequence $M = G_1 G_2 \ldots$ of elements of $\mathcal{G}$ that label a corresponding directed path in the CFG, often denoted

[⋆] `benamram.amir@gmail.com, cslee_sg@hotmail.com` by *cs* (for computation sequence, or call sequence—for functional programmers). We also view a multipath as the (finite or infinite) *layered directed graph* obtained by identifying the target nodes of G_i with the source nodes of G_{i+1}. A *thread* in M is a (finite or infinite) directed path in this graph. A thread is *descending* if it includes a strict arc; it is *infinitely descending* if it includes infinitely many strict arcs.

$\mathcal{G}$ is said to *satisfy SCT* (or "terminate") if every infinite multipath contains an infinitely-descending thread. This is a sufficient condition for termination of any program modelled by $\mathcal{G}$ (in fact, the most precise condition). In the rest of this paper, we only consider terminating instances.

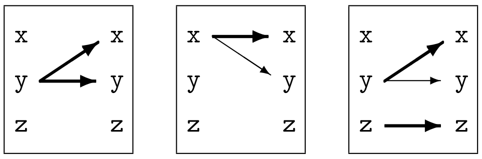
Let $P(s, s')$ be any predicate defined over pairs of states. We write $G \models P(s, s')$ if $G \models s \mapsto s' \Rightarrow P(s, s')$. A (global) ranking function for $\mathcal{G}$ is a function $\rho : St \to W$, where W is a well-ordered set, such that $G \models \rho(s) > \rho(s')$ for every $G \in \mathcal{G}$[1]. It is often convenient to write $\rho(f, [x_1 \to v_1, \ldots, x_n \to v_n])$ as $\rho_f(v_1, \ldots, v_n)$.

Constructing a ranking function for an SCT instance is sometimes a way to understand the type of "behaviour" that the instance expresses. Examples found in previous publications on SCT include programs where the maximum of parameters decreases (consider a standard recursive gcd function), programs where the minimum decreases and programs with a lexicographic descent in a tuple of parameters. It has recently been shown [4] that a ranking function can be constructed for any given SCT instance. It has the following form:

$$\rho(s) = \min(\max S_1, \max S_2, \ldots)$$

where $\max S_i$ represents the maximum element among a set S_i of *vectors* of parameter values and constants, where vectors are lexicographically ordered. We refer to the above form of expressions as $\min[\max[V]]$ where V refers to the type of vectors.

Here is an example: Consider an SCT instance consisting of the graphs $G_1, G_2, G_3 : f_0 \to f_0$ drawn below; the heavy arcs are strict.



A ranking function for $\mathcal{G}$ is $\rho_{f_0}(x, y, z) = \max\{\langle y, 0, z\rangle, \langle x, 1, z\rangle\}$. To verify this, check each graph in turn, considering each possibility among $y > x$, $y = x$ and $y < x$ for the state before and after the transition (plus the constraints expressed in the graph). In this example the *min* operator is unnecessary, so the expression has type $\max[V]$, with $V = Par(f_0) \times \{0, 1\} \times Par(f_0)$.

The above graph set is also an example of a *fan-in free* SCT instance (no two arcs enter the same node). Fan-in free graphs have been identified as an interesting class in previous work [5, 2, 1]. We will also consider the class of fan-out free instances—mostly because they are similar to fan-in free ones but easier to work with.

[1] We may omit explicit references to s, s' for convenience, and use variable names for their values, e.g., writing $G \models x > x'$ instead of the more precise $G \models \sigma_s(x) > \sigma_{s'}(x)$.

2 Statement of Results

Unfortunately, the size limit on this abstract precludes a presentation of our construction techniques. In this section, we state the results with a bit of commentary. The proofs will be published in the full paper.

Definition 2.1 (Tree Expressions). *For a class* `E` *of expressions, a tree expression over* `E` *is either an expression* `e` $\in$ `E` *or a conditional* `if x<y then e1 else e2`*, where* `x`*,* `y` *are parameter names and* `e1`, `e2` *are tree expressions over* `E`*.*

Definition 2.2 (Simple Multiset Ordering, SMO). *Let A, B be finite multisets over Val. We write $A > B$ if $|A| > |B|$ (the cardinality of A is larger) or if $|A| = |B|$ and the sets can be listed as $A = \{a_1, a_2, \ldots\}$ and $B = \{b_1, b_2, \ldots\}$ with $a_i \geq b_i$ for all i and $a_i > b_i$ for at least one i. We write $A \geq B$ for the non-strict variant.*

Let $k > 0$. When $|A| = |B| = k$, $A > B$ means that a sorted listing of A is lexicographically greater than a sorted listing of B. This is true for both descending sort and ascending sort; which gives two ways of completing it to a total order over k-element multisets. The first gives the multiset order of [3]; the second, the dual multiset order of [2]. Both are useful in our work. In fact, we need a total order in order to have well-defined min and max operators. We define the min operator to use the dual order, while the max operator is defined by the Dershowitz-Manna order.

Definition 2.3 (vectors). *For $f \in F$ and $B > 0$, V_f^B is the set of vectors $\mathbf{v} = \langle v_1, v_2, \ldots \rangle$, of even length, where for every odd i, v_i is a non-empty subset of Par(f), such that all odd positions constitute a partition of Par(f); while for even i, v_i is an integer between 0 and B. If $B = \omega$, i is any nonnegative integer.*

The *value* of $\mathbf{v} \in V_f$ in a given program state is obtained by substituting values for parameters. The odd positions thus become multisets over *Val*. Such vectors are ordered lexicographically, where multisets are ordered by some extension of SMO, and numbers by the natural order. We use the convention that the value is meant whenever $\mathbf{v}$ is referred to in a context that requires a value, e.g., when making statements about order; the state is supposed to be understood from context.

Theorem 2.4. *Let $\mathcal{G}$ be a terminating SCT instance, with m flow-points and n parameters per point.*

1. *Let $B = (1+m)(n!)(n^2)^{2^n}$. There is a ranking function for $\mathcal{G}$ of the form $\rho_f(\sigma) = \min_{S \in \mathcal{S}_f} \max S$, where $\mathcal{S}_f \subseteq \wp(V_f^B)$, $|S| \leq n!$ for all $S \in \mathcal{S}_f$, and $|\mathcal{S}_f| \leq (B^n n!)^{n!}$. There is also a ranking function where $\rho_f(\sigma)$ is a tree expression over V_f^B at the leaves; the size of the expression is $O(n^n)$.*
2. *If $\mathcal{G}$ is fan-out free, let $B = m \cdot 2^n$. There is a ranking function for $\mathcal{G}$ of the form $\rho_f(\sigma) = \min_{\mathbf{v} \in S_f} \mathbf{v}$ where $S_f \subseteq V_f^B$ and $|S_f| \leq n!$. For fan-in free graphs we have the same result with* max *instead of* min.

All the results are given by explicit constructions. In all of them, it is possible to restrict the set components of the vectors to singletons, thus avoiding the use of multiset orders. However, the constructions make use of multisets; we also observe that the use of multisets may help in getting a smaller expression for ρ_f. For a tiny example, consider the size-change graph $\{x \overset{\downarrow}{\to} y, y \to x\}$; we have the set-valued ranking function $\rho(x, y) = \{x, y\}$. Without multisets, we need a bigger expression.

The size of our ranking functions is a vast improvement over the triply-exponential upper bound of [4]. Is it optimal? Already for the fan-out free case, there is a complexity-theoretic argument against the existence of a polynomially-computable family of ranking functions. More interestingly perhaps, we have explicit constructions that provide tight lower bounds on the size of ranking expressions from a class that generalizes the expressions described in Theorem 2.4.

Definition 2.5. *For a state $s = (f, \sigma)$, let Order(s) be the ordering of the parameter values in s (represented, for example, as a graph on Par(f)).*

Definition 2.6. *A VSO function (for Vectors Selected by Order) is a function $\rho_f(s)$ that can be described by assigning to any order τ on Par(f) a vector $\rho^*_f(\tau) \in V^\omega_f$, such that $\rho_f(s)$ is given by evaluating ρ^*_f(Order(s)).*

If vectors contain entries that are sets of parameters, one has to specify in which sense they are meant to descend. Our first result refers to SMO, which is the way our ranking functions (Theorem 2.4) work.

Theorem 2.7. *There is a fan-out free, terminating SCT instance $\mathcal{H}$ with a single flow-point f, $n + 1$ parameters and $2n - 1$ size-change graphs, such that any VSO ranking function ρ_f for $\mathcal{H}$ (based on SMO) must use at least $n!$ different vectors.*

The following result concerns the (more flexible) Dual Multiset Order [2].

Theorem 2.8. *There is a fan-out free, terminating SCT instance $\mathcal{K}$ with a single flow-point f, $2n + 1$ parameters and $n + 1$ size-change graphs, such that any VSO ranking function ρ_f for $\mathcal{K}$ (based on DMO) must use at least 2^n different vectors.*

Note that the lower bound dropped from $2^{\Theta(n \log n)}$ to 2^n. It is an intriguing open problem to find out whether such use of multiset ordering can actually improve the upper bound.

References

[1] Amir M. Ben-Amram. Size-change termination with difference constraints. *ACM Transactions on Programming Languages and Systems*, 2007. To appear.

[2] Amir M. Ben-Amram and Chin Soon Lee. Size-change analysis in polynomial time. *ACM Transactions on Programming Languages and Systems*, 29(1), 2007.

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

[4] Chin Soon Lee. ranking functions for size-change termination. Submitted.

[5] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. 28th ACM Symposium on Principles of Programming Languages, pages 81–92. 2001.

Termination Analysis of Java Bytecode[⋆]

E. Albert[1], P. Arenas[1], M. Codish[2],
S. Genaim[3], G. Puebla[3], and D. Zanardini[3]

[1] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[2] Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel
[3] CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

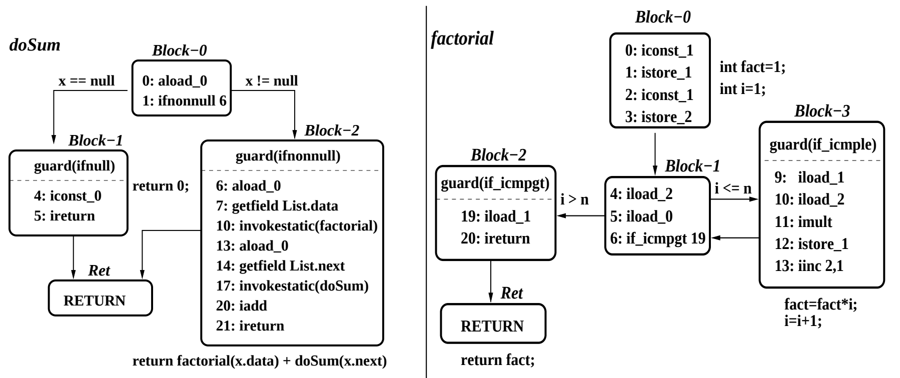
Introduction: The state of the art in termination analysis includes advanced techniques developed for logic and functional programming [12, 4, 9, 11, 10] and imperative languages [2, 5, 8, 6, 10], as well as for term rewriting systems [10]. In [6, 5] tools for proving termination of large industrial code are presented. However, termination of low-level languages, such as Java bytecode, has received little attention. In some situations, such as with *mobile code*, the user only has access to compiled code which may be obtained from an untrusted party. Termination analysis may help avoid, for example, *denial of service* attacks. Such analysis must be applied directly on the low-level code. Java bytecode [13] is widely used in this context due to its security features and platform-independence.

Termination analyses are typically based on *ranking functions*, which map program states to the elements of a well-founded domain. Termination is guaranteed if a ranking function which decreases during computation is found, in particular as the program goes through its loops. Termination analysis of Java bytecode presents some peculiar features which stem from its low-level and object-oriented nature: (1) loops originate from different sources (conditional and unconditional jumps, method calls, or even exceptions); (2) *size measures* must consider supported data types (primitive types, objects, and arrays); and (3) data can be stored in local variables, operand stack elements or heap locations.

In this work, we describe a termination analysis for Java bytecode, based on a translation into a structured *intermediate representation* similar to that of [1] where it is applied in the context of cost analysis. This representation captures in a uniform setting all loops (regardless of whether they originate from recursive calls or iterative loops) and all variables (either local variables or operand stack elements). Given this representation we consider a general form of the *size-change graphs* principle [11, 4] to prove termination of the intermediate program, which in turn implies the termination of the original program.

Termination of Java Bytecode by Example: We illustrate our approach by means of a running example which consists of two Java methods, doSum and

[⋆] This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project. S. Genaim was supported by a *Juan de la Cierva* Fellowship awarded by MEC.



```
static int doSum(List x) {
  if (x == null) return 0;
  else
    return factorial(x.data)+
                       doSum(x.next);
}
```

```
static int factorial(int n) {
  int fact = 1;
  for (int i = 1; i <= n; i++)
    fact = fact * i;
  return fact;
}
```

Fig. 1. Control flow graphs for the running example, with possible Java source code

factorial. Fig. 1 depicts their *control flow graphs* (CFGs for short), explained in more detail below, and provides a possible Java source for each method. The process is divided into five steps. The first two are required to obtain a structured representation of the program. The other three are required for proving termination.

Step I: Control Flow Graph. To facilitate standard termination proving techniques, we first transform the unstructured control flow of bytecode into a structured intermediate representation. A control flow graph consists of *guarded* basic blocks and edges which describe how control flows between blocks. Basic blocks are sequences of non-branching bytecode instructions, and edges are obtained from instructions which might branch such as virtual method invocation, conditional jumps, exceptions, etc. Observe in the CFG for doSum in Fig. 1 (left), the branching at Block-0 which distinguishes the base case (Block-1) and the recursive step (Block-2) for the recursive definition (in the Java code). The successive blocks at a branch have mutually exclusive *guards* since only one of them can be executed. Guards take the form guard(cond), where cond is a Boolean condition on the local variables and stack elements. For example, guard(ifnull) succeeds if the top stack element s_0 is a null reference. The branching at Block-1 in the CFG of factorial corresponds to the condition of the for loop in which the guard guard(if_icmpgt) on block Block-2 succeeds if the top two stack elements s_0 and s_1 satisfy $s_0 > s_1$. The special block Ret indicates a normal exit of the method. Note also that (static) method invocations –both the recursive call to doSum and the non-recursive call to factorial– appear within invokestatic bytecode instructions.

Step II: Intermediate Representation. In the next step, the CFG is represented in a procedural way by means of an *intermediate representation.* This representation consists of a set of *guarded rules* which are obtained from the blocks in the CFG. A principal advantage is that all possible forms of loops in the program are represented now in a uniform way. We can observe this, for instance, in the representation of the for loop by means of rule $factorial_1$ and the representation of the recursive procedure doSum in the rule *doSum.* Although method factorial is not recursive, its associated representation is, as $factorial_1$ contains a loop.

$$
\begin{aligned}
&\mathit{factorial}(n, ret) \leftarrow \mathit{factorial}_0(n, \mathit{fact}, i, ret).\\
&\mathit{factorial}_0(n, \mathit{fact}, i, ret) \leftarrow \texttt{iconst}(1, s_0), \texttt{istore}(s_0, \mathit{fact}), \texttt{iconst}(1, s_0),\\
&\qquad \texttt{istore}(s_0, i), \mathit{factorial}_1(n, \mathit{fact}, i, ret).\\[6pt]
&\mathit{factorial}_1(n, \mathit{fact}, i, ret) \leftarrow \texttt{iload}(i, s_0), \texttt{iload}(n, s_1),\\
&\qquad (\mathit{factorial}_2(n, \mathit{fact}, i, s_0, s_1, ret);\\
&\qquad \ \mathit{factorial}_3(n, \mathit{fact}, i, s_0, s_1, ret)).\\[6pt]
&\mathit{factorial}_2(n, \mathit{fact}, i, s_0, s_1, ret) \leftarrow \texttt{guard}(\texttt{if\_icmpgt}(s_0, s_1)), \texttt{iload}(\mathit{fact}, s_0),\\
&\qquad \texttt{ireturn}(s_0, ret).\\[6pt]
&\mathit{factorial}_3(n, \mathit{fact}, i, s_0, s_1, ret) \leftarrow \texttt{guard}(\texttt{if\_icmple}(s_0, s_1)),\\
&\qquad \texttt{iload}(\mathit{fact}, s_0), \texttt{iload}(i, s_1), \texttt{imul}(s_0, s_1, s_0),\\
&\qquad \texttt{istore}(s_0, \mathit{fact}), \texttt{iinc}(i, 1), \mathit{factorial}_1(n, \mathit{fact}, i, ret).
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{doSum}(x, ret) \leftarrow \mathit{doSum}_0(x, ret).\\
&\mathit{doSum}_0(x, ret) \leftarrow \texttt{aload}(x, s_0), (\mathit{doSum}_1(x, s_0, ret); \mathit{doSum}_2(x, s_0, ret)).\\
&\mathit{doSum}_1(x, s_0, ret) \leftarrow \texttt{guard}(\texttt{ifnull}(s_0)), \texttt{iconst}(0, s_0), \texttt{ireturn}(s_0, ret).\\
&\mathit{doSum}_2(x, s_0, ret) \leftarrow \texttt{guard}(\texttt{ifnonnull}(s_0)), \texttt{aload}(x, s_0), \texttt{getfield}(\mathit{List.data}, s_0, s_0),\\
&\qquad \mathit{factorial}(s_0, s_0), \texttt{aload}(x, s_1), \texttt{getfield}(\mathit{List.next}, s_1, s_1),\\
&\qquad \mathit{doSum}(s_1, s_1), \texttt{iadd}(s_1, s_0, s_0), \texttt{ireturn}(s_0, ret).
\end{aligned}
$$

A relevant feature of our representation is that the arguments of each rule correspond to: (1) the method's local variables (n, fact, i for factorial, and x for doSum in our example); (2) a single variable which corresponds to the method's return value (ret); and (3) the *active stack* elements s_i at the block's entry and exit, i.e., the stack elements are considered as local variables. The guards and the bytecodes which appear in the rules are written in a different font in the above example to distinguish them from calls to blocks. They are obtained from the block's guard and bytecode instructions, by adding the local variables and stack elements on which they operate as explicit arguments.

Step III: Inferring Size Relations and Binary Clauses. Next, a global size analysis is performed on the recursive representation to infer *calls-to size-relations* between the variables in the head of the rule and the variables used in the calls (to rules) occurring in the body. In our example, assuming that the input to doSum is an acyclic list, we obtain the following *calls-to* size-relations:

$$\begin{array}{ll}
\langle \mathit{factorial}(n) & \mapsto \mathit{factorial}_0(n', \mathit{fact}, i), \{n'{=}n\}\rangle \\
\langle \mathit{factorial}_0(n, \mathit{fact}, i) & \mapsto \mathit{factorial}_1(n', \mathit{fact}', i'), \{n'{=}n, \mathit{fact}'{=}1, i'{=}1\}\rangle \\
\langle \mathit{factorial}_1(n, \mathit{fact}, i) & \mapsto \mathit{factorial}_2(n', \mathit{fact}', i', s_0, s_1), \{s_0{=}i, s_1{=}n, s_0{\leq}s_1\} \cup \varphi_{id}\rangle \\
\langle \mathit{factorial}_1(n, \mathit{fact}, i) & \mapsto \mathit{factorial}_3(n', \mathit{fact}', i', s_0, s_1), \{s_0{=}i, s_1{=}n, s_0{>}s_1\} \cup \varphi_{id}\rangle \\
\langle \mathit{factorial}_3(n, \mathit{fact}, i, s_0, s_1) & \mapsto \mathit{factorial}_1(n', \mathit{fact}', i'), \{n'{=}n, i'{=}i{+}1\}\rangle \\
\langle \mathit{doSum}(x) & \mapsto \mathit{doSum}_0(x'), \{x'{=}x\}\rangle \\
\langle \mathit{doSum}_0(x) & \mapsto \mathit{doSum}_1(x', s_0), \{x'{=}x, s_0{=}x, s_0{=}0\}\rangle \\
\langle \mathit{doSum}_0(x) & \mapsto \mathit{doSum}_2(x', s_0), \{x'{=}x, s_0{=}x, s_0{>}0\}\rangle \\
\langle \mathit{doSum}_2(x, s_0) & \mapsto \mathit{factorial}(\_, \_), \{\}\rangle \\
\langle \mathit{doSum}_2(x, s_0) & \mapsto \mathit{doSum}(x'), \{x'{=}x{-}1\}\rangle
\end{array}$$

where $\varphi_{id} \equiv \{n'{=}n, \mathit{fact}'{=}\mathit{fact}, i'{=}i\}$. Size relations provide information about the changes in the size of the data structure (or in the value of integer variables) when control goes from one part of the program (e.g., a block or a method) to another one. For example, rule $\mathit{factorial}_3$ shows that the value of the loop counter i increases by 1. The analysis is done by: (1) abstracting bytecode instructions into the linear constraints they impose on their arguments size, where the size of an integer variable is its value [7], the size of a reference variable is the length of the maximal-path [14] reachable from that variable and the size of an array is its length; for instance, the abstraction of the bytecode instruction `iinc(i,1)` results in `i=i+1`; and (2) computing a fixpoint which tracks the *calls-to* relations. Note that the path-length analysis [14] abstracts cyclic data structures to top, therefore our analysis cannot infer termination when it depends on cyclic data structures.

Step IV: Transitive Closure of Binary clauses. The *calls-to* relations are exactly the *binary clauses* that represents the *direct calls* between the different blocks of the program. Starting from this set, we iteratively compute the transitive closure of their composition in order to obtain *binary clauses* for the indirect calls, in particular those that correspond to loops. Informally, composing two binary clauses $\langle p(\bar{x}) \mapsto q(\bar{y}), \varphi_1\rangle$ and $\langle q(\bar{y}) \mapsto h(\bar{z}), \varphi_2\rangle$ results in a new binary clause $\langle p(\bar{x}) \mapsto h(\bar{z}), \exists \bar{y}.\varphi_1 \wedge \varphi_2\rangle$. The next recursive binary clauses are, among unnecessary others, obtained by the transitive closure of the above *binary clauses*:

$$\begin{array}{ll}
\langle \mathit{factorial}_1(n, \mathit{fact}, i) & \mapsto \mathit{factorial}_1(n', \mathit{fact}', i'), \{n'{=}n, i'{>}i\}\rangle \\
\langle \mathit{doSum}(x) & \mapsto \mathit{doSum}(x'), \{x'{<}x, x{>}0\}\rangle
\end{array}$$

The notion of *binary clauses* of [4] is a general form of *size-change graphs* [11] in the sense that it allows the use of arbitrary constraints domains to describe the corresponding program states. This is useful in practice when different loops might diverge in different directions and when the decreasing measure is a non-trivial combination of some of the loop arguments. Similar notions to that of *binary clauses* have been introduced also in [12, 9, 16, 3].

Step V: Ranking Functions for Loops. Recursive *binary clauses* in the transitive closure represent all possible loops in the program, and in order to prove that the corresponding program terminates, it is sufficient to check that for each such recursive *binary clause* $\langle p(\bar{x}) \mapsto p(\bar{y}), \varphi\rangle$ there exists a function f over a well-founded domain such that $\varphi \models f(p(\bar{x})) > f(p(\bar{y}))$ [3]. In the last step we check the existence of a ranking function [15] for each recursive *binary clause* in the transitive closure. The functions $f(\mathit{factorial}_1(n, \mathit{fact}, i)){=}n{-}i$ and

$f(doSum(x))=x$ are ranking functions which guarantee that the corresponding loop will be traversed finitely, therefore the program terminates.

Summary: We have a preliminary implementation which is based on the size analysis of [1] and the *binary clauses* component of the TerminWeb analyzer [17]. Ongoing work is to formally justify the analysis following the approach of [4], namely by deriving a concrete *binary clauses* semantics for Java bytecode from which termination is observable, and then define the *size analysis* as an abstract interpretation of this semantics. From the practical side we plan to improve the analyzer by adopting some of the techniques described in [6], which have been proved to be very efficient in practice.

References

1. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *ESOP*, 2007.
2. A. R. Bradley, Z. Manna, and H. B. Sipma. Termination of polynomial programs. In *VMCAI*, 2005.
3. M. Bruynooghe, M. Codish, J. P. Gallagher, S. Genaim, and W. Vanhoof. Termination Analysis of Logic Programs through Combination of Type-Based Norms. *TOPLAS*, 2006. to appear.
4. 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.
5. M. Colón and H. Sipma. Practical methods for proving program termination. In *CAV*, 2002.
6. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, 1978.
8. P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *VMCAI*, 2005.
9. N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Appl. Algebra Eng. Commun. Comput.*, 12(1/2):117–156, 2001.
10. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In *IJCAR*, 2006.
11. C. S. Lee, N. D. Jones, and A. M. Ben-Ammar. The size-change principle for program termination. In *POPL*, 2001.
12. N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In *ICLP*, 1997.
13. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* A-W, 1996.
14. P. M. Hill, E. Payet, and F. Spoto. Path-length analysis of object-oriented programs. In *EAAI*, 2006.
15. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, 2004.
16. A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, 2004.
17. C. Taboch, S. Genaim, and M. Codish. Terminweb: Semantic based termination analyser for logic programs, 2002. `http://www.cs.bgu.ac.il/~mcodish/TerminWeb`.

Path resolution for recursive nested modules is undecidable

Keiko Nakata[1] and Jacques Garrigue[2]

[1] Kyoto University
[2] Nagoya University

Abstract. The ML module system supports the modular development of large programs, through decomposition, abstraction and reuse. To increase its flexibility, much work has been devoted to extending it with recursion. To keep type normalization terminating in such an extension, thus to keep type checking decidable, path references must be resolved in a terminating way. Here paths are a mechanism to refer to components of modules. In this paper, we show that the termination of path resolution is undecidable for a ML-like module system with recursive modules and first-order applicative functors, by encoding any Turing machine. This demonstrates the need for some restriction.

1 Introduction

The ML module system provides strong support for the modular development of programs [8, 5]. A programmer can decompose a large program hierarchically using *nested structures*. *Functors*, which are functions over modules, ease code reuse. Moreover, the programmer can control abstraction of programs with *signatures*, which represent types of modules. To increase its flexibility, much work has been devoted to extending the module system with recursion, which is currently prohibited in ML [1, 9, 3, 6].

In our previous work [6], we have proposed a type system for a module system extended with recursion and fully applicative functors. We had to be careful about the potential existence of cyclic type definitions. In a programming language with recursive modules and applicative functors, a programmer might carelessly write pathologically cyclic type definitions, for which the naïve implementation of type normalization can diverge and thus for which type checking can diverge. To keep type checking decidable, we designed a terminating type normalization by requiring functors not to take functors as arguments or to access sub-modules of arguments.

The first restriction sounds reasonable: type normalization in the presence of higher-order applicative functors and recursive modules amounts to normalization of a lambda calculus with recursion, which is clearly undecidable. While it may seem too strong, the second restriction is also critical; we prove in this paper that termination of type normalization is still undecidable only with first-order functors and sub-module access. Our proof works by encoding any Turing machine into a small calculus featuring *paths*, where paths are a mechanism to refer to modules components. To normalize types in ML, we need to resolve path references (i.e. to find the module that the path refers to). Undecidability of path resolution hence implies that of type normalization.

The result of this paper is important for us since it justifies the need for a restriction on nested arguments. Moreover, the encoding itself exposes the underlying nature of type normalization, which will be useful to find a more relaxed restriction, hence to make our type system more flexible.

This work is initially motivated by a desire to define a decidable type system for recursive modules. Yet the problem we consider is general; we examine termination of an untyped tiny calculus with recursion and labeled records. We also believe that our work is potentially useful for guaranteeing safe evaluation of recursive modules, where we want to ensure the absence of cyclic aliases between modules.

2 Syntax and Semantics

Below, we define a calculus for our formal study, where m and x are metavariables for field names and variables, respectively.

$$\begin{array}{lll} \textit{Expressions} & e & ::= \{m_1 = e_1 \cdots m_n = e_n\ \} \mid \lambda x.e \mid p \\ \textit{Paths} & p & ::= \epsilon \mid x \mid p.m \mid p_1(p_2) \\ \textit{Program} & P & ::= \{m_1 = e_1 \cdots m_n = e_n\ \} \end{array}$$

An expression, ranged over by e, is either a *structure*, a *functor* or a *path*. A structure $\{m_1 = e_1 \cdots m_n = e_n\}$ is a sequence of definitions, that is, a record of expressions e_i labeled with field names m_i. A functor $\lambda x.e$ represents a function over expressions; x is the name of the formal parameter and e is the body.

Paths (ranged over by p) are the most interesting construct of the calculus. They are built from 1) the root path ϵ, which refers to the toplevel structure; 2) variables x; 3) the dot notation "$p.m$", meaning access to the field named m of the structure that p refers to; 4) functor application $p_1(p_2)$, which applies the expression that p_1 refers to to the expression that p_2 refers to. As we shall see in an example later, paths can refer to a field at any level of nesting within the toplevel structure regardless of definition ordering. Thus paths introduce recursion to the calculus. A program, ranged over by P, is a toplevel structure. All occurrences of the root path ϵ in a program are considered to refer to the toplevel structure. We assume that any sequence of definitions in a structure does not bind the same field name twice and that a program does not contain free variables.

For instance, consider the program:

$$\begin{array}{l} \{\ \ \mathtt{m}_1 = \{\mathtt{n}_1 = \{\}\ \ \mathtt{n}_2 = \epsilon.\mathtt{m}_1.\mathtt{n}_1\ \} \\ \quad \mathtt{m}_2 = \lambda\mathtt{x}.\{\mathtt{n}_1 = \{\}\ \ \mathtt{n}_2 = \mathtt{x}.\mathtt{n}_2\ \ \mathtt{n}_3 = \epsilon.\mathtt{m}_2(\mathtt{x}).\mathtt{n}_1\ \} \\ \quad \mathtt{m}_3 = \epsilon.\mathtt{m}_2(\epsilon.\mathtt{m}_1).\mathtt{n}_2\ \} \end{array}$$

The path $\epsilon.\mathtt{m}_1.\mathtt{n}_1$ refers to the field $\mathtt{n}_1$ of the structure $\mathtt{m}_1$. Hence, the path $\epsilon.\mathtt{m}_1.\mathtt{n}_2$, which is an alias for $\epsilon.\mathtt{m}_1.\mathtt{n}_1$, refers to the field $\mathtt{n}_1$ of the structure $\mathtt{m}_1$, too. A path can contain functor applications. For instance, the path $\epsilon.\mathtt{m}_2(\mathtt{x}).\mathtt{n}_1$ refers to the field $\mathtt{n}_1$ of the body of the functor $\mathtt{m}_2$.

Resolution of path references may require more complex computation. For instance, $\epsilon.\mathtt{m}_2(\epsilon.\mathtt{m}_1).\mathtt{n}_2$ resolves to $\epsilon.\mathtt{m}_1.\mathtt{n}_1$; by reducing the functor application, we obtain $\epsilon.\mathtt{m}_1.\mathtt{n}_2$, which resolves to $\epsilon.\mathtt{m}_1.\mathtt{n}_1$, as we have explained above.

2.1 Path rewriting

A program defines a set of rewrite rules on paths. For instance, the previous example gives rewrite rules:

$$\{ \quad \epsilon.\mathtt{m}_1.\mathtt{n}_2 \rightarrow \epsilon.\mathtt{m}_1.\mathtt{n}_1, \quad \epsilon.\mathtt{m}_2(\mathtt{x}).\mathtt{n}_2 \rightarrow \mathtt{x}.\mathtt{n}_2,$$
$$\epsilon.\mathtt{m}_2(\mathtt{x}).\mathtt{n}_3 \rightarrow \epsilon.\mathtt{m}_2(\mathtt{x}).\mathtt{n}_1, \quad \epsilon.\mathtt{m}_3 \rightarrow \epsilon.\mathtt{m}_2(\epsilon.\mathtt{m}_1).\mathtt{n}_2 \ \}$$

According to these rules, we can induce the reduction steps:

$$\epsilon.\mathtt{m}_3 \rightarrow \epsilon.\mathtt{m}_2(\epsilon.\mathtt{m}_1).\mathtt{n}_2 \rightarrow \epsilon.\mathtt{m}_1.\mathtt{n}_2 \rightarrow \epsilon.\mathtt{m}_1.\mathtt{n}_1$$

which reflects our previous explanation of path resolution.

We say that a program P is *well-founded* if the rewrite rules that P defines do not induce infinite reduction steps. The reader will find formal definitions in the accompanying technical report [7].

Example 1. The program:

$$\{\mathtt{m}_1 = \epsilon.\mathtt{m}_2.\mathtt{m}_1 \ \ \mathtt{m}_2 = \epsilon.\mathtt{m}_1\}$$

is not well-founded, since it induces the infinite reduction:

$$\epsilon.\mathtt{m}_1 \rightarrow \epsilon.\mathtt{m}_2.\mathtt{m}_1 \rightarrow \epsilon.\mathtt{m}_1.\mathtt{m}_1 \rightarrow \epsilon.\mathtt{m}_2.\mathtt{m}_1.\mathtt{m}_1 \rightarrow \cdots$$

Example 2. The program:

$$\{\mathtt{m}_1 = \lambda\mathtt{x}.\mathtt{x} \ \ \mathtt{m}_2 = \epsilon.\mathtt{m}_1(\epsilon.\mathtt{m}_2)\}$$

is not well-founded, since it induces infinite reduction:

$$\epsilon.\mathtt{m}_2 \rightarrow \epsilon.\mathtt{m}_1(\epsilon.\mathtt{m}_2) \rightarrow \epsilon.\mathtt{m}_2 \rightarrow \cdots$$

The keen reader may have noticed that when a program does not contain functors at all, the problem of well-foundedness is reduced to termination of head-reduction of a string rewriting system, which is known to be decidable [2]. Yet for programs with first-order functors, well-foundedness is undecidable, as we shall show in the next section.

3 Translation of the Turing Machine

We encode any Turing machine into a first-order fragment of the calculus, defined by the syntax:

Expressions	$e ::= \{m_1 = e_1 \cdots m_n = e_n \ \} \mid \lambda x.e \mid p$
Paths	$p ::= \epsilon \mid x \mid \epsilon.m(p) \mid p.m$
Toplevel expression	$te ::= \lambda x.\{m_1 = e_1 \cdots m_n = e_n \ \}$
Program	$P ::= \{m_1 = te_1 \cdots m_n = te_n\}$

The new syntax is restricted in the following two ways to syntactically preclude higher-order functors. 1) Only paths of the form $\epsilon.m$ can appear in functor positions. 2) A program is a sequence of toplevel expressions, which are lambda abstraction of structures. Observe that, under these two restrictions, the rewrite rules of a program cannot yield paths of the forms $x(p)$ or $x.m(p)$.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, b, F)$ be a Turing machine[3], where Q is the set of states; $\Sigma \subseteq \Gamma$ is the set of input symbols; Γ is the set of tape symbols; δ is the transition function; $q_0 \in Q$ is the start state; b is the blank symbol, which is in Γ but not in Σ; F is the set of final states, which we assume to be empty. In particular, the arguments of $\delta(q, a)$ are a state q and a tape symbol a. The value of $\delta(q, a)$, if it is defined, is a triple (q', a', D), where q' is the next state; a' is the symbol in Γ to be written in the scanned cell of the tape; D is a direction, which is either R (for right) or L (for left).

A configuration $a_1 a_2 \cdots a_{i-1} q a_i a_{i+1} \cdots a_n$ of a Turing machine is represented by a path

$$\epsilon.q(\epsilon.a_{i-1}(\cdots(\epsilon.a_2(\epsilon.a_1(\epsilon.\hat{b}(\epsilon))))\cdots)).a_i.a_{i+1}.\cdots.a_n.\hat{b}$$

The special symbol $\hat{b}$ is not contained in Q or Γ. The intuition is that we encode the right hand side of the tape with the dots and the left side with functor applications. The head part $\epsilon.q$ of the path represents the current state. We put $\hat{b}$ at the inner most functor application and the outermost dot to represent the right and left limits of input symbols on the tape.

Given a Turing machine M, we construct a set of rewrite rules R_M, which is the union of the following sets:

1. $\{\epsilon.q(x).a \to \epsilon.q'(\epsilon.a'(x)) \mid \delta(q, a) = (q', a', R)\}$
2. $\{\epsilon.q(x).a \to x.q'.a' \mid \delta(q, a) = (q', a', L)$
3. $\{\epsilon.q(x).\hat{b} \to \epsilon.q(x).b.\hat{b} \mid q \in Q\}$
4. $\{\epsilon.\hat{b}(x).q \to \epsilon.q(\epsilon.\hat{b}(x)).b \mid q \in Q\}$
5. $\{\epsilon.a(x).q \to \epsilon.q(x).a \mid a \in \Gamma, a \in Q\}$

Below we observe 1) that we can construct a program P_M with R_M as the corresponding set of rewrite rules and 2) that the rewrite rules R_M encode the Turing machine M.

It is easy to see the first condition hold by considering that the left-hand side of every rewrite rule in R_M is of the form $\epsilon.q(x).a$ and that a program $\{q = \lambda x.\{a = p\}\}$ has the set $\{\epsilon.q(x).a \to p\}$ as the corresponding rewrite rule. Note also that R_M does not contain overlapping rules; this is important to avoid a structure containing duplicate definitions for the same name like $\{q = \lambda x.\{a = p_1 \ a = p_2\}\}$, which breaks the syntactic convention we mentioned in Section 2.

For instance, the rules from 5 require the toplevel structure of P_M to contain a definition $a = \lambda x.\{q_1 = \epsilon.q_1(x).a \ \cdots \ q_n = \epsilon.q_n(x).a\}$ when a is a tape symbol of the Turing machine M and $\{q_1, \cdots, q_n\}$ is the set of states.

Let's verify that the second condition holds. The first two sets of rules encode transitions of M. The rules from third and fourth sets allow us to elongate the tape, moving the edge by adding a blank symbol to the left or right on demand.

[3] We borrow the notations and terminology from [4], to which the reader is referred for a complete definition.

Finally, the rules from the last set allow commutation between state and tape symbol. A transition of M can be simulated either by a rule of 1, potentially followed by a rule of 3, or by a rule of 2 followed by a rule of 4 or 5.

For instance, suppose $\delta(q, a_i) = (q', a'_i, L)$; i.e., we have a move

$$a_1 \cdots a_{i-1} q a_i a_{i+1} \cdots a_n \vdash a_1 \cdots a_{i-2} q' a_{i-1} a'_i a_{i+1} \cdots a_n$$

Then we can induce the corresponding reduction of paths by rules $\epsilon.q(x).a_i \to x.q'.a'_i$ from 2, and $\epsilon.a_i(x).q' \to \epsilon.q'(x).a_i$ from 5:

$$\begin{array}{ll} & \epsilon.q(\epsilon.a_{i-1}(\cdots(\epsilon.a_1(\epsilon.\hat{b}(x)))\cdots)).a_i.a_{i+1}.\cdots.a_n.\hat{b} \\ \to & \epsilon.a_{i-1}(\epsilon.a_{i-2}(\cdots(\epsilon.a_1(\epsilon.\hat{b}(x)))\cdots)).q'.a'_i.a_{i+1}.\cdots.a_n.\hat{b} \\ \to & \epsilon.q'(\epsilon.a_{i-2}(\cdots(\epsilon.a_1(\hat{b}(x)))\cdots)).a_{i-1}.a'_i.a_{i+1}.\cdots.a_n.\hat{b} \end{array}$$

4 Conclusion

We have shown that termination of path resolution for first-order nested recursive modules is undecidable by an encoding of the Turing machine. While the result justifies a restriction on nested functor arguments, we think that the current restriction that prohibits all accesses to sub-modules of arguments is stronger than necessary.

Since path rewrite rules are derived from programs, they are already restricted: 1) there are no overlapping rules; 2) every rule is left-linear; 3) functor-application positions in the left-hand side of any rewrite rule must be module variables (*e.g.* a path like $\epsilon.m_1(\epsilon.m_2)$ cannot be in the left-hand side, but $\epsilon.m_1(x)$ can). Besides, in ML, functor parameters are explicitly typed, which means that we can statically know the possible nesting-depth of functor arguments.

A direction for future work would be to exploit these properties to find a relaxed restriction, thus making our type system stronger.

References

1. K. Crary, R. Harper, and S. Puri. What is a recursive module? In *Proc. PLDI'99*, pages 50–63, 1999.
2. M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. In *Proc. LICS'90*, 1990.
3. D. Dreyer. *Understanding and Evolving the ML Module System.* PhD thesis, Carnegie Mellon University, 2005.
4. J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*, chapter 8. Addison-Wesley, 2001.
5. X. Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.
6. K. Nakata and J. Garrigue. Recursive Modules for Programming. In *Proc. ICFP'06*. ACM Press, 2006.
7. K. Nakata and J. Garrigue. Path resolution for recursive nested modules is undecidable. Technical report, CNAM-Laboratoire Cédric, 2007.
8. R.Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML - Revised*. The MIT Press, 1997.
9. C. Russo. Recursive Structures for Standard ML. In *Proc. ICFP'01*, pages 50–61. ACM Press, 2001.

Proving Termination of Recursive Programs by Matching Against Simplified Program Versions and Construction of Specialized Libraries in Theorema

Nikolaj Popov, Tudor Jebelean*

Research Institute for Symbolic Computation, Linz, A–4232 Hagenberg, Austria
popov@risc.uni-linz.ac.at

Abstract. We report work in progress concerning the program verification environment in the *Theorema* system. As part of the system a specialized strategy for proving termination of recursive functional programs is presented. The detailed termination proofs may in many cases be skipped, because the termination conditions are reusable and thus collected in specialized libraries. Enlargement of the libraries is possible by proving termination of each candidate, but also by taking new elements directly from existing libraries.

Introduction.
Proving correctness of recursive programs is still challenging, especially when by correctness is meant total correctness. There are various approaches, however, there is no (and cannot be) general recipe. Termination proofs exposed in classical books (e.g., [8]) are very comprehensive, however, their orientation is theoretical rather than practical. On the other hand there are various tools for proving program correctness automatically or semiautomatically, (see, e.g., [9],[1]), and this is where our contribution falls into.

Termination proofs of individual programs are, in general, *expensive* from the automatic theorem proving point of view. In some cases, program termination, however, may be ensured – and this is our main contribution – by matching against *simplified versions* (of programs) collected in specialized libraries. An idea on how these libraries are constructed is given at the end of the next section.

In our approach, proving total correctness of a program is split into three distinct steps: first – proving coherence, second – proving partial correctness, and third – proving termination. (The combination of the three steps guaranties the total correctness.)

In more detail, we are given a program (by its source text) which computes the function F and we are given its specification by a precondition on the input $I_F[x]$ and a postcondition on the input and the output $O_F[x, y]$.

We say that a program is *coherent* if all the calls made to its auxiliary programs are such that the preconditions of the auxiliary programs are not violated. The following example gives an intuition on what we are doing. Let the program for computing F be:

$$F[x] = \textbf{If } Q[x] \textbf{ then } H[x] \textbf{ else } G[x],$$

where Q is a total predicate and H and G are auxiliary programs. The specification of F is (I_F, O_F) and the specifications of the auxiliary functions are (I_H, O_H) and (I_G, O_G), respectively. The two verification conditions, ensuring that the calls to the auxiliary functions have

* The program verification project in the frame of e-Austria Timişoara is supported by BMBWK (Austrian Ministry of Education, Science and Culture), BMWA (Austrian Ministry of Economy and Work) and MEC (Romanian Ministry of Education and Research). The Theorema project is supported by FWF (Austrian National Science Foundation) – SFB project F1302.

appropriate values, and that F is coherent are:

$$(\forall x : I_F[x])\ (Q[x] \implies I_H[x])$$
$$(\forall x : I_F[x])\ (\neg Q[x] \implies I_G[x]).$$

The coherence check – actually, proving the respective verification conditions – is done at the beginning of the verification process. If the program is not coherent it may still be correct, however, its verification would involve knowledge about the concrete implementation of the auxiliary functions. Thus, the modularity (the easy exchange of program implementations) is lost and it is considered to be out of the scope of our system.

Furthermore, partial correctness and termination are expressed as verification conditions which themselves may be proven without taking into account their order. Moreover, as we have shown in [7], a coherent program (of a certain recursive type) is totaly correct if and only if its verification conditions are valid as logical formulae.

Proving any of the three kinds of verification conditions has its own difficulty, however, our experience shows that proving coherence is relatively easy, proving partial correctness is more difficult and proving the termination verification condition (it is only one condition) is in general the most difficult one. The latter one is expressed by using a *simplified version* of the initial program, and the condition itself expresses a property of that *simplified version.* The proof typically needs an induction prover and the induction step may sometimes be difficult to find. Fortunately, due to the specific structure, the proof is not always necessary, and this is what we discuss in the next section.

Our work is performed in the frame of the *Theorema* system (an overview could be found in [3]), which is a mathematical computer assistant aiming at supporting all the phases of mathematical activity: construction and exploration of mathematical theories, definition of algorithms for problem solving, as well as experimentation and rigorous verification of them. *Theorema* provides functional as well as imperative programming constructs. Moreover, the logical verification conditions which are produced by the methods presented here can be passed to the automatic provers of the system in order to be checked for validity. *Theorema* includes a collection of general as well as specific provers for various interesting domains (e. g., integers, sets, reals, tuples, etc.).

Libraries of Terminating Programs.

In this section we describe the idea of proving termination of recursive programs by creating and exploring libraries of terminating programs, and thus avoiding redundancy of induction proofs. The core idea is that different recursive programs may have the same *simplified version.*

Let us consider the following very simple recursive program for computing the factorial function:

$$Fact[n] = \textbf{ If } n = 0 \textbf{ then } 1 \textbf{ else } n * Fact[n-1], \tag{1}$$

with the specification of $Fact$ (*Input:* $I_{Fact}[n] \iff n \in \mathbb{N}$ and *Output:* $O_{Fact}[n, m] \iff n! = m$). The verification condition for the termination of $Fact$ is expressed using a *simplified version* of the initial function:

$$Fact'[n] = \textbf{ If } n = 0 \textbf{ then } 0 \textbf{ else } Fact'[n-1], \tag{2}$$

namely, the verification condition is

$$(\forall n : n \in \mathbb{N})\ (Fact'[n] = 0). \tag{3}$$

More generally, when having a recursive program which may fit to the scheme:

$$F[x] = \textbf{ If } Q[x] \textbf{ then } S[x] \textbf{ else } C[x, F[R_1[x]], \ldots, F[R_k[x]]], \tag{4}$$

where Q is a predicate and S, C, R_1, …, R_k are auxiliary functions whose total correctness is assumed, the corresponding *simplified version* of F is:

$$F'[x] = \textbf{ If } Q[x] \textbf{ then } 0 \textbf{ else } F'[R_1[x]] + \cdots + F'[R_k[x]],$$

which only depends on Q, R_1, …, R_k. It is obtained by replacing the function S by 0, and the function C by addition (combining the recursive calls). Namely, the termination condition is

$$(\forall x\ : I_F[x])\ (F'[x] = 0),$$

which must be proven, based on the logical formulae corresponding to the definition of F' and the theory of the domain of Q, R_1, …, R_k. Moreover, proving that $(\forall x\ : I_F[x])\ (F'[x] = 0)$ is equivalent to proving termination of $F'[x]$, for all x satisfying $I_F[x]$ (it is so, because if $F'[x]$ terminates it returns 0, and vice versa), which may be used alternatively.

The program scheme (4) is in fact extended to a more general one – it may have more "else" branches with different C functions on each branch, and additionally, on any branch it may have many recursive calls with different R functions.

A soundness (in fact, soundness and completeness) theorem (a program is totaly correct if and only if its verification conditions are valid) concerning recursive schemes with many "else" branches but at most one recursive call on each branch has been proven [7]. As a consequence of that theorem, we know what the necessary (and sufficient) conditions for program correctness are – these are the verification conditions. A new soundness statement concerning schemes which may have many recursive calls on each branch is to be published.

Note, that different recursive programs may have the same *simplified version*. Notably, (2) becomes the *simplified version* of all the unary primitive recursive functions and thus (3) becomes the termination condition of that class. This is easily provable from the definition of primitive recursion.

For serving the termination proofs, we are now creating libraries containing *simplified versions* together with their input conditions, whose termination is proven. The proof of the termination may now be skipped if the *simplified version* is already in the library and this membership check is much easier than an induction proof – it only involves matching against simplified versions.

Starting from a small library – actually it is not only one, but more, because each recursive scheme has several domain based libraries – we intend to enlarge it. One way of doing so is by carrying over the whole proof of any new candidate, appearing during a verification process.

Enlargement within a library is also possible by applying special knowledge retrieval. As we have seen, termination depends on the *simplified version* F' and on the input condition I_F. Considering again the factorial example (1), in order to prove its termination we need to prove (3). Assume, now the pair (2),(3) is in our library. We may now strengthen the input condition I_{Fact} and actually produce a new one:

$$I_{F-new}[n] \Longleftrightarrow (n \in \mathbb{N} \wedge n \geq 100).$$

The *simplified version* $Fact'$ remains the same (2) – we did not change the initial program (1), however, the termination condition becomes:

$$(\forall n : n \in \mathbb{N} \wedge n \geq 100)\ (Fact'[n] = 0), \tag{5}$$

and (after proving the validity) we add it to the library. It is easy to see that any new version of a simplified program which is obtained by strengthening the input condition can also be included in the library without further proof. Assume

$$(\forall x \ : I_F[x]) \ (F'[x] = 0)$$

is a member of a library. Then for any "stronger" input condition $I_{F-strng}$, we have:

$$I_{F-strng}[x] \Longrightarrow I_F[x],$$

and thus

$$(\forall x \ : I_{F-strng}[x]) \ (F'[x] = 0).$$

This is of course not the case for weakening the input condition. Consider the following weakening of I_{Fact}:

$$I_{F-real}[n] \Longleftrightarrow (n \in \mathtt{R}),$$

which leads to nontermination of our $Fact'$ as defined in (3), that is:

$$(\forall n \ : n \in \mathtt{R}) \ (Fact'[n] = 0),$$

which is not anymore a valid formula.

Strengthening of input conditions leads to preserving the termination properties and thus enlarging a library without additional proof is possible. However, for a fixed *simplified version*, keeping (and collecting in some cases) the weakest input condition is the most efficient strategy, because then proving the implication from stronger to weaker condition is relatively easier.

Implementation and experiments. The method described above is implemented in the *Theorema* system. The recursive schemes, we have studied so far, are not covering all the possibilities, however, we are extending them to more complex ones, e.g., mutual recursion and nested recursion. Termination proofs may be done by using a *Theorema* prover (see, e.g., [3],[4]). However, delivering the proof problem itself to another specialized tool (e.g., [6],[5]) is also possible. Enlarging the libraries by taking (and adopting) *simplified versions* directly from other libraries (e.g., the *Coq Library* [2]) can be considered as well.

References

1. Y. Bertot, P. Casteran. Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer Verlag, 2004.
2. F. Blanqui , S. Hinderer, S. Coupet-Grimal, W Delobel, A. Kroprowski. A Coq library on rewriting and termination. *http://coq.inria.fr/contribs/CoLoR.html*
3. B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. In *Journal of Applied Logic, vol. 4, issue 4*, pp. 470–504, 2006.
4. B. Buchberger, D. Vasaru. Theorema: The Induction Prover over Lists. In *First International Theorema Workshop*, RISC, Hagenberg, Austria, June 1997.
5. B. Cook, A. Podelski, A. Rybalchenko. Terminator: Beyond safety. In *In Computer-Aided Verification (CAV06), LNCS 4144*, pp. 415–418, Seattle, USA, 2006.
6. N. Hirokawa, A. Middeldorp. Tyrolean Termination Tool. In *Proceedings of the 16th International Conference on Rewriting Techniques and Applications (RTA), LNCS 3467*, Nara, Japan, 2005.
7. L. Kovacs, N. Popov, T. Jebelean. Combining Logic and Algebraic Techniques for Program Verification in Theorema. In *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2006)*, Paphos, Cyprus, 2006.
8. J. Loeckx, K. Sieber. The Foundations of Program Verification. Teubner, second edition, 1987.
9. PVS: Specification and Verification System. *http://pvs.csl.sri.com*

Nontermination of String Rewriting using SAT[⋆]

Harald Zankl and Aart Middeldorp

Institute of Computer Science, University of Innsbruck, Austria

Abstract. This note presents a new approach for proving nontermination of SRSs. We encode rewrite sequences as propositional formulae which are satisfiable whenever the corresponding sequence includes a looping reduction, which can be constructed from the assignment.[1]

1 Introduction

Establishing termination of rewrite systems is a challenging endeavor since it is undecidable in general. Nevertheless these days many powerful algorithms exist which are able to automatically prove termination for a huge class of rewrite systems as witnessed by the annual termination competition.[2] The few techniques known for proving nontermination are based on narrowing [4], unfolding [5], matchbounds [6], and ancestor graphs [7].

2 Encoding Looping Reductions in SAT

A *string rewrite system* (SRS) $\mathcal{S}$ is called looping if there exist a context C, a substitution σ, and a term t with $t \to_{\mathcal{S}}^{+} C[t\sigma]$. When applying the dependency pair approach [1] one can forget about leading contexts due to minimality considerations, and a looping reduction takes the form $u \to_{\mathcal{S} \cup \mathsf{DP}(\mathcal{S})}^{+} u\sigma$ where rules in $\mathsf{DP}(\mathcal{S})$ are applied at root positions and the ones in $\mathcal{S}$ below the root.

Our idea is to encode a looping rewrite sequence in a matrix of dimension $n \times m$ with $(0,0)$ the top leftmost and $(n-1, m-1)$ the bottom rightmost entry. Every row in the matrix corresponds to a string and from row i to row $i+1$ a rewrite step is performed. We are now going to define the necessary properties one by one after the following example.

Example 1. Consider the SRS $\mathcal{S} = \{\mathsf{ab} \to \mathsf{bbaa}\}$ with the dependency pair $\mathsf{Ab} \to \mathsf{Aa}$ which forms a cycle in the dependency graph. In a 3×5 matrix a looping reduction is possible. The entries marked with $\cdot$ indicate that any symbol might appear at these positions.

$$\begin{array}{ccccc}
\mathsf{A} & \mathsf{b} & \mathsf{b} & \cdot & \cdot \\
\mathsf{A} & \mathsf{a} & \mathsf{b} & \cdot & \cdot \\
\mathsf{A} & \mathsf{b} & \mathsf{b} & \mathsf{a} & \mathsf{a}
\end{array}$$

[⋆] This research is supported by FWF (Austrian Science Fund) project P18763.

[1] The idea of encoding computation as propositional satisfiability goes back to [2].

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

For the propositional representation we need the following variables:

M_{ij}^a	indicating that symbol a occurs at position (i,j) in the matrix,
$R_i^{l\to r}$	indicating that in row i of the matrix rule $l \to r$ applies,
$\mathbf{p}_i$	indicating the position (= column) in row i where the rule is applied (in the example $\mathbf{p}_0 = 0$ and $\mathbf{p}_1 = 1$), and
$\mathbf{e}_i$	pointing to the last symbol of the term (in the example possible values for $\mathbf{e}_0$, $\mathbf{e}_1$, and $\mathbf{e}_2$ are 2, 2, and 4).

The variables $\mathbf{p}_i$ and $\mathbf{e}_i$ are not boolean but represent integer values which are implemented as lists of boolean variables encoding the actual integer value in binary. To distinguish them from proper propositional variables they are denoted in boldface.

Exactly one function symbol: To get exactly one function symbol per matrix entry, multiple symbols at the same entry have to be banned and at least one symbol must occur. Note that dependency pair symbols (those in $\mathcal{F}^\sharp \setminus \mathcal{F}$) can only appear at the root position. We express this property below where $X = \mathcal{F}$ if $j > 0$ and $X = \mathcal{F}^\sharp \setminus \mathcal{F}$ otherwise:

$$\alpha_{ij} = \Big(\bigvee_{a\in X} M_{ij}^a\Big) \wedge \bigwedge_{a\in X}\big(M_{ij}^a \to \bigwedge_{b\in X\setminus\{a\}} \neg M_{ij}^b\big)$$

Rule application: If a rule $l \to r$ applies in row i we have to match the left-hand side somewhere in row i and the right-hand side in row $i+1$ of the matrix (see $\mathit{applies}_i^{l\to r}$). Furthermore all entries which are not affected by a rule application must be copied from row i to row $i+1$ (see $\mathit{copy}_{ij}^{l\to r}$). The position of the rule application is fixed by $\mathbf{p_i}$ and satisfying $\mathit{copy}_{ij}^{l\to r}$ has the side effect that only one rule is applied.

$$\beta_i^{l\to r} = R_i^{l\to r} \to \big(\mathit{applies}_i^{l\to r} \wedge \bigwedge_{0\leqslant j<m} \mathit{copy}_{ij}^{l\to r}\big)$$

where in case of $l \to r \in \mathcal{S}$ we have

$$\begin{aligned}\mathit{applies}_i^{l\to r} = \bigwedge_{0\leqslant j<|l|} M_{i(\mathbf{p}_i+j)}^{l_{j+1}} \wedge \bigwedge_{0\leqslant j<|r|} M_{(i+1)(\mathbf{p}_i+j)}^{r_{j+1}} \\ \wedge\, (\mathbf{e}_{i+1} + |l| = \mathbf{e}_i + |r|) \wedge (\mathbf{e}_i \geqslant \mathbf{p}_i + |l|)\end{aligned}$$

and if $l \to r \in \mathsf{DP}(\mathcal{S})$ then $\mathbf{p}_i$ specializes to 0 and the conjunct $\mathbf{p}_i = 0$ is added. The formula $\mathit{applies}_i^{l\to r}$ applies the left-hand side in row i and the right-hand side in row $i+1$ at the abstract position $\mathbf{p}_i$. The last but one conjunct demands that the end pointer in line $i+1$ takes the value of $\mathbf{e}_i - |l| + |r|$. To ensure that the contracted redex fits in line i the last conjunct must be satisfied.

The formula for $\mathit{copy}_{ij}^{l\to r}$ is defined as $\top$ if $j + \max\{|l|, |r|\} \geqslant m$ (these entries would be outside the matrix) and otherwise as

$$\Big((j < \mathbf{p}_i) \wedge \bigwedge_{a\in X}\big(M_{ij}^a \leftrightarrow M_{(i+1)j}^a\big)\Big) \vee \Big((j \geqslant \mathbf{p}_i) \wedge \bigwedge_{a\in\mathcal{F}}\big(M_{i(j+|l|)}^a \leftrightarrow M_{(i+1)(j+|r|)}^a\big)\Big)$$

(where $X = \mathcal{F}^\sharp \setminus \mathcal{F}$ if $j = 0$ and $X = \mathcal{F}$ otherwise) if $l \to r \in \mathcal{S}$ and in case $l \to r \in \mathsf{DP}(\mathcal{S})$ as $\bigwedge_{a \in \mathcal{F}} \big(M^a_{i(j+|l|)} \leftrightarrow M^a_{(i+1)(j+|r|)}\big)$.

All entries in the matrix before the position where the rule is applied are copied from row i to $i+1$. The second disjunct copies the entries after $\mathbf{p}_i$ which are unaffected when applying the rule. The position of these entries change if the applied rule is not length-preserving. For rules $l \to r \in \mathsf{DP}(\mathcal{S})$ we know that $\mathbf{p}_i = 0$ and hence the formula simplifies to the one shown above.

Initial term is reached again: To recognize a loop the string in some line $i > 0$ must match the one in line zero. Furthermore the end pointer for line i is demanded not to be smaller than the one of line zero.

$$\gamma = \bigvee_{0<i<n} \Big((\mathbf{e}_i \geqslant \mathbf{e}_0) \wedge \bigwedge_{a \in \mathcal{F}^\sharp \setminus \mathcal{F}} \big(M^a_{00} \leftrightarrow M^a_{i0}\big) \wedge \bigwedge_{\substack{0<j<m \\ a \in \mathcal{F}}} \big(M^a_{0j} \leftrightarrow M^a_{ij}\big) \Big)$$

Putting everything together: The formula $NT_{\mathcal{S}}$ is defined as

$$\Big(\bigwedge_{0 \leqslant i < n} \big(\bigwedge_{0 \leqslant j < m} \alpha_{ij}\big) \wedge (\mathbf{e}_i < m) \wedge \beta_i \Big) \wedge \gamma$$

with

$$\beta_i = \bigvee_{l \to r \in \mathcal{S} \cup \mathsf{DP}(\mathcal{S})} R^{l \to r}_i \;\wedge \bigwedge_{l \to r \in \mathcal{S} \cup \mathsf{DP}(\mathcal{S})} \beta^{l \to r}_i$$

expressing that one rule must apply and it is applied properly. The condition $\mathbf{e}_i < m$ ensures that all rewrite steps are performed within the allowed matrix dimensions.

There are two types of variables keeping track of function symbols—concrete (M^a_{ij}) and abstract ones ($M^a_{i\mathbf{x}}$) where $\mathbf{x}$ is a list of propositional variables representing an integer in binary. The latter ones are needed when a rule is applied at the abstract position $\mathbf{p}_i$. In the current encoding, abstract variables $M^{\mathsf{a}}_{3[x_1;x_0]}$ and $M^{\mathsf{a}}_{3[y_1;y_0]}$ denote two different expressions and hence may take different values. If the assignments for x_1 and y_1 as well as x_0 and y_0 are the same, we want to enforce that the variables take identical values. In the implementation we test for every such abstract variable whether it matches a concrete one and we identify them if that is the case:

$$\varphi_{aux} = \bigwedge_{0 \leqslant i < n} \bigwedge_{0 \leqslant j < m} \bigwedge_{M^a_{i\mathbf{p}}} \big(\mathbf{p} = j \to (M^a_{ij} \leftrightarrow M^a_{i\mathbf{p}})\big)$$

Lemma 2. *If $NT_{\mathcal{S}} \wedge \varphi_{aux}$ is satisfiable then $\mathcal{S}$ admits a looping reduction.* □

3 Experimental Results

The 322 SRSs from the termination problem data base 2006 have been considered. All tests have been performed on a laptop with a clock rate of 2.2GHz and

1GB memory. A time limit of 60 seconds was used. Satisfiability of the propositional formulae was tested by the award winning SAT solver MiniSat [3]. Our implementation is build on top of T_TT2[3], the successor of T_TT.

After computing the SCCs in the estimated dependency graph our implementation tries to remove rules which cannot contribute to a nonterminating sequence (to be precise, linear polynomial interpretations with coefficients in $\{0, 1\}$ are applied to get smaller SCCs). We anticipate that applying more advanced termination criteria will result in a speedup of our implementation. Our solver tries matrices of different dimensions. The heuristic first tries small dimensions, then increases the number of rows and columns, respectively. The maximum number of rows and columns we try is 28. Needless to say, it is to some extent designed for the given database. Counterexamples are found mostly within a few seconds if they exist but the SAT solver seems to run forever if the SRS is not looping.

In the SRS category of the 2006 termination competition Jambox could disprove 25 SRSs terminating, Matchbox and AProVE came second with 12 systems followed by TORPA with 4 systems. TEPARLA proved nontermination of a single system whereas CiME, MultumNonMulta, TPA, and T_TTbox appear not to have implemented any method for proving nontermination. Our implementation proves nontermination of 24 SRSs within the given time limit. All of these SRSs are also handled by Jambox. The missing system is secret2006_matchbox_1 (which takes our tool 13 seconds using dimension 15×20). Nevertheless, our SAT based approach is not subsumed by the methods implemented in Jambox. None of the tools participating in last year's competition can handle any of the 335 challenging and small SRSs collected by Johannes Waldmann,[4] whereas we can disprove termination of several of them.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
2. S.A. Cook. The complexity of theorem-proving procedures. In *Proc. 3rd STOC*, pages 151–158. ACM, 1971.
3. N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. 7th SAT*, volume 2919 of *LNCS*, pages 502–518, 2004.
4. J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proc. 5th FroCoS*, volume 3717 of *LNAI*, pages 216–231, 2005.
5. É. Payet. Detecting non-termination of term rewriting systems using an unfolding operator. In *Proc. 16th LOPSTR*, volume 4407 of *LNCS*, pages 194–209, 2006.
6. J. Waldmann. Matchbox: A tool for match-bounded string rewriting. In *Proc. 15th RTA*, volume 3091 of *LNCS*, pages 85–94, 2004.
7. H. Zantema. Termination of rewriting proved automatically. *Journal of Automated Reasoning*, 34:105–139, 2005.

[3] `colo6-c703.uibk.ac.at/ttt2`
[4] `dfa.imn.htwk-leipzig.de/matchbox/new-problems/srs-2007-02-28/`

Implementing RPO using SAT[⋆]

Peter Schneider-Kamp[1], René Thiemann[1], Elena Annov[2], Michael Codish[2], and Jürgen Giesl[1]

[1] LuFG Informatik 2, RWTH Aachen, Germany,
{psk,thiemann,giesl}@informatik.rwth-aachen.de
[2] Department of Computer Science, Ben-Gurion University, Israel,
{mcodish,annov}@cs.bgu.ac.il

1 Introduction

We introduce a propositional encoding of the recursive path order with status (RPO) on terms. The RPO combines the multiset path order and the lexicographic path order, where one also considers permutations in the lexicographic comparison. The proposed encoding allows us to use SAT solvers to efficiently automate the search for an RPO when proving termination of term rewriting.

Since last year, several papers have illustrated the potential in applying SAT solvers for termination analysis of term rewrite systems (TRSs). The key idea is classic: the termination problem for a TRS is encoded to a propositional formula that is satisfiable if the TRS has the desired termination property (for RPO we even have “iff”). Satisfiability is decided using state of the art SAT solvers.

For the lexicographic path order (LPO), such an encoding is described in [1] and extended in [2] for the context of dependency pairs. This paper extends that work by introducing an encoding for RPO. The main new and interesting components are the encodings for the lexicographic comparison w.r.t. permutation and for the multiset extension of the base order.

2 Preliminaries

Let $\geq_{\mathcal{F}}$ denote a quasi-order (a so-called *precedence*) on the set of function symbols $\mathcal{F}$ and let $>_{\mathcal{F}} = (\geq_{\mathcal{F}} \setminus \leq_{\mathcal{F}})$ and $\approx_{\mathcal{F}} = (\geq_{\mathcal{F}} \cap \leq_{\mathcal{F}})$. Each precedence $\geq_{\mathcal{F}}$ and status function σ induce a recursive path order $\succ_{rpo}$ on terms. If $\ell \succ_{rpo} r$ holds for each rule $\ell \to r$ in a TRS, then the TRS is RPO-terminating.

Definition 1 (status and multiset cover). *A status function σ maps each $f \in \mathcal{F}$ of arity n to* `mul` *or to a permutation on $\{1, \ldots, n\}$ and each pair of tuples of terms $\bar{s} = \langle s_1, \ldots, s_n \rangle$ and $\bar{t} = \langle t_1, \ldots, t_m \rangle$ to a multiset cover (γ, ε) s.t. $\gamma : \{1, \ldots, m\} \to \{1, \ldots, n\}$, $\varepsilon : \{1, \ldots, n\} \to \{false, true\}$ and for each $1 \leq i \leq n$, if $\varepsilon(i)$ (indicating equality) then $\{j \mid \gamma(j) = i\}$ is a singleton set.*

The status indicates if the arguments are compared lexicographically w.r.t. a permutation or as multisets. That a status also maps pairs $\bar{s}$ and $\bar{t}$ to a multiset cover is non-standard, but facilitates the encoding to SAT. Such a multiset cover for $\bar{s}$ and $\bar{t}$ is a mapping between indices which specifies which element $i = \gamma(j)$ of $\bar{s}$ covers element j of $\bar{t}$ and in terms of $\varepsilon(i)$ if that covering is by equality.

[⋆] Supported by the Deutsche Forschungsgemeinschaft DFG under grant GI 274/5-1.

Definition 2 (recursive path order with status). *For a precedence $\geq_{\mathcal{F}}$ and status function σ we define the relations $\succ_{rpo}$ and $\sim_{rpo}$ on terms. We use the notation $\bar{s} = \langle s_1, \ldots, s_n \rangle$ and $\bar{t} = \langle t_1, \ldots, t_m \rangle$.*

- *$s \succ_{rpo} t$ iff $s = f(\bar{s})$ and one of the following holds:*
  (1) $s_i \succ_{rpo} t$ or $s_i \sim_{rpo} t$ for some $1 \leq i \leq n$; or
  (2) $t = g(\bar{t})$ and $s \succ_{rpo} t_j$ for all $1 \leq j \leq m$ and either:
  (i) $f >_{\mathcal{F}} g$ or (ii) $f \approx_{\mathcal{F}} g$ and $\bar{s} \succ_{rpo}^{f,g} \bar{t}$;
- *$s \sim_{rpo} t$ iff (a) $s = t$; or (b) $s = f(\bar{s})$, $t = g(\bar{t})$, $f \approx_{\mathcal{F}} g$, and $\bar{s} \sim_{rpo}^{f,g} \bar{t}$;*

where $\succ_{rpo}^{f,g}$ and $\sim_{rpo}^{f,g}$ are the tuple extensions of $\succ_{rpo}$ and $\sim_{rpo}$ defined by:

- *$\langle s_1, \ldots, s_n \rangle \succ_{rpo}^{f,g} \langle t_1, \ldots, t_m \rangle$ iff one of the following holds:*
  (1) σ maps f and g to permutations μ_f and μ_g; and
  $\mu_f \langle s_1, \ldots, s_n \rangle \succ_{rpo}^{lex} \mu_g \langle t_1, \ldots, t_n \rangle$ where $\langle u_1, \ldots, u_n \rangle \succ_{rpo}^{lex} \langle v_1, \ldots, v_m \rangle$ iff (a) $m = 0$ and $n > 0$; or (b) $u_1 \succ_{rpo} v_1$; or
  (c) $u_1 \sim_{rpo} v_1$ and $\langle u_2, \ldots, u_n \rangle \succ_{rpo}^{lex} \langle v_2, \ldots, v_m \rangle$;
  (2) σ maps f and g to `mul`; and $\langle \bar{s}, \bar{t} \rangle$ to a multiset cover (γ, ε) such that for all i, j, if $\gamma(j) = i$ then $\varepsilon(i) \rightarrow s_i \sim_{rpo} t_j$ and $\neg\varepsilon(i) \rightarrow s_i \succ_{rpo} t_j$; and for some i, $\neg\varepsilon(i)$, i.e., some s_i is not used for equality.
- *$\langle s_1, \ldots, s_n \rangle \sim_{rpo}^{f,g} \langle t_1, \ldots, t_m \rangle$ iff $n = m$ and one of the following holds:*
  (1) σ maps f and g to μ_f and μ_g; and for all i, $s_{\mu_f(i)} \sim_{rpo} t_{\mu_g(i)}$.
  (2) σ maps f and g to `mul`; and $\langle \bar{s}, \bar{t} \rangle$ to a multiset cover (γ, ε) such that for all i, $\varepsilon(i)$ and for some $1 \leq j \leq m$ we have $\gamma(j) = i$ and $s_i \sim_{rpo} t_j$.

Definition 2 can be specialized to other standard path orderings by taking specific forms of status functions: lexicographic path order (LPO) when σ maps all symbols to the identity permutation; lexicographic path order w.r.t. permutation (LPOS) when σ maps all symbols to some permutation; multiset path order (MPO) when σ maps all symbols to *mul*.

The RPO termination problem is to determine for a given TRS if there exists a precedence and a status function such that the system is RPO terminating. There are two variants of the problem: "strict-" and "quasi-RPO termination" depending on if the precedence, $\geq_{\mathcal{F}}$, is strict or not. The corresponding decision problems, strict- and quasi-RPO termination, are decidable and NP complete. In this paper we address the implementation of decision procedures for RPO termination problems by encoding them into corresponding SAT problems.

3 Encoding RPO problems

We introduce an encoding τ which maps constraints of the form $s \succ_{rpo} t$ to propositional statements about the status and the precedence of the symbols in the terms s and t. A satisfying assignment for the encoding of such a constraint indicates a precedence and a status function such that the constraint holds.

The first part of the encoding is straightforward and similar to [1, 2]. All "missing" cases (e.g., $\tau(x \succ_{rpo} t)$ for variables x) are defined to be *false*.

$$\tau(f(\bar{s}) \succ_{rpo} t) = \bigvee_{i=1}^{n} (\tau(s_i \succ_{rpo} t) \vee \tau(s_i \sim_{rpo} t)) \quad \vee \quad \tau_2(f(\bar{s}) \succ_{rpo} t)$$

$$\tau_2(f(\bar{s}) \succ_{rpo} g(\bar{t})) = \bigwedge_{j=1}^{m} \tau(f(\bar{s}) \succ_{rpo} t_j) \wedge \Big((f \succ_{\mathcal{F}} g) \vee ((f \approx_{\mathcal{F}} g) \wedge \tau(\bar{s} \succ_{rpo}^{f,g} \bar{t}))\Big)$$

$$\tau(s \sim_{rpo} s) = true \qquad\qquad \tau(f(\bar{s}) \sim_{rpo} g(\bar{t})) = (f \approx_{\mathcal{F}} g) \wedge \tau(\bar{s} \sim_{rpo}^{f,g} \bar{t})$$

We proceed to show how to encode lexicographic comparisons w.r.t. permutation and multiset comparisons by $\succ_{lex}^{f,g}$ and $\succ_{mul}$. Then, we combine the two into $\succ_{rpo}^{f,g}$.

With each symbol $f \in \mathcal{F}$ (of arity n) we associate n^2 propositional variables $f_{i,k}$ with $i, k \in \{1, \ldots, n\}$. Here, $f_{i,k}$ is *true* iff $\mu_f(i) = k$ (i.e., the i-th argument of $f(s_1, \ldots, s_n)$ is considered at k-th position when comparing lexicographically). For the encoding to be correct, we introduce constraints on the variables $f_{i,k}$ ensuring that they indeed correspond to a permutation on $\{1, \ldots, n\}$. To encode $\succ_{lex}^{f,g}$, we define auxiliary relations $\succ_{lex}^{f,g,k}$, where $k \in \mathbb{N}$ denotes that the k-th component of $\bar{s}$ and $\bar{t}$ is being compared. Thus, $\succ_{lex}^{f,g} = \succ_{lex}^{f,g,1}$ and we obtain:

$$\tau(\bar{s} \succ_{lex}^{f,g,k} \bar{t}) = \begin{cases} false & \text{if } k > n \\ true & \text{if } m < k \leq n \\ \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} (f_{i,k} \wedge g_{j,k} \rightarrow & \text{otherwise} \\ \quad (\tau(s_i \succ_{rpo} t_j) \vee (\tau(s_i \sim_{rpo} t_j) \wedge \tau(\bar{s} \succ_{lex}^{f,g,k+1} \bar{t})))) & \end{cases}$$

$$\tau(\bar{s} \sim_{lex}^{f,g} \bar{t}) = (n = m) \wedge \left(\bigwedge_{k=1}^{n} \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} f_{i,k} \wedge g_{j,k} \rightarrow \tau(s_i \sim_{rpo} t_j) \right)$$

With each pair $\bar{s}$ and $\bar{t}$ of term tuples, we associate $n * m$ propositional variables $\gamma_{i,j}$, where $\gamma_{i,j}$ is *true* iff s_i covers t_j, and n variables ε_i, where ε_i is *true* iff s_i is used to cover a t_j by equality. For the below encoding to be correct, we introduce constraints on these variables to ensure that the implied mappings are indeed a multiset cover. Then we obtain:

$$\tau(\bar{s} \succsim_{mul} \bar{t}) = \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} (\gamma_{i,j} \rightarrow ((\varepsilon_i \rightarrow \tau(s_i \sim_{rpo} t_j)) \wedge (\neg\varepsilon_i \rightarrow \tau(s_i \succ_{rpo} t_j))))$$

$$\tau(\bar{s} \succ_{mul} \bar{t}) = \tau(\bar{s} \succsim_{mul} \bar{t}) \wedge \neg \bigwedge_{i=1}^{n} \varepsilon_i \qquad\qquad \tau(\bar{s} \sim_{mul} \bar{t}) = \tau(\bar{s} \succsim_{mul} \bar{t}) \wedge \bigwedge_{i=1}^{n} \varepsilon_i$$

Finally, to combine $\succ_{lex}^{f,g}$ and $\succ_{mul}$ into $\succ_{rpo}^{f,g}$, we introduce for each symbol $f \in \mathcal{F}$ one variable m_f, which is *true* iff the arguments of f are to be compared as multisets (i.e., the status function maps f to `mul`). Then we encode:

$$\tau(\bar{s} \circ_{rpo}^{f,g} \bar{t}) = \Big(m_f \wedge m_g \wedge \tau(\bar{s} \circ_{mul} \bar{t})\Big) \vee \Big(\neg m_f \wedge \neg m_g \wedge \tau(\bar{s} \circ_{lex}^{f,g} \bar{t})\Big) \qquad \text{for } \circ \in \{\succ, \sim\}$$

Similar to Def. 2, the above encoding function τ can be specialized to other standard path orderings: lexicographic path order w.r.t. permutation when m_f is set to *false* for all $f \in \mathcal{F}$; lexicographic path order when additionally $f_{i,k}$ is set to *true* iff $i = k$; multiset path order when m_f is set to *true* for all $f \in \mathcal{F}$.

4 Implementation and Experiments

We have implemented the encoding of RPO with status in the termination analyzer AProVE [3]. The implementation is modularized, such that all path orders using lexicographic and/or multiset comparisons can be encoded. The implementation supports also RPO with argument filter.[3]

The table below summarizes the results of the experiments running on the set of 865 TRSs from the TPDB [4]. All experiments were run on a 2.2 GHz AMD Athlon 64 with a time-out of 60 seconds (comparable to the setting of the annual International Termination Competition). For each encoding we provide the number of TRSs which can be proved terminating (with the number of time-outs in brackets) and the total analysis times (in seconds) for the full collection.

The first two rows compare the performance of our new SAT-based approach to the dedicated solvers for path orders in AProVE 1.2 which do not use SAT solving. The third and the fourth row apply path orders (combined with argument filters) within the dependency pair framework.

The columns show data for LPO with strict and quasi-precedence (denoted *lpo*/*qlpo*), for LPO with status (*lpos*/*qlpos*), for MPO (*mpo*/*qmpo*), and for RPO with status (*rpo*/*qrpo*).

	Solver	*lpo*	*qlpo*	*lpos*	*qlpos*	*mpo*	*qmpo*	*rpo*	*qrpo*
1	SAT-based	**123** (0)	**127** (0)	**141** (0)	**155** (0)	**92** (0)	**98** (0)	**146** (0)	**162** (0)
	(direct)	**31.0**	**44.7**	**26.1**	**40.6**	**49.4**	**74.2**	**50.0**	**85.3**
2	dedicated	**123** (5)	**127**(16)	**141** (6)	**154**(45)	**92** (7)	**98**(31)	**145**(10)	**158** (65)
	(direct)	**334.4**	**1426.3**	**460.4**	**3291.7**	**653.2**	**2669.1**	**908.6**	**4708.2**
3	SAT-based	**357** (0)	**389** (0)	**362** (0)	**395** (2)	**369** (0)	**408** (1)	**375** (0)	**416** (2)
	(arg. filt.)	**79.3**	**199.6**	**69.0**	**261.1**	**110.9**	**267.8**	**108.8**	**331.4**
4	dedicated	**350**(55)	**374**(79)	**355**(57)	**380**(92)	**359**(69)	**391**(82)	**364**(74)	**394**(102)
	(arg. filt.)	**4039.6**	**5469.4**	**4522.8**	**6476.5**	**5169.7**	**5839.5**	**5536.6**	**7186.1**

The table shows an orders of magnitude improvement over existing dedicated solvers both for direct analysis with recursive path orders and for the combination of recursive path orders and argument filter in the dependency pair framework. Note that without a time-out, this effect would only be aggravated.

References

1. M. Codish, V. Lagoon, and P. J. Stuckey. Solving partial order constraints for LPO termination. In *Proc. RTA '06*, LNCS 4098, pages 4–18, 2006.
2. M. Codish, P. Schneider-Kamp, V. Lagoon, R. Thiemann, and J. Giesl. SAT solving for argument filterings. In *Proc. LPAR '06*, LNAI 4246, pages 30–44, 2006.
3. J. Giesl, P. Schneider-Kamp, and R. Thiemann AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In *Proc. IJCAR '06*, LNAI 4130, pages 281–286, 2006.
4. The termination problem data base. `http://www.lri.fr/~marche/tpdb/`.

[3] More details will be described in the full version of this paper. A preliminary version of this article was presented as a *Short Paper at LPAR '06*.

SAT Solving for Termination Analysis with Polynomial Interpretations[⋆]

Carsten Fuhs[1], Jürgen Giesl[1], Aart Middeldorp[2], Peter Schneider-Kamp[1], René Thiemann[1], and Harald Zankl[2]

[1] LuFG Informatik 2, RWTH Aachen, Germany,
[2] Institute of Computer Science, University of Innsbruck, Austria,

1 Termination of TRSs and Polynomial Interpretations

Most termination methods for TRSs transform the termination problem into a set of inequalities between terms. For example, a classical approach is to generate inequalities $\ell \succ r$ for all rules $\ell \to r$ or, if one uses the dependency (DP) framework [1, 3, 5], then one generates strict inequalities for the DPs and non-strict ones for the usable rules. Consider the following example for subtraction.

$$\mathsf{p}(\mathsf{0}) \to \mathsf{0} \qquad \mathsf{minus}(x, \mathsf{0}) \to x$$
$$\mathsf{p}(\mathsf{s}(x)) \to x \qquad \mathsf{minus}(x, \mathsf{s}(y)) \to \mathsf{minus}(\mathsf{p}(x), y)$$

For the DP of the recursive minus-call we get the following constraints. Here, M is the tuple-symbol of minus.

$$\mathsf{p}(\mathsf{0}) \succsim \mathsf{0} \quad (1) \qquad \mathsf{p}(\mathsf{s}(x)) \succsim x \quad (2) \qquad \mathsf{M}(x, \mathsf{s}(y)) \succ \mathsf{M}(\mathsf{p}(x), y) \quad (3)$$

A popular method to search for relations $\succ$ and $\succsim$ automatically are *polynomial interpretations* [8]. A polynomial interpretation $\mathcal{P}ol$ maps each n-ary function symbol f to a polynomial $f_{\mathcal{P}ol}$ over n variables $x_1, \ldots, x_n$ with coefficients from $\mathbb{N} = \{0, 1, 2, \ldots\}$. It is extended to a mapping $[\cdot]_{\mathcal{P}ol}$ on terms where $[x]_{\mathcal{P}ol} = x$ for variables x and $[f(t_1, \ldots, t_n)]_{\mathcal{P}ol} = f_{\mathcal{P}ol}\{x_1/[t_1]_{\mathcal{P}ol}, \ldots, x_n/[t_n]_{\mathcal{P}ol}\}$. We often write $[\cdot]$ if $\mathcal{P}ol$ is clear from the context. Now a term u is greater (resp. greater-equal) than v iff $[u] \geq [v] + 1$ (resp. $[u] \geq [v]$) holds for all instantiations of the variables with natural numbers. For instance, the constraints of the example are satisfied using the polynomial interpretation $\mathcal{P}ol_1$ with $\mathsf{M}_{\mathcal{P}ol_1} = x_2$, $\mathsf{p}_{\mathcal{P}ol_1} = x_1$, $\mathsf{s}_{\mathcal{P}ol_1} = x_1 + 1$, and $\mathsf{0}_{\mathcal{P}ol_1} = 0$. Thus, termination of the example is proved.

To find such interpretations automatically, one starts with an *abstract* polynomial interpretation. In the linear case we obtain

$$f_{\mathcal{P}ol} = f_0 + f_1 x_1 + \cdots + f_n x_n \quad \text{for each } f \text{ with arity } n \tag{4}$$

where the coefficients f_i are left open. Then one translates the term constraints into polynomial constraints. In the example we obtain

$$\mathsf{p}_0 + \mathsf{p}_1 \mathsf{0}_0 \geq \mathsf{0}_0 \quad (5) \qquad (\mathsf{p}_0 + \mathsf{p}_1 \mathsf{s}_0) + (\mathsf{p}_1 \mathsf{s}_1) * x \geq x \quad (6)$$
$$(\mathsf{M}_0 + \mathsf{M}_2 \mathsf{s}_0) + \mathsf{M}_1 * x + \mathsf{M}_2 \mathsf{s}_1 * y \geq (\mathsf{M}_0 + \mathsf{M}_1 \mathsf{p}_0 + 1) + \mathsf{M}_1 \mathsf{p}_1 * x + \mathsf{M}_2 * y \quad (7)$$

Next one simplifies these constraints by deleting the variables $x, y, \ldots$ that are

[⋆] Supported by the DFG grant GI 274/5-1 and the FWF project P18763.

(implicitly) universally quantified. To this end, instead of an inequality between polynomials we only compare the respective coefficients ("absolute positiveness" [7]). In the example, the resulting constraints are (5), (8) and (9) (using $x = 0 + 1 * x$), and (10) – (12).

$$\mathsf{p}_0 + \mathsf{p}_1\mathsf{s}_0 \geq 0 \quad (8) \qquad \mathsf{p}_1\mathsf{s}_1 \geq 1 \quad (9)$$
$$\mathsf{M}_2\mathsf{s}_0 \geq \mathsf{M}_1\mathsf{p}_0 + 1 \quad (10) \qquad \mathsf{M}_1 \geq \mathsf{M}_1\mathsf{p}_1 \quad (11) \qquad \mathsf{M}_2\mathsf{s}_1 \geq \mathsf{M}_2 \quad (12)$$

Now to prove termination one has to show the *satisfiability* of such *Diophantine constraints* over the naturals. In the example, a solution of the constraints is $\mathsf{0}_0 = \mathsf{p}_0 = \mathsf{M}_0 = \mathsf{M}_1 = 0$ and $\mathsf{p}_1 = \mathsf{s}_0 = \mathsf{s}_1 = \mathsf{M}_2 = 1$. In this way, the abstract interpretation is turned into the polynomial interpretation $\mathcal{P}ol_1$.

In the next section, we show how to check this satisfiability using SAT solvers.

2 Encoding Diophantine Constraints to SAT

To encode Diophantine constraints into SAT we first present a mapping $||\cdot||$ from polynomials to tuples of propositional formulas which are interpreted as *binary representations* of the polynomials. We restrict the search to coefficients in the range $\{0, \ldots, 2^k - 1\}$ for a fixed k. Then each coefficient f is encoded into $||f|| = \langle f^{k-1}, \ldots, f^0\rangle$ where $f^0, \ldots, f^{k-1}$ are propositional variables. Similarly, a natural number $n = b_\ell * 2^\ell + \ldots b_1 * 2^1 + b_0$ is encoded into $||n|| = \langle b_\ell, \ldots, b_1, b_0\rangle$ where 0 and 1 are identified with false and true. So if $k = 2$ then $||\mathsf{s}_0|| = \langle \mathsf{s}_0^1, \mathsf{s}_0^0\rangle$ and $||6|| = \langle 1, 1, 0\rangle$. For addition and multiplication, we introduce operations B^+ and B^* on tuples of propositional formulas and define

$$||p + q|| = B^+(||p||, ||q||) \quad \text{and} \quad ||p * q|| = B^*(||p||, ||q||)$$

for all polynomials p and q. For B^+ we essentially use the idea of a ripple-carry-adder. The details are presented in [2]. For example $||\mathsf{s}_0 + 6|| = \langle \mathsf{s}_0^1, \neg\mathsf{s}_0^1, \neg\mathsf{s}_0^1, \mathsf{s}_0^0\rangle$. We encode multiplication by summing up partial products as follows:

- $B^*(\langle\varphi_1, \ldots, \varphi_n\rangle, \langle\psi\rangle) = \langle\varphi_1 \wedge \psi, \ldots, \varphi_n \wedge \psi\rangle$
- $B^*(\langle\varphi_1, \ldots, \varphi_n\rangle, \langle\psi_1, \ldots, \psi_m\rangle) = B^+(\langle\varphi_1 \wedge \psi_1, \ldots, \varphi_n \wedge \psi_1, \overbrace{0, \ldots, 0}^{m-1 \text{ times}}\rangle,$
  $B^*(\langle\varphi_1, \ldots, \varphi_n\rangle, \langle\psi_2, \ldots, \psi_m\rangle)\,)$ if $m \geq 2$.

Now we extend $||\cdot||$ to map each Diophantine constraint to a formula (not to a tuple). To this end, we define the operation $B^{\geq}$ which encodes comparisons.

$$||p \geq q|| = B^{\geq}(||p||, ||q||)$$

For $B^{\geq}$ we apply zero-padding and compare tuples lexicographically:

- $B^{\geq}(\langle\varphi\rangle, \langle\psi\rangle) = \psi \to \varphi$
- $B^{\geq}(\langle\varphi_1, \ldots, \varphi_n\rangle, \langle\psi_1, \ldots, \psi_n\rangle) = (\varphi_1 \wedge \neg\psi_1) \vee$
  $((\varphi_1 \leftrightarrow \psi_1) \wedge B^{\geq}(\langle\varphi_2, \ldots, \varphi_n\rangle, \langle\psi_2, \ldots, \psi_n\rangle))$ if $n \geq 2$.

So to determine the satisfiability of a set of Diophantine constraints $p_i \geq q_i$ with coefficients from $\{0, \ldots, 2^k - 1\}$, we encode it as a conjunction $\bigwedge_i ||p_i \geq q_i||$ of propositional formulas. Then we use a SAT solver to find an assignment for the coefficients. Note that the space complexity of our encoding is polynomial. More precisely, whenever all numbers in "$p \geq q$" are smaller than $2^k - 1$, then the size of $||p \geq q||$ is in $\mathcal{O}(|p \geq q|^2 * k^2)$.

3 Polynomials with Negative Constant

Now we regard polynomials $f_{\mathcal{P}ol}$ which may have a negative constant coefficient (i.e., in (4) one may have $f_0 < 0$). All other coefficients still have to be natural numbers. This is needed if we replace the recursive minus-rule by $\mathsf{minus}(x, \mathsf{s}(y)) \to \mathsf{minus}(\mathsf{p}(x), \mathsf{p}(\mathsf{s}(y)))$. Then the constraints are (1), (2), and

$$\mathsf{M}(x, \mathsf{s}(y)) \succ \mathsf{M}(\mathsf{p}(x), \mathsf{p}(\mathsf{s}(y))), \tag{13}$$

which cannot be satisfied using non-negative polynomial interpretations. Thus, we use a polynomial interpretation $\mathcal{P}ol_2$ with $\mathsf{p}_{\mathcal{P}ol_2} = x_1 - 1$. To avoid that terms are mapped to negative integers (which would violate well-foundedness), [6] modified the interpretation of terms $[\cdot]$. Now one defines $[\mathsf{p}(x)] = \max(x-1, 0)$.

The problem is that with these interpretations, inequalities like $u \succsim v$ are transformed into $[u] \geq [v]$ where $[u]$ and $[v]$ are no polynomials anymore, as they contain "max". To solve this problem, let us first regard *concrete* polynomial interpretations (where the coefficients are actual numbers). Here, [6] presented an approach to transform inequalities like $[u] \geq [v]$ into ordinary polynomial inequalities without "max". The idea is to define an under-approximation $[.]^{left}$ and an over-approximation $[.]^{right}$ which do not contain "max" anymore. Then instead of $[u] \geq [v]$ one requires $[u]^{left} \geq [v]^{right}$.

Definition 1 (**$[.]^{left}$ [6]**). *For every polynomial q we denote its constant part by $con(q)$. For any term t, we define the polynomial $[t]^{left}$ as follows:*[3]

$$[t]^{left} = \begin{cases} t & \text{if } t \text{ is a variable} \\ 0 & \text{if } t = f(t_1, \ldots, t_n),\ q - con(q) = 0, \text{ and } 0 > con(q) \\ q & \text{if } t = f(t_1, \ldots, t_n), \text{ otherwise} \end{cases}$$

where $q = f_{\mathcal{P}ol}([t_1]^{left}, \ldots, [t_n]^{left})$.

For example, using $\mathcal{P}ol_2$ we obtain $[\mathsf{p}(x)]^{left} = x - 1 \leq \max(x-1, 0) = [\mathsf{p}(x)]$. The reason is that $q = \mathsf{p}_{\mathcal{P}ol_2}(x) = x - 1$ and thus $con(q) = -1$ and $q - con(q) = x$. If $\mathcal{P}ol_2$ is defined like our previous interpretation $\mathcal{P}ol_1$ on all remaining function symbols except p, then the constraints (1), (2), and (13) are satisfied and termination of our modified example is proved.

The disadvantage of Def. 1 is that one can only compute $[t]^{left}$ and $[t]^{right}$ for concrete polynomial interpretations, since otherwise q contains coefficients and it depends on the assignment whether $q - con(q) = 0$ and $0 > con(q)$ is valid.

For example, we would use an abstract interpretation with $\mathsf{p}_{\mathcal{P}ol} = \boldsymbol{\mathsf{p}_0} + \mathsf{p}_1 x_1$. Here, p_1 may only be instantiated by natural numbers, whereas a coefficient like $\boldsymbol{\mathsf{p}_0}$ (written in **bold** face) may be instantiated by integers.

The idea is to introduce new coefficients $\boldsymbol{b_t^{left}}$ and $\boldsymbol{b_t^{right}}$ for any term t and to

[3] The definition of $[.]^{right}$ is similar to $[.]^{left}$, see [2, 6] for details.

create additional Diophantine constraints α_t^{left} and α_t^{right} which guarantee that $\boldsymbol{b}_t^{left}$ and $\boldsymbol{b}_t^{right}$ are instantiated correctly, depending on the possible values for the other coefficients. To this end, we express the conditions $q - con(q) = 0$ and $0 > con(q)$ from Def. 1 as Diophantine constraints and one can encode $u \succsim v$ by

$$\alpha_t^{left} \wedge \alpha_t^{right} \wedge [u]^{left} \geq [v]^{right}.$$

Here, $[\cdot]^{left}$ and $[\cdot]^{right}$ interpret terms with the help of the additional coefficients $\boldsymbol{b}_t^{left}$ and $\boldsymbol{b}_t^{right}$. For example, $\mathsf{p}(x) \succsim x$ is transformed into:

$$\begin{array}{rll} (\mathsf{p}_1 = 0 \wedge 0 > \mathbf{p_0}) \rightarrow \boldsymbol{b}_{\mathsf{p}(x)}^{left} = 0 & \wedge & \\ \neg(\mathsf{p}_1 = 0 \wedge 0 > \mathbf{p_0}) \rightarrow \boldsymbol{b}_{\mathsf{p}(x)}^{left} = \mathbf{p_0} & \wedge & \mathsf{p}_1 x + \boldsymbol{b}_{\mathsf{p}(x)}^{left} \geq x \end{array}$$

So we obtain polynomial constraints containing **bold** coefficients like $\mathbf{p_0}$ and $\boldsymbol{b}_{\mathsf{p}(x)}^{left}$ which may be instantiated by *integers*. In [2] it is shown how to remove these coefficients in order to apply our SAT encoding afterwards.

4 Implementation, Experiments, and Conclusion

We implemented our new SAT-based approach for polynomial interpretations in the termination prover AProVE [4] and used state-of-the art SAT solvers for the experiments. The SAT-based search for polynomial interpretations is by orders of magnitude faster than other non-SAT-based search algorithms. This holds in particular if one considers polynomials with higher coefficients or degrees. For details on the experiments and for the full paper [2] we refer to our evaluation web site at `http://aprove.informatik.rwth-aachen.de/eval/SATPOLO`.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
2. C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT Solving for Termination Analysis with Polynomial Interpretations. In *Proc. SAT'07*, LNCS 4501, pages 340–354, 2007.
3. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In *Proc. LPAR'04*, LNAI 3452, pages 301–331, 2005.
4. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the Dependency Pair Framework. In *Proc. IJCAR'06*, LNAI 4130, pages 281–286, 2006.
5. N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1,2):172–199, 2005.
6. N. Hirokawa and A. Middeldorp. Tyrolean termination tool: Techniques and features. *Information and Computation*, 205(4):474–511, 2007.
7. H. Hong and D. Jakuš. Testing positiveness of polynomials. *Journal of Automated Reasoning*, 21(1):23–38, 1998.
8. D. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA, 1979.

Solving polynomial constraints over the reals in proofs of termination[⋆]

Salvador Lucas

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

1 Introduction

Polynomial interpretations are receiving an increasing attention for generating orderings which can be used to solve the constraints which are computed in termination provers [1, 2, 5]. In fact, many termination tools use polynomials as a principal ingredient to achieve termination proofs. Although most tools implement polynomials with (non-negative) *integer* coefficients, the use of polynomials with *real* coefficients in proofs of polynomial termination of rewriting has been recently proved strictly more powerful than using polynomial interpretations with integer coefficients only [6]. This also happens when automatic proofs of termination proceed by using or mixing more powerful techniques, see [6, 7] for further motivation. In this setting, we have shown that it is enough to consider the set $\mathbb{R}_{\text{alg}}$ of the *real algebraic numbers*[1] for the coefficients of the polynomials [7]. *Efficiently* solving the obtained constraints over the appropriate domains of such coefficients is a crucial aspect which has to be addressed in order to make this framework really useful in practice.

2 Solving polynomial constraints over small domains

We have developed an algorithm to solve existential polynomial constraints over *small* subsets of real algebraic numbers [7]. The idea of the algorithm is very simple: it basically tries to evaluate the polynomial part P of a constraint $P > 0$ (or $P \geq 0$) by instantiating the variables occurring in P with values in D. Of course, expressed in this way, this is just a 'brute force' approach which, in general, would be unfeasible due to the combinatory explosion and the huge amount of arithmetic operations. In the following, we sketch how to avoid these problems by choosing suitable domains for the coefficients, appropriate representations for the polynomial components of the constraints, and a number of techniques for safely avoiding a complete exploration of the search space.

⋆ Work partially supported by the EU (FEDER) and the Spanish MEC, under grants TIN 2004-7943-C04-02 and HA 2006-0007, and by the Generalitat Valenciana under grant GV06/285.

1 A real number $x \in \mathbb{R}$ is said to be *algebraic* if it satisfies an equation $x^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0 = 0$, of finite degree n where $a_i \in \mathbb{Q}$ for $0 \leq i \leq n-1$.

Finite domains of square roots of powers of 2. The algorithm works for subsets D of powers of 2: rational numbers like $D \subseteq D_m = \{0\} \cup \{\pm(2^i) \mid i \in \mathbb{Z}, 0 \leq |i| \leq m\}$ for $m \in \mathbb{N}$ or even square roots of powers of 2: $D \subseteq \overline{D}_m = \{0\} \cup \{\pm(2^{\frac{i}{2}}) \mid i \in \mathbb{Z}, 0 \leq |i| \leq 2m\}$. We write D_m^+ if we restrict the attention to non-negative numbers. In particular, $D_1^+ = \{0, 2^{-1}, 2^0, 2^1\} = \{0, \frac{1}{2}, 1, 2\}$ includes the non-negative coefficients used (by default) in all currently available termination tools; furthermore, $D_0^+ = \{0, 2^0\} = \{0, 1\}$ already includes the default option in tools like AProVE and $D_0 = \{-1, 0, 1\}$ is the domain used in TTT. We can also use $\overline{D}_1 = \{0, \pm 2^{-1}, \pm 2^{-\frac{1}{2}}, \pm 2^0, \pm 2^{\frac{1}{2}}, \pm 2^1\} = \{0, \pm\frac{1}{2}, \pm\frac{1}{\sqrt{2}}, \pm 1, \pm\sqrt{2}, 2\}$.

Restricting the attention to such kind of domains allows us reducing the costs of polynomial *arithmetics*, by changing most powers and products by binary shiftings and additions, see [7] for details.

Polynomial constraints. We deal with constraints $P \geq 0$ or $P > 0$ where polynomial P is represented as $P = (V, M, N, K)$. Here, V is the set of variables in P; M and N are the positive and negative non-constant monomials in P, respectively, and K is the constant coefficient (which can be either positive, negative or null).

Constraints C are sets of *basic* constraints $c = (P, cond)$ where $cond \in \{weak, strict\}$ indicates how P compares to 0: *weakly* ($P \geq 0$) or *strictly* ($P > 0$).

Constraint propagation. The constraint solving algorithm makes extensive use of *partial evaluation* of polynomials P w.r.t. one of its variables for doing *constraint propagation*. For instance, given $P(X_1, X_2, \ldots, X_n)$ and some $d \in D$, we could need to obtain $P_{1,d}(X_2, \ldots, X_n) = P(d, X_2, \ldots, X_n)$. This involves the partial evaluation of *each* monomial $\mathsf{m} = cX_1^{\alpha_1} \cdots X_n^{\alpha_n}$ in P and the reconfiguration of the obtained polynomial as a tuple $(V - \{X_1\}, M', N', K')$.

An important aspect of the algorithm is performing frequent *partial checkings* of the constraints in order to cut the search space. This means that we are often able to conclude the truth or falsity of a basic constraint $c = (P, cond)$ *with variables* without instantiating any variable in P. A (three valued) predicate $checkCS$ performs this task. $checkCS(c)$ returns either $true$ if c is definitely true, or $false$ if c is definitely false, or ?? otherwise. According to the representation $P = (V, M, N, K)$, and assuming that D is a domain of *non-negative* numbers, we have the following cases (here expressed in logical form for saving space):

$$\begin{array}{ll} M \equiv 0 \wedge K < 0 \Rightarrow P \not\geq 0 \wedge P \not> 0 & N \equiv 0 \wedge K > 0 \Rightarrow P \geq 0 \wedge P > 0 \\ N \equiv 0 \wedge K = 0 \Rightarrow P \geq 0 & M \equiv 0 \wedge K = 0 \Rightarrow P \not> 0 \end{array}$$

here $M \equiv 0$ means that M is identically null. More elaborated mechanisms are described in [7].

The algorithm. We can describe our algorithm by means of two mutually recursive functions $solveCS$ and $solveCSvar$. The initial call is $solveCS(D, [], [], C)$. Now, we briefly describe these functions:

1. $solveCS(D, V, pSol, C)$ performs an initial checking of all basic constraints $(P, cond)$ in the constraint C by using $checkCS$. If all constraints are true, then a singleton containing a pair $(V, pSol)$ consisting of the list of previously visited variables V and the list $pSol$ of partial solutions for these variables is returned. A partial solution is just a list $d_1, \ldots, d_k$ of values which correspond to the current list of visited variables $x_1, \ldots, x_k$, i.e., $x_i \mapsto d_i$ will be a binding of the final solution of the constraint. When the final solution is returned, variables x which were not instantiated receive a binding $x \mapsto d$ for an arbitrary $d \in D$ (typically $x \mapsto 0$).
2. $solveCSvar(D, V, pSol, C)$ tries all values $d \in D$ on a variable x_i occurring in a constraint $c = (P, cond)$ in C. The instantiation of x_i with a value d yields a new constraint $c_{i,d} = (P_{i,d}, cond)$ consisting of the partial evaluation $P_{i,d}$ of P with d on the variable x_i and the same condition $cond$. The constraint $c_{i,d}$ is checked by using $checkCS$ and if the inconsistency of $c_{i,d}$ is proved, then d is discarded as a possible value for solving c on x_i. Otherwise, the variable x_i is recorded as 'visited' and the value d which permits to make progress is registered in the list of tuples which are partial solutions. Also, each constraint in $C - \{c\}$ is partially evaluated w.r.t. x_i and d as above and a new problem $C_{i,d}$ is raised. If $c_{i,d}$ is found true, then the constraint solving process continues with $C_{i,d}$. If nothing can be said about $c_{i,d}$, then the constraint solving process continues with $\{c_{i,d}\} \cup C_{i,d}$.

The complete description of the two functions can be found in [7].

3 Implementation

We have implemented the algorithm as part of the tool MU-TERM [4] which is written in the functional language Haskell. We call this new version MU-TERMSD. In order to evaluate the algorithm, we have compared its performance with respect to the previous implementation of MU-TERM (MU-TERM 4.3).

In [4, 5] we show how to use a *Diophantine* constraint solver (like C*i*ME's [1]) to generate non-negative rational coefficients: coefficients c which are intended to be non-negative rational number are treated as fractions $\frac{p}{q}$ of non-negative integer coefficients which (after appropriately transforming the constraints) can be solved by using C*i*ME's constraint solver (see [4, 5] for more details).

In our benchmarks we have considered termination problems taken from the 2006 Termination Problems Data Base[2] (TPDB) (version 3.2). We have considered the first 20 examples in the TRS/Zantema folder of the TPDB. With MU-TERMSD (restricted to domains of rational coefficients), 12 of the 20 examples in Zantema's folder could be solved by using polynomial interpretations with rational coefficients. Furthermore, since MU-TERM's expert first try is for integer coefficients, we can actually say that the use of rational coefficients is actually *necessary* for MU-TERM to achieve the proof. In this setting, MU-TERM 4.3 only succeeded on 4 examples (it was unable to get a proof for the remaining 8 within the usual 60s. time-out), see

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

`http://www.dsic.upv.es/~slucas/mutermSD/Zantema`

This confirms that (as expected) handling constraints over the rationals using a direct solver like the one reported here is better than proceeding as explained in [4, 5] (and implemented in MU-TERM 4.3). In some cases, the improvements on the time required for solving particular problems is quite impressive.

Example 1. Consider the following example:

```
H(x,C(y,z)) -> H(C(S(y),x),z)
H(C(S(x),C(S(0),y)),z) -> H(y,C(S(0),C(x,z)))
```

from TPDB 3.2 folder `secret06/jambox/1.trs`. A proof of termination of this TRS can be obtained with MU-TERM 4.3. There is a single SCC in the estimated dependency graph which consists of the dependency pairs

```
(1) H'(x,C(y,z)) -> H'(C(S(y),x),z)
(2) H'(C(S(x),C(S(0),y)),z) -> H'(y,C(S(0),C(x,z)))
```

The following polynomial interpretation

```
[H](X1,X2) = 0            [S](X) = 1/2.X    [H'](X1,X2) = 2.X1 + X2
[C](X1,X2) = 2.X1 + X2    [0] = 2
```

makes (2) strictly compatible with the ordering $>$ whereas (1) is compatible with the quasi-ordering $\geq$ induced by the interpretation. Thus, (2) is removed and a new SCC consisting of (1) remains to be solved. But it is easily treated by using the subterm criterion [3]. MU-TERM 4.3 obtains the proof in around 15 seconds. A much faster proof can be obtained with MU-TERMSD in less than 0,05 seconds: the observed speed-up is more than 300 for this example.

Furthermore, we have a good evidence that the new solver is competitive even when it works as a Diophantine constraint solving algorithm, see [7].

References

1. E. Contejean, C. Marché, A.-P. Tomás, and X. Urbain. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, 34(4):325-363, 2006.
2. N. Hirokawa and A. Middeldorp. Polynomial Interpretations with Negative Coefficients. In *Proc. of AISC'04*, LNAI 3249:185-198, Springer-Verlag, Berlin, 2004.
3. N. Hirokawa and A. Middeldorp. Dependency Pairs Revisited. In *Proc. of RTA'04*, LNCS 3091:249-268, Springer-Verlag, Berlin, 2004.
4. S. Lucas. MU-TERM: A Tool for Proving Termination of Context-Sensitive Rewriting In *Proc. of RTA'04*, LNCS 3091:200-209, 2004. Available: `http://www.dsic.upv.es/~slucas/csr/termination/muterm`.
5. S. Lucas. Polynomials over the reals in proofs of termination: from theory to practice. *RAIRO Theoretical Informatics and Applications*, 39(3):547-586, 2005.
6. S. Lucas. On the relative power of polynomials with real, rational, and integer coefficients in proofs of termination of rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 17(1):49-73, 2006.
7. S. Lucas. Practical Use of Polynomials Over the Reals in Proofs of Termination. In *Proc. of PPDP'07*, to appear, 2007.

Fast SAT-based polynomial constraint solving for termination tools[⋆]

Salvador Lucas and Rafael Navarro

DSIC, Universidad Politécnica de Valencia, Spain
`e-mail:{slucas,rnavarro}@dsic.upv.es`

1 Introduction

Proofs of termination in term rewriting involve solving constraints between terms s and t coming from (parts of) the rules of the Term Rewriting System (TRS). Many termination tools use polynomials as a principal ingredient to achieve termination proofs. In this setting, each k-ary symbol $f \in \mathcal{F}$ is given a *parametric* polynomial like, e.g., $a_k x_k + \cdots + a_1 x_1 + a_0$, where $a_0, \ldots, a_k$ are *unknown* coefficients. Constraints $s \succeq t$ and $s > t$ are treated as *polynomial constraints* $P_{s,t} \geq 0$ and $P_{s,t} > 0$, respectively, where $P_{s,t} = [s] - [t]$ is the polynomial obtained from terms s and t by interpreting them as polynomials $[s]$ and $[t]$, see [1, 5]. Such constraints are often transformed into *existential* polynomial constraints (where all variables correspond to the *parametric coefficients*) which are expected to be solved on a suitable domain D of *coefficients*. In practice, all termination tools take (as default or unique option) small domains D like $\{0, 1\}$, $\{-1, 0, 1\}$, $\{0, 1, 2\}$, or $\{0, \frac{1}{2}, 1, 2\}$.

A recent paper proposes the use of SAT techniques for solving polynomial constraints in termination provers [2]. The paper shows how to encode polynomial constraints —including integers within a given *range*, arithmetics and comparisons— as *propositional formulae* which can be sent to a state-of-the-art SAT solver (e.g., MiniSat[1]) in order to eventually get the original constraint solved. Fuhs et al.'s benchmarks show that, indeed, using $D = \{0, 1\}$ as the domain for coefficients in polynomial interpretations (i.e., fixing the *range* to 1) is already a very powerful option in comparison to bigger domains. The information in the following table has been taken from [2].

Coeff. Range:	1	2	3
# Success	421	431	434
% Success	49,1	49,8	50,2
Time	45.5 s.	91.8 s.	118.6 s.

It corresponds to the benchmarks performed with the new version of AProVE [3] which implements a SAT-based solver for polynomial constraints (AProVE-SAT). The termination problems come from the (TRS termination category of

⋆ Work partially supported by the EU (FEDER) and the Spanish MEC, under grants TIN 2004-7943-C04-02 and HA 2006-0007, and by the Generalitat Valenciana under grant GV06/285.

1 `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/MiniSat.html`

the) Termination Problem Data Base (TPDB, version 3.2)[2]. 865 examples were considered. Three different ranges for coefficients were considered, corresponding to $N_1 = \{0,1\}$, $N_2 = \{0,1,2\}$, and $N_3 = \{0,1,2,3\}$.

Table above shows that APro-VE-SAT increments the ratio of solved examples in $0,7\%$ when using coefficients from N_2 instead of N_1, but the time required for achieving the proofs is *duplicated!* This means that specifically considering the (small) domain N_1 in order to obtain a more efficient solver on this particular domain still makes sense. We have addressed this task. Our results are reported in the following sections.

2 SAT-solving for constraints over N_1

In this section we give a simple encoding of polynomial constraints as propositional formulas which obtains a better performance when N_1 is considered for solving them. Since variables in the considered polynomials range on N_1 and for all $x \in N_1$ and all $n > 0$ we have $x^n = x$, when considering the representation of a polynomial P, we can *replace* monomials $\mathsf{m} = cX_1^{\alpha_1} \cdots X_n^{\alpha_n}$ in P by $\mathsf{m}' = cX_1^{\beta_1} \cdots X_n^{\beta_n}$ where $\beta_i = 1$ if $\alpha_i \neq 0$ and $\beta_i = 0$ if $\alpha_i = 0$. Then, we add all coefficients of monomials of the same degree $\beta_1, \ldots, \beta_n$ to obtain a single one and proceed like that to obtain a simpler representation P' of P.

The definition of the translation function τ can be found in Figure 1, where Q is a polynomial, c and K are numeric constants (with $c \neq 0$), $X_1, \ldots, X_n$ are variables (ranging on N_1), $\mathsf{rmM}_{X_1,\ldots,X_n}(P)$ removes all monomials in P which include *all* variables $X_1, \ldots, X_n$, and $\mathsf{rmV}_{X_1,\ldots,X_n}(P)$ removes from P all occurrences of variables in $X_1, \ldots, X_n$. According to the discussion above, we only have to deal with polynomials consisting of monomials like $cX_1 \cdots X_n$ (i.e., *without* any power greater than 1). C and C' are polynomial constraints. We only consider the 'and' boolean operator $\wedge$; the extension to other connectives is straightforward.

$$
\begin{aligned}
\tau(K \geq 0) &= \mathit{True}, \quad \text{if } K \geq 0 \\
\tau(K \geq 0) &= \mathit{False}, \quad \text{if } K < 0 \\
\tau(K > 0) &= \mathit{True}, \quad \text{if } K > 0 \\
\tau(K > 0) &= \mathit{False}, \quad \text{if } K \leq 0 \\
\tau(cX_1 \ldots X_n + Q \geq 0) &= \left(\left(\bigvee\nolimits_{1 \leq i \leq n} \neg X_i\right) \wedge \tau(\mathsf{rmM}_{X_1,\ldots,X_n}(Q) \geq 0)\right) \vee \\
&\quad \left(\left(\bigwedge\nolimits_{1 \leq i \leq n} X_i\right) \wedge \tau(\mathsf{rmV}_{X_1,\ldots,X_n}(Q) + c \geq 0)\right) \\
\tau(cX_1 \ldots X_n + Q > 0) &= \left(\left(\bigvee\nolimits_{1 \leq i \leq n} \neg X_i\right) \wedge \tau(\mathsf{rmM}_{X_1,\ldots,X_n}(Q) > 0)\right) \vee \\
&\quad \left(\left(\bigwedge\nolimits_{1 \leq i \leq n} X_i\right) \wedge \tau(\mathsf{rmV}_{X_1,\ldots,X_n}(Q) + c > 0)\right) \\
\tau(C \wedge C') &= \tau(C) \wedge \tau(C')
\end{aligned}
$$

Fig. 1. SAT encoding of polynomial constraints over N_1

[2] see `http://www.lri.fr/~marche/tpdb` and `http://www.lri.fr/~marche/termination-competition`.

In order to obtain a propositional formula in CNF format which is sent to MiniSat, we have used the algorithm in [6].

3 Benchmarks

We have compared the behavior of MU-TERM [4] when different SAT-based polynomial constraint solving engines are used to prove termination of TRSs and the domain of coefficients for polynomials is restricted to N_1:

1. MU-TERM-SAT implements the translation of polynomial constraints into propositional formulas in Section 2 and then calls to MiniSat to obtain a solution.
2. MU-TERM-ApSAT calls to an external module implementing the SAT-based constraint solving algorithm described in [2]; it also uses MiniSat.

As in [2], we have considered *all* 865 TRS termination examples in the TPDB, version 3.2 (this number is denoted as E below). The tools were executed under OS Linux Ubuntu 4.1.1-13ubuntu5, on a Intel Core 2 CPU at 2.13 GHz and 1 GByte of primary memory. Complete information about all benchmarks can be found here: `http://www.dsic.upv.es/~rnavarro/wst07/benchmarks`. The following table summarizes our results.

	MU-TERM-SAT	MU-TERM-ApSAT
Yes	345	346
No	500	486
Time-outs	20	33
Time	1311,46 s.	3893,93 s.

It shows the number of successful and failed proofs (respectively *Yes* and *No*) and the number of time-outs obtained by the two versions of MU-TERM. Again, all benchmarks were made under the usual 60 seconds time out of the termination competition. Row *Time* shows the total time taken by MU-TERM to either find a proof, or conclude that it is *not* able to find a proof. As can be deduced from this table, MU-TERM-SAT is $\frac{3893{,}93}{1311{,}46} \approx 3,0$ times *faster* than MU-TERM-ApSAT in the average.

Tools for proving termination do not use a *single* technique for proving termination. Termination provers rather proceed stepwise by following some particular sequence of several techniques which are given 'local' time-outs which are a (small) fraction of the global time-out. In order to evaluate the efficiency of our SAT-encoding in this respect, we have also considered the behavior of the two solvers when different time-outs (below 60 seconds) are considered. The following table shows our results.

Tool:	MU-TERM-SAT			MU-TERM-ApSAT		
Time out	Yes	No	TO	Yes	No	TO
1 s.	328	404	133	244	135	486
10 s.	342	460	63	326	399	140
30 s.	344	492	29	341	456	68
60 s.	345	500	20	346	486	33

The benchmarks show that our encoding of polynomial constraints over N_1 as propositional formulas and the use of state-of-the-art SAT solvers (e.g., MiniSat) is actually faster that the one in [2] when used with N_1. Furthermore, this is crucial when small time-outs are used.

3.1 Local behavior of the solvers

It is also interesting to pay some attention to the the *particular* behavior of the solvers on each example. In order to explore this aspect, we have taken as a reference the results obtained by MU-TERM-SAT on each example and then computed the speed-up w.r.t. MU-TERM-ApSAT: $S_i = \frac{t'_i}{t_i}$ for $1 \leq i \leq E$, where t_i is the time taken by MU-TERM-SAT on the i-th example, t'_i corresponds to the time taken by MU-TERM-ApSAT, and $E = 865$ is the number of considered examples. Then, we have taken the arithmetic mean ($MS = \frac{\sum_{i=1}^{E} S_i}{E}$) and the geometric mean ($GS = \sqrt[E]{\Pi_{i=1}^{E} S_i}$). Thus, the bigger MS or GS is than 1, the faster is MU-TERM-SAT than MU-TERM-ApSAT on *each* particular example (in the average). We get $MS \simeq 25$ and $GS \simeq 10$.

4 Conclusions

Our work shows the advantage of using *dedicated* algorithms for dealing with constraints to be solved over particular (but sufficiently interesting) domains. A subject for future work is investigating similar specialized encodings of polynomial constraints over bigger (e.g., N_2) or more general domains (e.g., subsets of rational numbers) using SAT techniques.

Acknowledgements. We warmly thank Jürgen Giesl and Peter Schneider-Kamp for providing an executable version of the SAT-based polynomial constraint solving engine implemented in AProVE-SAT [2].

References

1. E. Contejean, C. Marché, A.-P. Tomás, and X. Urbain. Mechanically proving termination using polynomial interpretations. Journal of Automated Reasoning, 34(4):325-363, 2006.
2. C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT Solving for Termination Analysis with Polynomial Interpretations. In *Proc. of SAT'07*, LNCS 4501:340-354, 2007.
3. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In *Proc. of IJCAR'06*, LNAI 4130:281-286, 2006.
4. S. Lucas. MU-TERM: A Tool for Proving Termination of Context-Sensitive Rewriting In *Proc. of RTA'04*, LNCS 3091:200-209, 2004.
5. S. Lucas. Polynomials over the reals in proofs of termination: from theory to practice. *RAIRO Theoretical Informatics and Applications*, 39(3):547-586, 2005.
6. D. Sheridan. The Optimality of a Fast CNF Conversion and its use with SAT. In *Proc. of SAT'04*, 2004.

The Termination Competition 2007

Claude Marché[1] and Johannes Waldmann[2] and Hans Zantema[3]

[1] INRIA Futurs, ProVal, Parc Orsay Université, F-91893 Orsay cedex, France
`Claude.Marche@inria.fr`
[2] Hochschule für Technik, Wirtschaft und Kultur (FH) Leipzig
Fb IMN, PF 30 11 66, D-04251 Leipzig, Germany
`waldmann@imn.htwk-leipzig.de`
[3] Department of Computer Science, Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
`h.zantema@tue.nl`

Abstract. The fourth Termination Competition took place in June 2007. We present the background, results and conclusions of this competition.

1 Motivation, history, rules

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

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

In the mean time the competition has been held four times: in 2004, 2005, 2006 and 2007. For the history of the first three instances we refer to [1]; this report focuses on the 2007 results. All details on the termination competition including past editions, rules and all results, are found on

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

The rules of the competition are as follows.

– All tools apply on all problems in the corresponding TPDB categories, all on the same machine. The required output of every tool is
  • "YES", followed by the text of a termination proof, or
  • "NO", followed by the text of a non-termination proof, or

  - anything else, interpreted as ”DON’T KNOW”.
- Execution of more than a fixed period for any tool on any termination problem causes a time-out, interpreted as ”DON’T KNOW”. In earlier in instances this fixed period was 1 minute, this year it was 2 minutes.
- All results are reported on-line, including generated proof text, and statistics about scores and running time.
- Any tool generating a wrong answer will be disqualified. This year this applied for a few tools: disqualification was implemented by removing the tool from the corresponding category.
- There are no formal rules and consequences of being a ”winner”, apart from the honor of having a high or the highest score in some category.

Due to submission of several new problems in TPDB, TPDB became very large and quite unbalanced. For instance, two directories were generated and submitted containing all string rewriting systems of a particular small size (at most three distinct symbols, at most 12 symbols in total) that could not be solved by any of the 2006 tools. These contain several hundreds of systems in total. It was decided that for directories in TPDB containing more than 128 systems, a random choice of exactly 128 systems was drawn for participation in the competition.

2 The categories and the tools

There were seven categories with the following participating tools. For further information on the tools, like the authors, we refer to the web page: simply click on the name of the tool.

- **Standard term rewriting.**
  This is the greatest category consisting of 975 termination problems, after restricting to 128 per directory. Participating tools were AProVE, Jambox, NTI and TTT2.
- **Context-sensitive term rewriting.**
  Here were two participating tools: AProVE and MU-Term. For other sub-categories of term-rewriting this year we did not apply the competition since all these sub-categories had at most one participant.
- **Certified termination.**
  This category is new this year: here apart from the YES/NO answer not a proof sketch is required but a fully certified proof. The list of termination problems was the same as for standard term rewriting. In this category every participant consists of a combination of a tool searching for the proof and an environment for certification. For all participants the final certification was done in Coq. The three participants were the combinations CiME+Coccinelle, TPA+Rainbow+CoLoR and TTT2+Rainbow+CoLoR.
- **Standard string rewriting.**
  This is the category with the greatest number of participants: AProVE, Jambox, Matchbox, MultumNonMulta, NTI, TORPA and TTT2.

- **Relative termination for string rewriting.**
  Participating tools were Jambox, Matchbox, MultumNonMulta and TORPA.
- **Logic programming.**
  Participating tools were AProVE, NTI, Polytool and TALP.
- **Functional programming.**
  The only participant was AProVE, so the category was run as a demonstration. Since we want to show that automated termination proving not only applies for rewriting but also for other programming paradigms, this category was installed. We emphasize that all termination problems in this category come directly (or by obvious transformations) from real Haskell code (Prelude and standard libraries).

3 The results

Detailed results including

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

are available from

`http://www.lri.fr/~marche/termination-competition/2007/`

In this section we restrict to the main observations, subdivided over the categories.

- **Standard term rewriting.** As in all earlier years in the category of standard term rewriting the tool AProVE performed the best with 723 termination proofs and 128 non-termination proofs. Second was TTT2 with 574 termination proofs and 69 non-termination proofs. Disappointing was the Jambox score of 330 and 16, respectively: this score is much worse than the score of the same tool in 2006. The new tool NTI is only on non-termination, and found 117 non-termination proofs.
- **Context-sensitive term rewriting.**
  Here MU-Term performed best with 68 termination proofs, while the other tool AProVE found 64 termination proofs. Non-termination proofs were not found in this category.
- **Certified termination.**
  Here the combination TPA+Rainbow+CoLoR performed best with 354 certified termination proofs. The other two combinations CiME+Coccinelle and TTT2+Rainbow+CoLoR found 317 and 289 certified termination proofs, respectively.
- **Standard string rewriting.**
  Here Matchbox performed best with 337 termination proofs and 65 non-termination proofs. Second and third were TORPA and TTT2 with 304 and

215 termination proofs, respectively. For non-termination NTI was second with 15 proofs; apart form NTI and Matchbox all other tools found at most 6 non-termination proofs.
– **Relative termination for string rewriting.**
In this small category Jambox performed best with 36 termination proofs; non-termination proofs were not found.
– **Logic programming.**
AProVE, Polytool and TALP respectively found 248, 210 and 170 termination proofs. These tools did not find non-termination proofs, while the remaining tool NTI only found non-termination proofs: 42 in total.
– **Functional programming.**
From the randomly drawn 128 Haskell programs in this category, AProVE succeeded in proving termination of 98 of them. Non-termination was not proved.

4 Conclusions

As an important objective for the future, last year we formulated formal correctness check of generated proofs. This year a giant leap into this direction has been made: three tools participated in the new category of certified termination, and succeeded in both finding and fully certifying termination proofs for a substantial number of rewrite systems, of which proving termination of several of them was considered to be very hard until a few years ago. All three participants can certify proofs based on dependency graph criteria. The main difference between them is on which kind of term orderings they can handle: Coccinelle handles recursive path orderings where CoLoR does not, on the other hand CoLoR supports matrix interpretations whereas Coccinelle does not. If both were supported, we could expect that more than 400 problems could be certified terminating, that is more or less 50% of the terminating problems of the TPDB. Apart from that, TPA also supports direct matrix interpretation proofs without dependency pairs. Of course, a future challenge is to support other termination criteria in order to be able to certify other kinds of termination proofs, with the ultimate goal of certification of all automatically found proofs.

In standard term rewriting the leading position of AProVE was continued. In contrast to last year when the matrix method was introduced, this year there was no similar break through. An impressive improvement in AProVE is their new technique for rewrite systems describing while-loops in which an induction argument is required, for example for TRS/Zantema06-while consisting of the three rules

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

which was presented as a challenge last year and now is solved by AProVE.

In standard string rewriting for a great number of small systems now termination could be proved for which all tools from 2006 failed, like SRS/Waldmann-jw1, consisting of the two rules

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

which was presented as an open problem last year and now was proved to be terminating both by Matchbox and TORPA. This is mainly due to the new techniques of arctic matrices and quasi-periodic interpretations. Being the only tool using arctic matrices, in this way Matchbox convincingly showed up to be the strongest tool in this category. The termination proof of SRS/Zantema-z090 found by TTT2 is the first automatically found termination proof of a string rewriting system having non-primitive-recursive reduction lengths.

Instead of only counting the number of YES scores and NO scores, this year we also experimented with weighted scores. The idea is that for a proof that was found by only a few other tools you get more points than if it was found by several other tools, with the highest number of points if it was found by no other tool. However, one can think of scenarios in which comparing total weighted scores has undesired effects. In the present competition for all categories the strongest tool remains the same, no matter whether total unweighted or total weighted scores are compared.

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

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

Details of the future format of the competition are under discussion.

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

References

1. Claude Marché and Hans Zantema. The Termination Competition. In F. Baader, editor, *Proceedings of the 18th International Conference on Rewriting Techniques and Applications (RTA)*, Lecture Notes in Computer Science. Springer, 2007.

Convergent Term Rewriting Systems for Inverse Computation of Injective Functions*

Naoki Nishida, Masahiko Sakai, and Terutoshi Kato

Graduate School of Information Science, Nagoya University
Furo-cho, Chikusa-ku, Nagoya 464-8603, Japan
`{nishida,sakai}@is.nagoya-u.ac.jp` `kato@sakabe.i.is.nagoya-u.ac.jp`

Abstract. This paper shows a sufficient syntactic condition for constructor TRSs whose inverse-computation CTRSs generated by Nishida, Sakai and Sakabe's inversion compiler are confluent and operationally terminating. By replacing the unraveling at the second phase of the compiler with Serbanuta and Rosu's transformation, we generate convergent TRSs for inverse computation of injective functions satisfying the sufficient condition.

1 Introduction

Given a program written in a functional language, an *inversion compiler* for the language generates another program written in the same language, so-called an *inverse(-computation) program* of the given program, that defines inverses of functions defined in the given one. Several inversion compilers for functional languages have been proposed [8, 2, 4, 5]. Inversion compilers are useful in automatically generating inverse programs that should have high reliability, such as *compression/decompression* tools and *shared key encryption/decryption* functions. Therefore, developing the compilers and theoretically proving their correctness are valuable.

The inversion compiler proposed in [4, 5] is applicable to constructor term rewriting systems. Given a term rewriting system (TRS), it first generates an inverse conditional TRS as an intermediate result, and then transforms the conditional TRS (CTRS) into a TRS that is equivalent with the CTRS with respect to inverse computation. The first phase of the compiler works as an *inversion* by itself. At the second phase, the compiler employs a variant of Ohlebusch's *unraveling* [6] that is a transformation from deterministic 3-CTRSs into TRSs. Unfortunately, inverse computation by the generated TRSs sometimes have several garbage normal forms that mean dead ends by wrong choices at inverse-computation branches. Note that given a normal form, it is decidable whether the normal form is a solution or a garbage. The cause of the occurrence of such normal forms is that unravelings generally produce TRSs approximating the corresponding CTRSs [6]. We face this problem even when we restrict functions

* This work is partly supported by MEXT. KAKENHI #17700009 and #18500011, and Kayamori Foundation of Informational Science Advancement.

to injective ones. For example, consider the following constructor TRS where $\mathsf{Snoc}(xs, y)$ produces the list obtained from xs by adding y as the last element:

$$R_1 = \begin{cases} \mathsf{Snoc}(\mathsf{nil}, y) \to \mathsf{cons}(y, \mathsf{nil}) \\ \mathsf{Snoc}(\mathsf{cons}(x, xs), y) \to \mathsf{cons}(x, \mathsf{Snoc}(xs, y)). \end{cases}$$

The compiler transforms R_1 into the following TRS:

$$\mathbb{U}(\mathcal{I}nv(R_1)) = \begin{cases} \mathsf{InvSnoc}(\mathsf{cons}(y, \mathsf{nil})) \to (\mathsf{nil}, y) \\ \mathsf{InvSnoc}(\mathsf{cons}(x, ys)) \to \mathsf{U}_1(\mathsf{InvSnoc}(ys), x) \\ \mathsf{U}_1((xs, y), x) \to (\mathsf{cons}(x, xs), y) \\ \mathsf{InvSnoc}(\mathsf{Snoc}(xs, y)) \to (xs, y). \end{cases}$$

Here, we abbreviate the tuple $\mathsf{tp}_n(t_1, \ldots, t_n)$ of n terms $t_1, \ldots, t_n$ to $(t_1, \ldots, t_n)$, and the list $\mathsf{cons}(t_1, \mathsf{cons}(t_2, \cdots, \mathsf{cons}(t_n, \mathsf{nil}) \cdots))$ to $[t_1, t_2, \ldots, t_n]$. The unique normal form of $\mathsf{Snoc}([\mathsf{A}, \mathsf{B}], \mathsf{C})$ is $[\mathsf{A}, \mathsf{B}, \mathsf{C}]$ but $\mathsf{InvSnoc}([\mathsf{A}, \mathsf{B}, \mathsf{C}])$ has two normal forms, a solution $([\mathsf{A}, \mathsf{B}], \mathsf{C})$ and a garbage $\mathsf{U}_1(\mathsf{U}_1(\mathsf{U}_1(\mathsf{InvSnoc}(\mathsf{nil}), \mathsf{C}), \mathsf{B}), \mathsf{A})$.

In this paper, we propose a method to generate convergent inverse TRSs of injective functions. More precisely, we show a sufficient syntactic condition for input constructor TRSs whose inverse CTRSs generated by the compiler are confluent and operationally terminating [3], and then we show that Serbanuta and Rosu's transformation [7] from CTRSs into TRSs generates convergent TRSs from the intermediate CTRSs of the compilers if the input TRSs satisfy the sufficient condition. Finally, we show an example of non-injective functions such that Serbanuta and Rosu's transformation does not preserve confluence of the inverse CTRSs, and another example that their transformation does not preserve operational termination of the inverse CTRSs. The full version of this paper containing the proofs of the theorems in this paper is available from the author's web site [1].

This paper follows the general notions of term rewriting [6].

2 Inversion Compiler for Constructor TRSs

In this section, using an example, we briefly explain the first phase of the inversion compiler for constructor TRSs [5], and some of its properties.

Consider the TRS R_1 again. By introducing a fresh variable for each subterm in the right-hand side of every rule that is rooted with a defined symbol, we obtain from R_1 the following CTRS:

$$R_1' = \begin{cases} \mathsf{Snoc}(\mathsf{nil}, y) \to \mathsf{cons}(y, \mathsf{nil}) \\ \mathsf{Snoc}(\mathsf{cons}(x, xs), y) \to \mathsf{cons}(x, ys) \Leftarrow \mathsf{Snoc}(xs, y) \to ys. \end{cases}$$

The first phase of the compiler, denoted by $\mathcal{I}nv$, exchanges the both sides of rules and conditions, reverses the order of conditional parts, applies inverse symbols,

[1] `http://www.sakabe.i.is.nagoya-u.ac.jp/~nishida/papers/`

removes $InvF(F(\ ))$, adds some special rules (necessary for partial functions [4, 5]), and then generates the following CTRS as an intermediate result:

$$\mathcal{I}nv(R_1) = \begin{cases} \mathsf{InvSnoc}(\mathsf{cons}(y, \mathsf{nil})) \to (\mathsf{nil}, y) \\ \mathsf{InvSnoc}(\mathsf{cons}(x, ys)) \to (\mathsf{cons}(x, xs), y) \Leftarrow \mathsf{InvSnoc}(ys) \to (xs, y) \\ \mathsf{InvSnoc}(\mathsf{Snoc}(xs, y)) \to (xs, y). \qquad\qquad \text{(special rule)} \end{cases}$$

Theorem 1 ([5]). *Let R be a convergent constructor TRS.*

– *$\mathcal{I}nv(R)$ is a non-erasing constructor deterministic CTRS.*
– *If R is non-erasing, then $\mathcal{I}nv(R)$ is a 3-CTRS.*
– *Let F be an n-ary defined symbol of R, and $t_1, \ldots, t_n, s$ be normal forms of R. $F(t_1, \ldots, t_n) \xrightarrow{*}_R s$ if and only if $InvF(s) \to_{\mathcal{I}nv(R)} (t_1, \ldots, t_n)$.*

3 Convergence of Inverse Systems for Injective Functions

In this section, we first give a sufficient syntactic condition for input constructor TRSs whose inverse CTRSs generated by $\mathcal{I}nv$ are confluent and operationally terminating. Then, we show that in this case, Serbanuta and Rosu's transformation [7] generates convergent inverse TRSs.

Definition 2. *Let R be a convergent constructor TRS. A defined symbol F of R is called* injective (with respect to normal forms) *if for all normal forms $s_1, \ldots, s_n$ and $t_1, \ldots, t_n$ of R, $F(s_1, \ldots, s_n) \downarrow_R F(t_1, \ldots, t_n)$ implies $s_i \equiv t_i$ for all i. The TRS R is called* injective (with respect to normal forms) *if all of its defined symbols are injective.*

Proposition 3. *Every injective TRS is non-erasing.*

Theorem 4. *Let R be a non-erasing constructor TRS. Suppose that for every rule $F(u_1, \ldots, u_n) \to r$ in R, if r is not a variable, then the root symbol of r does not depend [2] on F (either a constructor or a defined symbol not depending on F). Then, all of the following hold:*

(a) the CTRS $\mathcal{I}nv(R)$ is operationally terminating, and
(b) if R is injective, then the CTRS $\mathcal{I}nv(R)$ is confluent.

Note that the CTRS $\mathcal{I}nv(R)$ is convergent if R is injective, because operational termination implies termination (non-existence of infinite reduction sequences).

Serbanuta and Rosu's transformation [7], denoted by $\mathbb{T}$, can preserve convergence when transforming CTRSs into TRSs. Their transformation introduce the special constant $\bot$ and the unary symbol $\{\}$, and extends the arities of defined symbols of CTRSs S. More precisely, the new arity of an n-ary defined symbol F is $n + m$ where m is the number of conditions in F-rules in S. For a term t in the original signature, $\bar{t}$ denotes the term on the extended signature, that is obtained from t by adding $\bot$ to the extended arguments of defined symbols in t.

[2] We say that an n-ary symbol G of R *depends on a symbol* F if (G, F) is in the transitive closure of the relation $\{ (G', F') \mid G'(\cdots) \to C[F'(\cdots)] \in R \}$.

$$\begin{cases} \mathsf{InvSnoc}(\mathsf{cons}(y,\mathsf{nil}),z) \to \{(\mathsf{nil},y)\} \\ \mathsf{InvSnoc}(\mathsf{cons}(x,ys),\bot) \to \mathsf{InvSnoc}(\mathsf{cons}(x,ys),\{\mathsf{InvSnoc}(ys,\bot)\}) \\ \mathsf{InvSnoc}(\mathsf{cons}(x,ys),\{(xs,y)\}) \to \{(\mathsf{cons}(x,xs),y)\} \\ \mathsf{InvSnoc}(\mathsf{Snoc}(xs,y),z) \to \{(xs,y)\} \\ \{\{x\}\} \to \{x\}, \quad \mathsf{InvSnoc}(\{xs\},z) \to \{\mathsf{InvSnoc}(xs,\bot)\} \\ \mathsf{cons}(\{x\},xs) \to \{\mathsf{cons}(x,xs)\}, \quad (\{x\},y) \to \{(x,y)\}, \quad \mathsf{Snoc}(\{xs\},y) \to \{(xs,y)\} \\ \mathsf{cons}(x,\{xs\}) \to \{\mathsf{cons}(x,xs)\}, \quad (x,\{y\}) \to \{(x,y)\}, \quad \mathsf{Snoc}(xs,\{y\}) \to \{(xs,y)\}. \end{cases}$$

Fig. 1. Rewrite rules in $\mathbb{T}(\mathcal{I}nv(R_1))$.

Example 5. The CTRS $\mathcal{I}nv(R_1)$ is transformed by $\mathbb{T}$ into $\mathbb{T}(\mathcal{I}nv(R_1))$ in Fig. 1 [7]. The ground term $\mathsf{InvSnoc}(\,[\mathsf{A},\mathsf{B},\mathsf{C}]\,,\bot)$ $(= \overline{\mathsf{InvSnoc}(\,[\mathsf{A},\mathsf{B},\mathsf{C}]\,)})$ has the unique ground normal form $\{(\,[\mathsf{A},\mathsf{B}]\,,\mathsf{C})\}$ of $\mathbb{T}(\mathcal{I}nv(R_1))$.

Theorem 6 ([7]). *Let S be a deterministic 3-CTRS. If S is finite, ground confluent and operationally terminating on t, then $\mathbb{T}(S)$ is computationally equivalent with S, that is,*

- *$\mathbb{T}$ is sound and complete ($s \xrightarrow{*}_S t$ if and only if $\overline{s} \xrightarrow{*}_{\mathbb{T}(S)} \overline{t}$ for any $s,t \in \mathcal{T}(\mathcal{F})$), and*
- *$\mathbb{T}(S)$ is ground confluent and terminating on terms reachable from $\overline{t}$.*

Corollary 7. *Let R be an injective TRS that satisfies the assumption in Theorem 4, F be an n-ary defined symbol of R, and $t_1,\ldots,t_n,t$ be ground normal forms of R such that $F(t_1,\ldots,t_n) \xrightarrow{*}_R t$. Then, $\mathbb{T}(\mathcal{I}nv(R))$ is terminating on $InvF(t,\bot,\ldots,\bot)$ that has the unique ground normal form $\{(t_1,\ldots,t_n)\}$.*

Note that in the above corollary, $\overline{t} \equiv t$ because every functional symbol $F \in \mathcal{F}$ is a constructor of $\mathcal{I}nv(R)$. For every data list ts, R_1 is ground-convergent on $\mathsf{InvSnoc}(ts,\bot)$ and computationally equivalent with $\mathcal{I}nv(R_1)$ because R_1 satisfies the assumption in Theorem 4.

4 Discussion on Non-Injective Cases

Let's consider the following constructor TRS:

$$R_2 = \left\{\, \mathsf{D}(x) \to \mathsf{Add}(x,x), \quad \mathsf{Add}(0,y) \to y, \quad \mathsf{Add}(\mathsf{s}(x),y) \to \mathsf{s}(\mathsf{Add}(x,y)) \,\right\}.$$

The defined symbol D is injective and R_2 is convergent. However, R_2 is not injective because Add is not injective. The term $\mathsf{InvD}(\mathsf{s}^{2n}(0))$ $(= \overline{\mathsf{InvD}(\mathsf{s}^{2n}(0))})$ for some n (> 0) has more than two normal forms of $\mathbb{U}(\mathcal{I}nv(R_2))$ and $\mathbb{T}(\mathcal{I}nv(R_2))$, respectively, although the CTRS $\mathcal{I}nv(R_2)$ is confluent on $\mathsf{InvD}(\mathsf{s}^{2n}(0))$. Thus, similarly to the unraveling $\mathbb{U}$, Serbanuta and Rosu's transformation $\mathbb{T}$ cannot preserve confluence of CTRSs for inverses of non-injective TRSs.

Next, we give an example showing that $\mathbb{T}$ can generates non-terminating inverse TRSs for TRSs with erasing rules. When input TRSs have erasing rules, the generated CTRSs are not 3-CTRSs, that is, the CTRSs have extra variables in the right-hand side not in the conditional part. In such cases, *narrowing* can be used for inverse computation by the unraveled inverse CTRSs [4, 5]. Consider the following constructor TRS computing multiplication:

$$R_3 = R_2 \cup \left\{ \begin{array}{ll} \mathsf{Mul}(\mathsf{0}, y) \to \mathsf{0}, & \mathsf{Mul}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{s}(\mathsf{Add}(\mathsf{Mul}(x, \mathsf{s}(y)), y)) \\ \mathsf{Mul}(x, \mathsf{0}) \to \mathsf{0} & \end{array} \right\}.$$

Narrowing from $\mathsf{InvMul}(\mathsf{s}^n(\mathsf{0}), \bot, \bot)$ does not terminate on $\mathbb{T}(\mathcal{I}nv(R_3))$ while narrowing from $\mathsf{InvMul}(\mathsf{s}^n(\mathsf{0}))$ does on $\mathbb{U}(\mathcal{I}nv(R_3))$ and gives us desired solutions. The cause of non-termination is the added rules $c(x_1, \ldots, \{x_i\}, \ldots, x_n) \to \{c(x_1, \ldots, x_n)\}$ where c is a constructor. For example, we have the infinite narrowing sequence $\overline{\mathsf{InvMul}(\mathsf{0})} \equiv \mathsf{InvMul}(\mathsf{0}, \bot, \bot) \leadsto_{\mathbb{T}(\mathcal{I}nv(R_3))} \{(\mathsf{0}, z)\} \leadsto_{\mathbb{T}(\mathcal{I}nv(R_3))} \{\{(\mathsf{0}, z')\}\} \;_{\{z \mapsto \{z'\}\}}{\leadsto}_{\mathbb{T}(\mathcal{I}nv(R_3))} \cdots$ because $(x, \{y\}) \to \{(x, y)\} \in \mathbb{T}(\mathcal{I}nv(R_3))$. Note that $\mathbb{U}(\mathcal{I}nv(R_3))$ is terminating on $\mathsf{InvMul}(\mathsf{s}^n(\mathsf{0}))$ with respect to narrowing. Therefore, it can be said that $\mathbb{T}$ always generates non-terminating inverse-TRSs for TRSs with erasing-rules.

In conclusion, comparing with the unraveling, Serbanuta and Rosu's transformation is more effective for injective TRSs at the second phase of the inversion compiler, incomparable for non-injective and non-erasing TRSs, and less effective for the remaining case. In the last case, their transformation should not be employed at the second phase of the inversion compiler.

The class of injective TRSs satisfying the sufficient condition in this paper is incomparable with that of injective TRSs for which Dershowitz and Mitra's Inversion Algorithm [1] terminates. There is a TRS whose inverse TRS is convergent, and for which termination of the algorithm is not guaranteed. As a related work, Kawabe and Glück proposed a transformation based on LR-parsing [2], in order to generate convergent inverses of injective functions in a functional languages. Comparison of our method with theirs is one of future works.

References

1. Dershowitz, N., Mitra, S.: Jeopardy. In: Proceedings of RTA'99. Volume 1631 of LNCS, Springer (1999) 16–29
2. Kawabe, M., Glück, R.: The program inverter lrinv and its structure. In: Proceedings of PADL'05. Volume 3350 of LNCS, Springer (2005) 219–234
3. Lucas, S., Marché, C., Meseguer, J.: Operational termination of conditional term rewriting systems. Information Processing Letters **95**(4) (2005) 446–453
4. Nishida, N., Sakai, M., Sakabe, T.: Partial inversion of constructor term rewriting systems. In: Proceedings of RTA'05. Volume 3467 of LNCS, Springer (2005) 264–278
5. Nishida, N., Sakai, M., Sakabe, T.: Generation of inverse computation programs of constructor term rewriting systems. The IEICE Transactions on Information and Systems **J88-D-I**(8) (2005) 1171–1183 (in Japanese)
6. Ohlebusch, E.: Advanced Topics in Term Rewriting. Springer-Verlag (2002)
7. Serbanuta, T. F., Rosu, G.: Computationally equivalent elimination of conditions. In: Proceedings of RTA'06. Volume 4098 of LNCS, Springer (2006) 19–34
8. Romanenko, A.: Inversion and metacomputation. In: Proceedings of PEPM'91. Volume 26 of SIGPLAN Notices, ACM Press (1991) 12–22

On Proving and Characterizing Operational Termination of Deterministic Conditional Rewrite Systems

Felix Schernhammer and Bernhard Gramlich
TU Wien, Austria, {felixs,gramlich}@logic.at

1 Introduction and Overview

Conditional term rewriting systems (CTRSs) are a natural extension of unconditional such systems (TRSs) allowing rules to be guarded by conditions. Conditional rules tend to be very intuitive and easy to formulate and are therefore used in several programming and specification languages, such as Maude or ELAN. Besides, the particular class of *deterministic* (oriented) CTRSs (DCTRSs) has been used for instance in proofs of termination of (well-moded) logic programs, [2]. Recently, the notion of *operational termination* of DCTRSs was defined in [3], which guarantees finite reductions in existing rewrite engines. In [8], based on the idea of *unravelings* of [5], a transformation from DCTRSs into TRSs was proposed, such that termination of the transformed TRS implies *quasi-reductivity* of the DCTRS. In this extended abstract [1] we provide an alternative definition of *quasi-reductivity*, which allows us to prove the property for strictly more DCTRSs than Ohlebusch's transformation was able to handle, while preserving the important implications of quasi-reductivity. The key concept used is context-sensitive rewriting [4]. We define a transformation from DCTRSs into context-sensitive (unconditional) TRSs such that termination of the transformed context-sensitive TRS implies *context-sensitive quasi-reductivity* of the DCTRS. Furthermore we relate the latter notion to other known ones and give equivalent characterizations of operational termination.

2 Preliminaries

We assume familiarity with the basic concepts and notations of (context-free and context-sensitive) term rewriting (cf. e.g. [1], [4]). We are concerned with *oriented* 3-TRSs. Such systems consist of rules of the form $l \to r \Leftarrow c$, with c of the form $s_1 \to^* t_1, ..., s_n \to^* t_n$ such that l is not a variable and $Var(r) \subseteq Var(l) \cup Var(c)$. The conditional rewrite relation induced by a CTRS $\mathcal{R}$ is inductively defined as $\to_{\mathcal{R}} = \bigcup_{i \geq 0} R_i$ where $R_0 = \emptyset$ and $R_{i+1} = \{\sigma l \to \sigma r \mid l \to r \Leftarrow s_1 \to^* t_1, ..., s_n \to^* t_n \in \mathcal{R} \wedge \sigma s_i \to^*_{R_i} \sigma t_i$ for all $1 \leq i \leq n\}$. A deterministic CTRS (DCTRS) is an oriented 3-CTRS where for each rule $l \to r \Leftarrow s_1 \to^* t_1, ..., s_n \to^* t_n$ it holds that $Var(s_i) \subseteq l \cup \bigcup_{j=1}^{i-1} Var(t_j)$.

A DCTRS (Σ, R) is called *quasi-reductive*, cf. [8] [2], if there exists an extension Σ' of Σ and a well-founded partial order $\succ$ on $\mathcal{T}(\Sigma', V)$, which is monotonic, i.e., closed under contexts, such that for every rule $l \to r \Leftarrow s_1 \to^* t_1, ..., s_n \to^* t_n \in R$, every $\sigma \colon V \to \mathcal{T}(\Sigma', V)$ and every $i \in \{0, ..., n-1\}$:

– If $\sigma s_j \succeq \sigma t_j$ for every $1 \leq j \leq i$, then $\sigma l \succ_{st} \sigma s_{i+1}$.

[1] A long version of this paper with complete proofs is available at http://www.logic.at/people/schernhammer/papers/wst07-long.pdf.

– If $\sigma s_j \succeq \sigma t_j$ for every $1 \leq j \leq n$, then $\sigma l \succ \sigma r$.

Here $\succ_{st} = (\succ \cup \rhd)^+$ ($\rhd$ denotes the proper subterm relation).

A DCTRS $\mathcal{R} = (\Sigma, R)$ is *quasi-decreasing* [8] if there is a well-founded partial ordering $\succ$ on $\mathcal{T}(\Sigma, V)$, such that $\rightarrow_{\mathcal{R}} \subseteq \succ$; $\succ = \succ_{st}$; and for every rule $l \rightarrow r \Leftarrow s_1 \rightarrow^* t_1, ..., s_n \rightarrow^* t_n \in R$, every substitution σ and every $i \in \{0, ..., n-1\}$ it holds that $\sigma s_j \rightarrow^*_{\mathcal{R}} \sigma t_j$ for all $j \in \{1, ..., i\}$ implies $\sigma l \succ \sigma s_{i+1}$.

In [3] a notion of *operational termination* of (D)CTRSs is defined via the absence of infinite well-formed trees in a certain logical inference system. In the case of DCTRSs, this notion is shown to be equivalent to quasi-decreasingness [3]. The relation between the latter notions is ([8], [3]): Quasi-reductivity $\Rightarrow$ quasi-decreasingness $\Leftrightarrow$ operational termination $\Rightarrow$ effective termination.

3 Context-sensitive quasi-reductivity

Definition 1. *A DCTRS $\mathcal{R}$ ($\mathcal{R} = (\Sigma, R)$) is called* context-sensitively quasi-reductive *(*cs-quasi-reductive*) if there is an extension of the signature Σ' ($\Sigma' \supseteq \Sigma$), a replacement μ (s.t. $\mu(f) = \{1, ..., ar(f)\}$ for all $f \in \Sigma$) and a μ-monotonic, well-founded partial order $\succ_\mu$ on $\mathcal{T}(\Sigma', V)$ satisfying for every rule $l \rightarrow r \Leftarrow s_1 \rightarrow^* t_1, ..., s_n \rightarrow^* t_n$, every $\sigma\colon V \rightarrow \mathcal{T}(\Sigma, V)$ and every $i \in \{0, ..., n-1\}$:* [2]

– If $\sigma s_j \succeq_\mu \sigma t_j$ for every $1 \leq j \leq i$ then $\sigma s_{i+1} \prec^{st}_\mu \sigma l$.

– If $\sigma s_j \succeq_\mu \sigma t_j$ for every $1 \leq j \leq n$ then $\sigma r \prec_\mu \sigma l$.

The ordering $\prec^{st}_\mu$ is defined as $(\prec_\mu \cup \lhd_\mu)^+$ where $t \lhd_\mu s$ if and only if s is a proper subterm of t at some position $p \in Pos^\mu(t)$.

Lemma 1. *Quasi-reductivity implies cs-quasi-reductivity which in turn implies quasi-decreasingness.*

We will define a transformation from DCTRSs into CSRSs, such that μ-termination of the transformed CSRS implies cs-quasi-reductivity of the original DCTRS. The transformation is actually a variant of the one in [8].

Definition 2. [8] *Let $\mathcal{R} = (\Sigma, R)$ be a DCTRS. For every rule $\alpha : l \rightarrow r \Leftarrow s_1 \rightarrow^* t_1, ..., s_n \rightarrow^* t_n$ we use n new functions symbols U^α_i ($i \in \{1, ..., n\}$). Then α is transformed into a set of unconditional rules in the following way:*

$$\begin{aligned} l &\rightarrow U^\alpha_1(s_1, Var(l)) \\ U^\alpha_1(t_1, Var(l)) &\rightarrow U^\alpha_2(s_2, Var(l), \mathcal{E}Var(t_1)) \\ &\vdots \\ U^\alpha_n(t_n, Var(l), \mathcal{E}Var(t_1), ..., \mathcal{E}Var(t_{n-1})) &\rightarrow r \end{aligned}$$

Here $Var(s)$ denotes the sequence of variables in a term s rather than the set. The set $\mathcal{E}Var(t_i)$ is $Var(t_i) \setminus (Var(l) \cup \bigcup_{j=1}^{i-1} Var(t_j))$. Again, abusing notation, by $\mathcal{E}Var(t_i)$ we mean an arbitrary but fixed sequence of the variables in the set $\mathcal{E}Var(t_i)$. Any unconditional rule of $\mathcal{R}$ is transformed into itself (this degenerate case is missing in [8, Def. 7.2.48]). The transformed system $U(\mathcal{R}) = (U(\Sigma), U(R))$ is obtained by transforming each rule of $\mathcal{R}$ where $U(\Sigma)$ is Σ extended by all new function symbols.

[2] Note that – in contrast to the definition of quasi-reductivity – the substitution maps variables only into terms over the original signature. This restriction is crucial for some results (cf. Theorem 2 and Corollary 1).

This transformation is sound w.r.t. quasi-reductivity, i.e., whenever the transformed system $U(\mathcal{R})$ is terminating, the original DCTRS $\mathcal{R}$ is *quasi-reductive* [8]. The transformation is not complete in this respect, though.

Example 1. [**5**] Consider the deterministic CTRS $\mathcal{R} = (\Sigma, R)$ given by

$$\begin{array}{rlll}
a \to c & & b \to c & c \to e \\
a \to d & & b \to d & c \to l \\
k \to l & & k \to m & d \to m \\
h(x,x) \to g(x,x,f(k)) & & g(d,x,x) \to A & \alpha : f(x) \to x \Leftarrow x \to^* e \\
A \to h(f(a), f(b)). & & &
\end{array}$$

The system $U(\mathcal{R}) = (U(\Sigma), U(R))$ is given by $U(\Sigma) = \Sigma \cup \{U_1^\alpha\}$ and $U(R) = R$ except that rule α is replaced by the rules $f(x) \to U_1^\alpha(x,x)$ and $U_1^\alpha(e,x) \to x$. $\mathcal{R}$ is quasi-reductive, nevertheless $U(\mathcal{R})$ is non-terminating ([8]).

Roughly speaking, the problem in the latter example is that subterms at the second position of U_1^α are reduced, which are actually only supposed to "store" the variable bindings for future rewrite steps. These reductions can be avoided by using context-sensitivity.

Definition 3. *Let $\mathcal{R} = (\Sigma, R)$ be a deterministic conditional term rewriting system. The context-sensitive rewrite system $U_{cs}(\mathcal{R})$ uses the same signature and the same rules as $U(R)$. Additionally, we use a replacement map $\mu_{U_{cs}(\mathcal{R})}$ with $\mu_{U_{cs}(\mathcal{R})}(f) = \{1\}$ if $f \in U_{cs}(\Sigma) \backslash \Sigma$ and $\mu_{U_{cs}(\mathcal{R})}(f) = \{1, ..., ar(f)\}$ if $f \in \Sigma$.*

Proposition 1 (simulation completeness). *Let $\mathcal{R} = (\Sigma, R)$ be a DCTRS and $U_{cs}(\mathcal{R})$ its transformed CSRS. For every $s, t \in \mathcal{T}(\Sigma, V)$ we have: If $s \to_{\mathcal{R}} t$, then $s \to^+_{U_{cs}(\mathcal{R})} t$.*

Proposition 2 (simulation soundness). *Let $\mathcal{R} = (\Sigma, R)$ be a DCTRS and let $U_{cs}(\mathcal{R}) = (U(\Sigma), U(R))$ be its transformed CSRS. For every $s, t \in \mathcal{T}(\Sigma, V)$ we have: If $s \to^+_{U_{cs}(\mathcal{R})} t$, then $s \to^+_{\mathcal{R}} t$.*

Furthermore, $\mu_{U_{cs}(\mathcal{R})}$-termination of the transformed system $U_{cs}(\mathcal{R})$ implies cs-quasi-reductivity of the original DCTRS $\mathcal{R}$, cf. the more general Theorem 2 below.

Example 2. Consider the DCTRS $\mathcal{R}$ of Example 1. The transformed system $U_{cs}(\mathcal{R})$ (which is identical to $U(\mathcal{R})$, except for the fact that an additional replacement map is used) is $\mu_{U_{cs}(\mathcal{R})}$-terminating. To show this one can use induction on the term depth.

Unfortunately, and interestingly, cs-quasi-reductivity of a DCTRS $\mathcal{R}$ does not imply $\mu_{U_{cs}(\mathcal{R})}$-termination of $U_{cs}(\mathcal{R})$, cf. [8, Ex. 7.2.51]. Yet, we do get $\mu_{U_{cs}(\mathcal{R})}$-termination of $U_{cs}(\mathcal{R})$ on original terms, even for quasi-decreasing systems. Conversely, cs-quasi-reductivity follows from termination of the transformed system on original terms.

Theorem 1. *Let $\mathcal{R} = (\Sigma, R)$ be a DCTRS. If $\mathcal{R}$ is quasi-decreasing, then $U_{cs}(\mathcal{R})$ is $\mu_{U_{cs}(\mathcal{R})}$-terminating on $\mathcal{T}(\Sigma, V)$.*

Theorem 2. *Let $\mathcal{R} = (\Sigma, R)$ be a DCTRS. If $U_{cs}(\mathcal{R})$ is $\mu_{U_{cs}(\mathcal{R})}$-terminating on $\mathcal{T}(\Sigma, V)$, then $\mathcal{R}$ is cs-quasi-reductive.*

Corollary 1. *Let $\mathcal{R} = (\Sigma, R)$ be a DCTRS. The following properties of $\mathcal{R}$ are equivalent: $\mu_{U_{cs}(\mathcal{R})}$-termination of $U_{cs}(\mathcal{R})$ on $\mathcal{T}(\Sigma, V)$, cs-quasi-reductivity, quasi-decreasingness, and operational termination.*

From a practical point of view it remains unclear whether the restricted proof task of showing $\mu_{U_{cs}(\mathcal{R})}$-termination of $U_{cs}(\mathcal{R})$ on $\mathcal{T}(\Sigma, V)$ is really easier than proving $\mu_{U_{cs}(\mathcal{R})}$-termination of $U_{cs}(\mathcal{R})$ on all terms over $\Sigma' = U(\Sigma)$.

Furthermore, it is currently open to what degree termination proofs of DCTRSs based on cs-quasi-reductivity are in practice more successful than those based on the previous notion of quasi-reductivity. This seems to depend strongly on the power of existing termination tools for context-sensitive systems.

4 Related Work and Discussion

A very similar modification of the transformation in [8] was proposed by [7]. However, there, besides context-sensitivity, the authors additionally impose a *membership condition* on the rewrite relation of the transformed CSRS. In [6] a further refinement of the transformation of [7] is presented, which is able to reduce the number of U symbols in the transformed system in some cases.

Our notion of cs-quasi-reductivity provides a new sufficient (in fact, equivalent) criterion for operational termination. Furthermore, cs-quasi-reductivity can be verified by proving termination of the resulting CSRS (on original terms). We have shown that our new transformation yields operational termination of strictly more DCTRSs than Ohlebusch's context-free transformation. However, we could not automatically, i.e., with termination tools, verify operational termination of the DCTRS of Example 1. Thus more powerful termination tools and/or features appear to be necessary.

References

[1] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.

[2] H. Ganzinger and U. Waldmann. Termination proofs of well-moded logic programs via conditional rewrite systems. In *Proc. CTRS'92, Pont-à-Mousson, France, July 1992*, pp. 430–437, LNCS 656, Springer, 1993.

[3] S. Lucas, C. Marchè, and J. Meseguer. Operational termination of conditional term rewriting systems. *Inf. Process. Lett.*, 95(4):446–453, 2005.

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

[5] M. Marchiori. Unravelings and ultra-properties. In *Proc. ALP'96, Aachen*, LNCS 1139, pp. 107–121. Springer, September 1996.

[6] N. Nishida, T. Mizutani, and M. Sakai. Transformation for refining unravled conditional term rewriting systems. In S. Antoy, ed., *Prelim. Proc. WRS'06, Seattle, Washington, USA, August 11, 2006*, pp. 34–48, August 2006.

[7] N. Nishida, M. Sakai, and T. Sakabe. Partial inversion of constructor term rewriting systems. In J. Giesl, ed., *Proc. RTA'05, Nara, Japan, April 19-21, 2005*, LNCS 3467, pp. 264–278. Springer, 2005.

[8] Enno Ohlebusch. *Advanced topics in term rewriting.* Springer, 2002.

A General Technique for Analyzing Termination in Symmetric Proof Calculi

Daniel J. Dougherty[1], Silvia Ghilezan[2] and Pierre Lescanne[3]

[1] Worcester Polytechnic Institute, USA, dd@cs.wpi.edu
[2] Faculty of Engineering, University of Novi Sad, Serbia, gsilvia@uns.ns.ac.yu
[3] École Normale Supérieure de Lyon, France, Pierre.Lescanne@ens-lyon.fr

Abstract. Proof-term calculi expressing a computational interpretation of classical logic serve as tools for extracting the constructive content of classical proofs and at the same time can be seen as pure programming languages with explicit representation of control. The inherent symmetry in the underlying logic presents technical challenges to the study of the reduction properties of these systems. We explore the use of intersection types for these calculi, note some of the challenges inherent in applying intersection types in a symmetric calculus, and show how to overcome these difficulties. The approach is applicable to a variety of systems under active investigation in the current literature; we apply our techniques in a specific case study: characterizing termination in the symmetric lambda-calculus of Barbanera and Berardi.

1 Introduction

The Curry-Howard correspondence [10] expresses a fundamental connection between logic and computation. In its traditional form, terms in the ‘l-calculus encode proofs in intuitionistic natural deduction and proofs serve as typing derivations for the terms. Griffin extended the Curry-Howard correspondence to classical logic in his seminal 1990 POPL paper [8], by observing that classical tautologies suggest typings for certain control operators. This initiated a vigorous line of research: on the one hand classical calculi can be seen as pure programming languages with explicit representations of control, while at the same time terms can be tools for extracting the constructive content of classical proofs [13, 2]. In particular the $\overline{\lambda}\mu$ calculus of Parigot [15] has been the foundation of a number of investigations [16, 6, 14, 3, 1] into the relationship between classical logic and theories of control in programming languages.

In contrast to natural deduction proof systems (upon which Parigot's $\overline{\lambda}\mu$, for example, is based) sequent calculi exhibit inherent symmetries in proof structures. There are several term calculi based on sequent calculus, in which reduction corresponds to cut elimination. Examples include [9, 19, 4, 20, 12]. This symmetry is appealing in its way, but it actually creates considerable technical difficulty in analyzing the reduction behavior of these calculi. The bedrock traditional technique of reducibility makes essential use of fact that function types are "higher" in a natural sense than argument types, permitting semantic definitions to proceed by induction on types. The symmetry in classical calculi blocks a straightforward adaptation of the traditional reducibility technique.

The Symmetric Lambda Calculus (here denoted λ^{sym}) of Barbanera and Berardi [2] is an elegant calculus designed with the goal of extracting constructive content from classical proofs (actually what we call λ^{sym} here is the propositional version of their calculus, in [2] the system is extended to Peano arithmetic). Barbanera and Berardi proved termination for their calculus using a fundamental insight called the "symmetric candidates" technique. (A completely different approach is the arithmetical proof of termination in a symmetric $\overline{\lambda}\mu$ calculus by David and Nour [5].)

The use of symmetric candidates is a robust technique that applies in a variety of settings [19, 17]. But a fundamental tool for deep analysis of the reduction properties of λ-calculus, as well as semantic investigations, is the extension of simple typing to intersection types — and *the adaptation of the symmetric candidates technique to the intersection-types setting is not straightforward.* Such an adaptation is the topic of this paper. We analyze the problems that arise and show how to overcome them. Our technique applies generally to all of the symmetric proof-calculi we have investigated, including the $\overline{\lambda}\mu\widetilde{\mu}$-calculus of Curien and Herbelin [4, 7], and the dual calculus of Wadler [20]. For concreteness here we outline the treatment for λ^{sym}, chosen because it is syntactically most familiar and the issues with duality can be seen most clearly. We believe that the presentation here may also clarify some of the subtleties of Barbanera and Berardi's proof even for simply-typed λ^{sym}.

The technical contribution of this paper can be summarized as follows. The key to the symmetric candidates technique is to interpret types in certain families of *saturated* sets: these are sets which are, roughly speaking, closed under inverse β-reduction. Using a clever fixed-point computation Barbanera and Berardi [2] define a family of saturated sets appropriate for interpreting simply-typed terms. But when we wish to consider intersection types we face the following obstacle. In standard semantics of intersection types, the interpretation of an intersection type $(A \cap B)$ is simply the intersection of the interpretations of A and B. But in general *intersections of saturated sets are not saturated.* So it is not clear how to interpret a type of the form $(A \cap B)$.

The solution we adopt is the following. We take seriously the fact that saturated sets are fixed points of a monotonic operator. And as is well-known, a monotonic operator has *a complete lattice* of fixed points. So we take the interpretation $[\![(A \cap B)]\!]$ of an intersection type to be the meet $[\![A]\!] \curlywedge [\![B]\!]$ in this lattice.

A consequence of the fact that the interpretation of an intersection is not the intersection of interpretations is that the standard typing rule for intersection-introduction is not sound. So our type system has an intersection-elimination rule only. As it turns out this is not a problem since intersection-introduction is not needed for characterizing termination.

A natural question arises: in the absence of intersection-introduction, how do any terms ever receive a type which is an intersection? The answer is: by double-negation elimination! Referring to the type system of Definition 2, if x is declared with type $(A \cap B)^{\perp}$ in a typing of $c : \perp$, then $\lambda x.c$ will be typed with type $(A \cap B)$.

2 Intersection types for the λ^{sym}-calculus

The syntax of λ^{sym} expressions is given by the following:

$$t := x \mid \langle t_1, t_2 \rangle \mid \sigma_1(t), \mid \sigma_2(t), \mid \lambda x.c \mid (t_1 * t_2)$$

We depart from [2] in that we treat the operator $*$ is syntactically commutative, that is, we consider $(t_1 * t_2)$ and $(t_2 * t_1)$ to be the same term.

The reduction rules of the calculus are

$$(\lambda x.b * a) \to b[x \leftarrow a] \qquad (\langle t_1, t_2 \rangle * \sigma_i(u)) \to \langle t_i, u \rangle \qquad \lambda x.(b * x) \to b \text{ if } x \text{ not free in } b$$

The system of [2] had an "η-reduction" rule as well as a reduction (labeled *Triv* there) which allowed replacing a term by one of its closed subterms under certain circumstances. We have omitted these reductions here for convenience (only): adding it would not violate any of our results and would complicate the presentation.

Definition 1. *The set* $\mathbb{T}$ *of* raw types *is generated from an infinite set* $TVar$ *of type-variables as follows.*

$$\mathbb{T} ::= TVar \mid \mathbb{T} \wedge \mathbb{T} \mid \mathbb{T} \vee \mathbb{T} \mid \mathbb{T}^{\perp} \mid \mathbb{T} \cap \mathbb{T}$$

We consider raw types modulo the equations

$$A^{\perp\perp} = A \qquad (A \wedge B)^{\perp} = A^{\perp} \vee B^{\perp} \qquad (A \vee B)^{\perp} = A^{\perp} \wedge B^{\perp}$$

A *type* is either an equivalence class modulo these equations or the special type $\perp$. Note that by orienting the equations above left-to-right each type has a normal form, in which the $(\cdot)^{\perp}$ operator is applied only to type variables or intersections. It is then easy to see that each type other than $\perp$ is uniquely of one of the following forms (here τ is a type variable).

$$\tau \quad \tau^{\perp} \quad (A_1 \wedge \cdots \wedge A_n) \quad (A_1 \vee \cdots \vee A_n) \quad (A_1 \cap \cdots \cap A_n) \quad (A_1 \cap \cdots \cap A_n)^{\perp}$$

Definition 2 (Typing rules of the system $\mathcal{B}$). *The type assignment system $\mathcal{B}$ is given by the following typing rules.*

$$\frac{}{\Sigma,\ x:(T_1\cap\cdots\cap T_k)\ \vdash\ x:T_i}\,(ax)$$

$$\frac{\Sigma\ \vdash\ t_1:A_1 \qquad \Sigma\ \vdash\ t_2:A_2}{\Sigma\ \vdash\ \langle t_1,t_2\rangle : A_1\wedge A_2}\,(\wedge) \qquad\qquad \frac{\Sigma\ \vdash\ t:A_i}{\Sigma\ \vdash\ \sigma_i(t):A_1\vee A_2}\,(\vee)$$

$$\frac{\Sigma,\ x:A\ \vdash\ c:\perp}{\Sigma\ \vdash\ \lambda x.c\ :\ A^{\perp}}\,(\perp) \qquad\qquad \frac{\Sigma\ \vdash\ p:A \qquad \Sigma\ \vdash\ q:A^{\perp}}{\Sigma\ \vdash\ (p\ *\ q):\perp}\,(cut)$$

Theorem 3 (Main result). *A λ^{sym} term is terminating if and only if it is typable in $\mathcal{B}$.*

The direction "every terminating term is typable" follows the standard pattern from traditional λ-calculus. The nuance to be acknowledged is that, for the purpose of establishing this result, the standard intersection-introduction typing rule is not needed. This is crucial for our approach since, as noted in the introduction, this rule seems to be an obstacle to the other direction of the result, "every typable term is terminating."

The remainder of the paper is an outline of the proof that every typable term is terminating, with an emphasis on those aspects of the argument requiring novel treatment beyond the proof for traditional λ-calculus.

Definition 4. *A* pair *is given by two sets of terms X and Y, each of which is non-empty.*
The pair $\{X_1,X_2\}$ is stable *if for every $r\in X_1$ and every $e\in E$, the command $(r\ *\ e)$ is terminating.*

For example, the pair $\{Vars,Vars\}$ is stable. Since the sets in a pair are non-empty, any stable pair consists of terminating expressions. We write pairs as $\{X,Y\}$ but we stress that this is a multiset, that is, it may be that X and Y are the same set.

The following technical condition will be crucial to the use of pairs to interpret types (it is this condition which makes the Type Soundness Theorem go through, specifically the cases of typing a λ expression).

Definition 5. *A pair $\{X_0,X_1\}$ is* saturated *if for each i, whenever $\lambda x.c$ satisfies: $\forall e\in X_i, c[x\leftarrow e]$ is terminating, then $\lambda x.c\in X_{1-i}$.*

We can always expand a pair to be saturated. It is more delicate to expand a stable pair to be saturated and remain stable. The development below achieves this.

Definition 6. *An expression is* simple *if it is not a 'l-abstraction; a set X is simple if each term in X is simple.*

Definition 7. *Define the map $\Phi: 2^{\Lambda}\to 2^{\Lambda}$ by*

$$\begin{aligned}\Phi(X) = {} & \{e \mid e \textit{ is of the form } \lambda x.c \textit{ and } \forall r\in X, c[x\leftarrow r] \textit{ is terminating}\}\\ & \cup\{e \mid e \textit{ is simple and } \forall r\in X, (r\ *\ e) \textit{ is terminating}\}\end{aligned}$$

Note that if $X\neq\emptyset$ then $\Phi(X)$ is a set of terminating terms. Also, if $X\subseteq SN$ then all variables are in $\Phi(X)$.

It is easy to see that Φ is antimonotone. So the map $(\Phi\circ\Phi)=\Phi^2$ is monotone. By the Knaster-Tarski fixed point theorem [11, 18] Φ^2 has a complete lattice of fixed points, ordered by set inclusion.

Definition 8. *A pair $\{X,Y\}$ is a* mutual fixed point *for Φ if $\Phi(X)=Y$ and $\Phi(Y)=X$.*

It is easy to see that if $\{X,Y\}$ is a mutual fixed point for Φ then X is a fixed point for Φ^2.

Lemma 9. *Suppose $\{X,Y\}$ is a mutual fixed point for Φ. Then $\{X,Y\}$ is stable and saturated.*

So now our task is to show how to make mutual fixed points which have the right structure for interpreting types. The strategy for defining saturated pairs for types is slightly different depending on whether the type to be interpreted is (i) an arrow-type or its dual or (ii) an intersection or its dual. In the former case we need to establish the that the operator Φ^2 is inflationary on strongly normalizing simple sets.

Lemma 10. *If X is a simple set of terminating terms then $X \subseteq \Phi^2(X)$.*

As is well-known, when $\mathcal{G}$ is a monotone operator on a complete lattice of sets and X satisfies $X \subseteq \mathcal{G}(X)$ then $\mathcal{G}$ has a fixed point containing X. This, in light of Lemma 10, justifies the following definition.

Definition 11. *If X is a simple set of terminating terms let $X^{\uparrow}$ be the least fixed point of Φ^2 with the property that $X \subseteq X^{\uparrow}$.*

We note the following facts. If X is a simple set of terminating terms then $\{X^{\uparrow}, \Phi(X^{\uparrow})\}$ is a mutual fixed point of Φ, with $X \subseteq X^{\uparrow}$. If $\{X,Y\}$ is a stable pair and Y is a set of simple terms then $Y \subseteq \Phi(X^{\uparrow})$.

In Definition 13 we will use the above construction to interpret types which are not intersections (or their duals). When the types we want to interpret are intersections, or types of the form $(T_1 \cap \cdots \cap T_k)^{\circ}$ the above construction does not work. The essential problem is that the intersection of saturated pairs does not in general yield a saturated pair. This means that the interpretation of an intersection type $(A \cap B)$ will not be the intersection of the interpretations of A and B. But the collection of fixed points of Φ^2 carries its own lattice structure under inclusion, and this is all we require to interpret intersection types.

Definition 12. *Let Fix_{Φ^2} be the set of fixed points of the operator Φ^2. If $R_1, \ldots, R_k$ are fixed points of Φ^2, let $(R_1 \curlywedge \ldots \curlywedge R_k)$ denote the meet of these elements in the lattice Fix_{Φ^2}.*

The set of objects of the lattice Fix_{Φ^2} is a subset of the set 2^{Λ}. Since this lattice is ordered by set inclusion, we have $(R_1 \curlywedge \ldots \curlywedge R_k) \subseteq R_i$ for each i. Since $\cap$ is the greatest lower bound operator in 2^{Λ}, $(R_1 \curlywedge \ldots \curlywedge R_k) \subseteq (R_1 \cap \cdots \cap R_k)$.

We stress that $(R_1 \curlywedge \ldots \curlywedge R_k)$ is a fixed point of Φ^2 and so the pair $\{(R_1 \curlywedge \ldots \curlywedge R_k),\ \Phi(R_1 \curlywedge \ldots \curlywedge R_k)\}$is a mutual fixed point of Φ.

Definition 13 (Interpretation of types). *For each type T we define the set $[\![T]\!]$.*

1. *When T is $\perp$ then $[\![T]\!]$ is the set of terminating terms.*
2. *When T is a type variable we set R to be the set of term variables, then construct the pair $(R^{\uparrow}, \Phi(R^{\uparrow})$. We then take $[\![T]\!]$ to be $R^{\uparrow}$ and $[\![T^{\perp}]\!]$ to be $\Phi(R^{\uparrow})$.*
3. *Suppose T is $(A_1 \wedge A_2)$. Set R to be $\{\langle t_1, t_2 \rangle \mid t_i \in [\![A_i]\!],\ i = 1,2\}$. We then take $[\![T]\!]$ to be $(R^{\uparrow})$ and $[\![T^{\perp}]\!] = [\![{A_1}^{\perp} \vee {A_2}^{\perp}]\!]$ to be $\Phi(R^{\uparrow})$.*
4. *When T is $(A_1 \cap A_2 \cdots \cap A_n)$, $\quad n \geq 2$, we take $[\![T]\!]$ to be $([\![A_1]\!] \curlywedge \ldots \curlywedge [\![A_n]\!])$ and then take $[\![T^{\perp}]\!]$ to be $\Phi([\![T]\!])$.*

Note that the interpretation $[\![A_1 \vee A_2]\!]$ of a disjunction-type is determined in part 3 above since any type $B_1 \vee B_2$ is the dual of ${B_1}^{\perp} \wedge {B_2}^{\perp}$. Also note that by definition, for each type T the pair $([\![T]\!], [\![T^{\perp}]\!])$ is a mutual fixed point of Φ and Φ and so constitutes a stable saturated pair.

Lemma 14.

1. *$[\![T]\!]$ is a set of terminating terms.*
2. *$[\![A_1 \wedge A_2]\!] \supseteq \{\langle t_1, t_2 \rangle \mid t_i \in [\![A_i]\!],\ i = 1,2\}$.*
3. *$[\![A_1 \vee A_2]\!] \supseteq \{\sigma_1(p) \mid p \in [\![A_1]\!]\} \cup \{\sigma_2(p) \mid p \in [\![A_2]\!]\}$.*
4. *$(\lambda x.c) \in [\![A]\!]$ if for all $e \in [\![A^{\perp}]\!]$ we have $c[x \leftarrow e]$ terminates.*
5. *$[\![(A_1 \cap \cdots \cap A_k)]\!] \subseteq ([\![A_1]\!] \cap \cdots \cap [\![A_k]\!])$.*

Since each $[\![T]\!]$ consists of terminating expressions the following theorem implies that all typable expressions are terminating.

Theorem 15 (Type Soundness). *If expression t is typable with type T then $t \in [\![T]\!]$.*

The proof is by induction over typing derivations. All the hard work has been done in Lemma 14; the proof is organized by cases according to the last inference in the derivation, and each case follows readily from the clauses of the Lemma.

References

1. Z. M. Ariola and H. Herbelin. Minimal classical logic and control operators. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, volume 2719 of *LNCS*, pages 871–885. Springer-Verlag, 2003.
2. F. Barbanera and S. Berardi. A symmetric lambda calculus for classical program extraction. *Information and Computation*, 125(2):103–117, 1996.
3. G. M. Bierman. A computational interpretation of the $\lambda\mu$-calculus. In *Proc. of Symp. on Mathematical Foundations of Computer Science.*, volume 1450 of *LNCS*, pages 336–345. Springer-Verlag, 1998.
4. P.-L. Curien and H. Herbelin. The duality of computation. In *Proc. of the 5th ACM SIGPLAN Int. Conference on Functional Programming, ICFP'00*, Montreal, Canada, 2000. ACM Press.
5. R. David and K. Nour. Arithmetical proofs of strong normalization results for the symmetric $\overline{\lambda}\mu$-calculus. In *Typed Lambda Calculus and application, TLCA 2005*, volume 3461 of *LNCS*, pages 162–178. Springer-Verlag, 2005.
6. Ph. de Groote. On the relation between the $\lambda\mu$-calculus and the syntactic theory of sequential control. In *Logic Programming and Artificial Reasoning, LPAR'94*, volume 822 of *LNCS*, pages 31–43. Springer-Verlag, 1994.
7. D. Dougherty, S. Ghilezan, and P. Lescanne. Characterizing strong normalization in the Curien-Herbelin symmetric lambda calculus: extending the Coppo-Dezani heritage. In S. Berardi and U de' Liquoro, editors, *Theoretical Computer Science*, volume Festschrift Coppo, Dezani, Ronchi. To appear.
8. T. Griffin. A formulae-as-types notion of control. In *Proc. of the 19th Annual ACM Symp. on Principles Of Programming Languages, (POPL'90)*, pages 47–58, San Fransisco (Ca., USA), 1990. ACM Press.
9. H. Herbelin. *Séquents qu'on calcule : de l'interprétation du calcul des séquents comme calcul de λ-termes et comme calcul de stratégies gagnantes*. Thèse, U. Paris 7, Janvier 1995.
10. W. A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490, New York, 1980. Academic Press.
11. B. Knaster. Un théorème sur les fonctions d'ensembles. *Annales de la Societé Polonaise de Mathematique*, 6:133–134, 1928.
12. S. Lengrand. Call-by-value, call-by-name, and strong normalization for the classical sequent calculus. In Bernhard Gramlich and Salvador Lucas, editors, *ENTCS*, volume 86. Elsevier, 2003.
13. Ch. R. Murthy. Classical proofs as programs: How, what, and why. In J. Paul Myers Jr. and Michael J. O'Donnell, editors, *Constructivity in Computer Science*, volume 613 of *LNCS*, pages 71–88. Springer, 1991.
14. C.-H. L. Ong and C. A. Stewart. A Curry-Howard foundation for functional computation with control. In *POPL 24*, pages 215–227, 1997.
15. M. Parigot. An algorithmic interpretation of classical natural deduction. In *Proc. of Int. Conf. on Logic Programming and Automated Reasoning, LPAR'92*, volume 624 of *LNCS*, pages 190–201. Springer-Verlag, 1992.
16. M. Parigot. Proofs of strong normalisation for second order classical natural deduction. *The J. of Symbolic Logic*, 62(4):1461–1479, December 1997.
17. E. Polonovski. Strong normalization of $\lambda\mu\tilde{\mu}$-calculus with explicit substitutions. In Igor Walukiewicz, editor, *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004*, volume 2987 of *LNCS*, pages 423–437. Springer-Verlag, 2004.
18. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
19. C. Urban and G. M. Bierman. Strong normalisation of cut-elimination in classical logic. In *Typed Lambda Calculus and Applications, TLCA'99*, volume 1581 of *LNCS*, pages 365–380. Springer-Verlag, 1999.
20. Ph. Wadler. Call-by-value is dual to call-by-name. In *Proc. of the 8th ACM SIGPLAN Int. Conference on Functional Programming, ICFP'03*, pages 189–201, 2003.

A Cyclic Termination Proof of In-Place List Reversal using Separation Logic

James Brotherston[1], Richard Bornat[2], and Cristiano Calcagno[1]

[1] Imperial College, London
[2] Middlesex University, London

Abstract. We give a cyclic termination proof, based upon separation logic, for the classical imperative program performing in-place reversal of a "frying-pan" linked list segment (so called because the segment possibly has a cyclic tail).

1 Introduction

Algorithms that manipulate list structures, typically by means of pointer arithmetic, are one of the cornerstones of traditional imperative programming. Despite their importance, such algorithms have often defied formal analysis.

Consider a basic programming language in which programs are lists of labelled commands, the available commands include assignment ($x := y$) and conditional branching ($\texttt{if}\, \mathit{Cond}\, \texttt{goto}\, i$), and assignments permit pointer dereferencing ($y := [x]$, $[x] := y$). Now consider a segment of a standard linked list (containing no data other than the pointers from cell to cell) whose first element is pointed to by x and whose last element contains the pointer y. The classical algorithm for in-place reversal of such a list segment can be written in our toy programming language as follows:

```
1. $y := \mathsf{nil}$                      4. $x := [x]$        7. goto 2
2. if $x = \mathsf{nil}$ goto 8             5. $[z] := y$        8. stop        (1)
3. $z := x$                                 6. $y := z$
```

Separation logic [8, 10] offers a convenient formalism for describing heap structures. Its separating conjunction $*$ is used to describe a division of the heap into two disjoint parts in which the two conjuncts hold respectively. Thus, a linked list segment can be described by the following inductive definition in separation logic:

$$\mathit{ls}\, x\, y \Leftrightarrow (x = y \wedge \mathsf{emp}) \vee (\exists x'.\, x \mapsto x' * \mathit{ls}\, x'\, y) \tag{2}$$

where emp denotes the empty heap and $x \mapsto y$ denotes a single-celled heap whose contents are y and which is pointed to by x. Note that this definition does not preclude the possibility that the "end pointer" y points to a cell in the segment itself, i.e., that such segments may be cyclic. Under such conditions, it becomes hard to prove termination of the algorithm (1) above; we are not aware of an existing proof. Using a technique based upon *cyclic proof*, however, the proof becomes straightforward. For logics featuring inductive predicates or similar fixed-point constructions, cyclic proof provides an alternative to traditional inductive proof, modelled on Fermat's infinite descent [3, 6, 11].

2 Cyclic Termination Proofs in Separation Logic

We now outline a cyclic proof system tailored to proving termination of simple imperative programs, the full details of which will appear as [5].

We write *termination judgements* of the form $F \vdash_i$, where F is a formula of separation logic (including inductive predicates) and i is a label of the program, with the intended interpretation that the considered program always terminates when started from line i in a state satisfying the "precondition" F. Our proof rules are of two types: logical rules that operate on the preconditions (cf. [4]); and symbolic execution rules that operate on the program (cf. [1]). For the former, we use the notion of *bunches* (cf. [9]): trees whose leaves are formulas and whose internal nodes are labelled either by $\wedge$ or $*$. $\Gamma(-)$ is then notation for a bunch with a hole. Most of the rules are obvious, and we give only the most important ones here (we use t to range over logical terms and E to range over program expressions, and use the convention that primed variables x' are chosen fresh):

$$\frac{\mathit{Cond} \wedge F \vdash_j \quad \neg \mathit{Cond} \wedge F \vdash_{i+1}}{F \vdash_i} C_i \equiv \texttt{if}\, \mathit{Cond}\, \texttt{goto}\, j \qquad \frac{}{F \vdash_i} C_i \equiv \texttt{stop}$$

$$\frac{x = E[x'/x] * F[x'/x] \vdash_{i+1}}{F \vdash_i} C_i \equiv x := E \qquad \frac{E_1 \mapsto E_2 * F \vdash_{i+1}}{E_1 \mapsto t * F \vdash_i} C_i \equiv [E_1] := E_2$$

$$\frac{x = t[x'/x] \wedge (E \mapsto t * F)[x'/x] \vdash_{i+1}}{E \mapsto t * F \vdash_i} C_i \equiv x := [E]$$

$$\frac{\Gamma(t_1 = t_2 \wedge \mathsf{emp}) \vdash_i \quad \Gamma(t_1 \mapsto x' * \mathit{ls}\, x'\, t_2) \vdash_i}{\Gamma(\mathit{ls}\, t_1\, t_2) \vdash_i} \texttt{ls} \qquad \frac{\Gamma(F_2) \vdash_i}{\Gamma(F_1) \vdash_i} \; F_1 \vdash F_2 \quad (\text{Cut})$$

where the side condition in the (Cut) rule appeals to an independent proof method as in e.g. [4]. A *cyclic pre-proof* of a judgement $\Gamma \vdash_i$ is a finite derivation tree, with root $\Gamma \vdash_i$, in which every node to which no proof rule has been applied (called a *bud* of the tree) is assigned a interior node in the tree (called the *companion* of the bud) labelled by the same judgement. The *graph* of a pre-proof is obtained by identifying each bud with its companion.

Pre-proofs are not sound in general. Briefly, a pre-proof is a *bona fide* proof if for every infinite path in its graph, there is a tail of the path along which some inductive definition is unfolded infinitely often via its case-split rule (cf. the rule `ls` above). All such paths can then be disregarded from the proof by an infinite descent argument, whereby the remaining finite portion of proof is sound by standard methods. (This informal soundness condition can be formalised via a notion of *trace*, see e.g. [4, 6] for details.)

3 Example: in-place reversal of a frying-pan list

A cyclic list segment $ls\,x\,j$ in which the terminating pointer j points to a node already in the segment can be seen as a separated three-part structure of two acyclic list segments and a "join node", represented in separation logic as:

$$\exists k.(ls\,x\,j * j \mapsto k * ls\,k\,j) \tag{3}$$

Diagrammatically, such segments resemble a frying pan in which $ls\,x\,j$ is the handle and $ls\,k\,j$ is the pan. The reversal algorithm (1) goes down the handle, reversing it until it reaches the join node, which it redirects towards the reversed handle; then it goes round the pan, reversing that; then it re-redirects the join node to the reversed pan; finally it comes back up the handle, re-reversing it. The precondition is (3) and the invariant is:

$$\exists k1, k2, k3 \cdot \begin{pmatrix} (ls\,x\,j * ls\,y\,\mathsf{nil} * j \mapsto k1 * ls\,k1\,j) \vee \\ (ls\,k2\,\mathsf{nil} * j \mapsto k2 * ls\,x\,j * ls\,y\,j) \vee \\ (ls\,x\,\mathsf{nil} * ls\,y\,j * j \mapsto k3 * ls\,k3\,j) \end{pmatrix} \tag{4}$$

This invariant is sufficiently obvious that it can be discovered automatically [7], but it's hard to see what formula to use as a measure for this process, because x plays different rôles at different stages of the proof.

The interesting point of the proof is the loop from lines 2 to 8 (line 1 serves simply to establish $ls\,y\,\mathsf{nil}$ and thus the first disjunct of the invariant). Figure 1 shows a pre-proof of the judgement $I \vdash_2$, where I is the invariant (4) (with the existential quantification omitted for simplicity). The assignment of a companion to each bud is denoted by the arrows A–E. We have omitted some simple proof steps, required at the top of each branch to ensure that each bud has the same syntactic form as its companion. In the case of A and C, we replace y' by $k2$ and $k3$ respectively (an application of substitution), and $y = z \wedge z = j \wedge \mathsf{emp}$ by $ls\,y\,j$ (requiring an equality step and a cut step). For B, we replace $y = z \wedge (z \mapsto y' * ls\,y'\,\mathsf{nil})$ by $ls\,y\,\mathsf{nil}$ (requiring an equality step and a cut step) and, for D and E, we similarly replace $y = z \wedge (z \mapsto y' * ls\,y'\,j)$ by $ls\,y\,j$.

To see that our pre-proof is a cyclic proof, we must show that it meets the soundness condition. We observe that every infinite path in the proof graph has a tail that stays within one of the "cycles" induced by the arrows B, D, and E. Now: on the B cycle, the inductive formula $ls\,x\,j$ can be traced through the cycle and is unfolded infinitely often on it; for D, the same is true of the same formula; and for E, the same is true of the formula $ls\,x\,\mathsf{nil}$. Thus some inductive definition is unfolded infinitely often on some tail of every infinite path as required.

4 Conclusion

Our example proof above is long and tedious, but it involves no arithmetic and is very much the sort of thing an automatic tool (see e.g. [2]) might be expected to do. One can imagine list processing in such a tool being so stylised that it could recognise the need to contract the ls definitions to form the cycles B,D,E in the proof, and invent the empty heap definitions to form the cycles A and C.

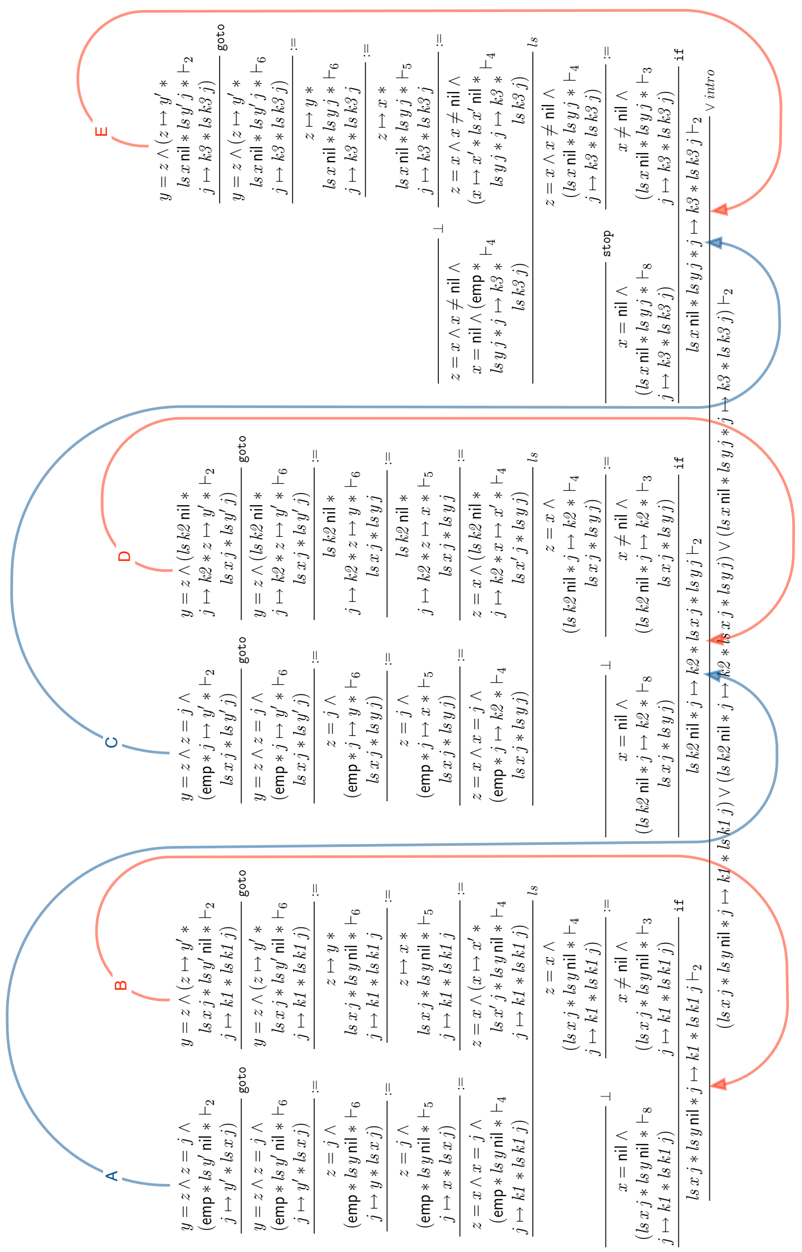


Fig. 1. A termination proof of in-place reversal of a frying-pan list

References

1. J. Berdine, C. Calcagno, and P.W. O'Hearn. Symbolic execution with separation logic. In K. Yi, editor, *APLAS 2005*, volume 3780 of *LNCS*, 2005.
2. J. Berdine, C. Calcagno, and P.W. O'Hearn. Smallfoot: Automatic modular assertion checking with separation logic. In *4th FMCO*, pp115-137, 2006.
3. James Brotherston. Cyclic proofs for first-order logic with inductive definitions. In B. Beckert, editor, *Proceedings of TABLEAUX 2005*, volume 3702 of *LNAI*, pages 78–92. Springer-Verlag, 2005.
4. James Brotherston. Formalised inductive reasoning in the logic of bunched implications. Submitted; a preliminary version appeared at the HAV 2007 workshop, 2007.
5. James Brotherston, Cristiano Calcagno, and Richard Bornat. Cyclic proofs of program termination in separation logic. Forthcoming.
6. James Brotherston and Alex Simpson. Complete sequent calculi for induction and infinite descent. To appear in Proceedings of LICS-22, 2007.
7. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. 16th TACAS, pp287–302, 2006.
8. S. Isthiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *28th POPL*, pages 36–49, 2001.
9. P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 99.
10. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th LICS*, pp 55-74, 2002.
11. Christoph Sprenger and Mads Dam. On the structure of inductive reasoning: circular and tree-shaped proofs in the μ-calculus. In *Proceedings of FOSSACS 2003*, volume 2620 of *Lecture Notes in Computer Science*, pages 425–440, 2003.

Author Index