11th International Workshop on Termination

WST 2010

Edinburgh, July 14–15, 2010

Peter Schneider-Kamp (editor)

University of Edinburgh, School of Informatics

An Abstract Path Ordering

Nachum Dershowitz[*]
School of Computer Science
Tel Aviv University
Ramat Aviv, Israel
nachum.dershowitz@cs.tau.ac.il

Abstract

Abstract combinatorial commutation properties for separating well-foundedness of unions of relations can be applied to generic path orderings used in termination proofs.

1 Introduction

Path orderings provide a convenient and popular method of proving termination, particularly of term-rewriting systems. Here, we set out to prove the well-foundedness of an abstract path ordering—in the style of [15, 3]—which includes the usual path orderings on first-order terms as special cases. We will apply the commutation methods of [7] plus a strong variant of lifting.

2 The Selection Property

All relations herein are binary. Juxtaposition is used for composition of relations. We represent union by $+$, and denote the reflexive, transitive, and reflexive-transitive closures of relation E by E^ε, E^+, and E^*, respectively. We will use E^∞ to represent the set of "immortal" elements s for which there is an infinite E-chain $s\ E\ s'\ E\ s''\ E\ \cdots$ of elements of the underlying set.

Definition 1 (Selection [7])**.** *Relation B* selects *relation A if*

$$BA^+ \subseteq A(A+B)^* + B^+ \ .$$

In other words, if one can get from an element s to an element t by one B-step followed by one or more A-steps, then one can also get from s to t by first taking an A-step and then some combination of A- and B-steps, or else one can get there by one or more B-steps alone. This is a weaker requirement than the "local" condition explored in [11, 12] and called "lazy commutation" in [7].

Theorem 2 ([7, Theorem 72][1])**.** *If relation B selects relation A, then*

$$(A+B)^\infty = A^*B^*(A^\infty + B^\infty) \ .$$

This notation is meant to convey that one can get from any element that is immortal in the union $A+B$ to an element that is immortal in one of the two component relations by taking some number of A-steps followed by some number of B-steps. This implies, of course, that the union is well-founded whenever both A and B are.

When A is transitive, as it will be in the cases of interest here, selection is the same as the following local condition:

[*]Research supported in part by the Israel Science Foundation (grant no. 254/01).

[1]The proof in [7] relies on a more general claim (Theorem 39) about "constriction". The latter, however, is phrased there too broadly. Nevertheless, it does apply to the case in hand. I am grateful to Ori Brostovski for pointing this out.

Definition 3 (Jumping). *Relation A* jumps over *relation B if*

$$BA \subseteq A(A+B)^* + B^+ .$$

This is noticeably weaker than lazy commutation [11, 12, 7], which allows only one B rather than B^+.

Corollary 4. *If transitive relation A jumps over relation B, then*

$$(A+B)^\infty = A^\varepsilon B^* (A^\infty + B^\infty) .$$

This, too, implies "separation" of termination of the union $A+B$.

3 The Abstract Path Ordering

We propose the following generic definition of path orderings:

Definition 5 (Abstract Path Ordering). *The* abstract path ordering *is a relation $>$ (not necessarily transitive) on some set T, parameterized by two other abstract relations, $\gg$ and well-founded $\rhd$, and by arbitrary binary conditions C and D, defined as follows:*

$$t > s \quad \text{if} \quad \begin{cases} t \succ s \text{ and } t\ C\ s & \text{(a)} \\ \text{or} & \\ t \gg s \text{ and } t\ (\succ + >)/\rhd\ s \text{ and } t\ D\ s\ , & \text{(b)} \end{cases}$$

where $\succ$ is short for $\rhd^+ >^*$ (or just $\rhd >^*$, in the transitive $\rhd$ case), and the "division" operator is defined by $B/A = \{\langle x,y\rangle : \forall z.\ yAz \Rightarrow xBz\}$. In other words, in case (b), $t \succ u$ or $t > u$ for all $\rhd$-neighbors u of s.

This is a generalization of the abstract ordering given in [15].

Let $\sqsupset$ be short for $\gg \cap (\succ + >)/\rhd$, the main part of case (b). By the cases of the definition, we have

$$> \ \subseteq \ \succ + \sqsupset \ \subseteq \ \succ + \gg \ .$$

Lemma 6. *For the above abstract path ordering, relation $\sqsupset$ selects $\rhd$.*

Proof. By the division clause of the second case (b), one has $\sqsupset \rhd \subseteq \succ + >$. Also, the recursive definition of $>$ must expand so that $> \subseteq (\rhd + \sqsupset)^+$. Pasting the various facts together, we get

$$\sqsupset \rhd \ \subseteq \ \succ + > \ \subseteq \ \succ + (\succ + \sqsupset) \ \subseteq \ \rhd (\rhd^* >^*) + \sqsupset \ \subseteq \ \rhd (\rhd + \sqsupset)^* + \sqsupset \ .$$

So, in fact, $\rhd$ commutes lazily over $\sqsupset$, which implies selection (by an easy induction). □

It follows from Theorem 2 that $>$ is well-founded if $\sqsupset$ is. Of course, $\sqsupset$ is well-founded if $\gg$ is. So:

Proposition 7. *An abstract path ordering is well-founded whenever its component relation $\gg$ is.*

This works, as is, for some interpretation-based termination orderings.

4 Lifting and Escaping

The problem is that, for path orderings, $\gg$ is normally defined, recursively, in terms of $>$ applied to subterms.

Theorem 8. *An abstract path ordering $>$ defined on a set T is well-founded if well-foundedness of $>$ for the $\rhd$-neighbors of an element $s \in T$ implies well-foundedness of $\sqsupset$ (or of $\gg$) for s.*

Proof. We are presuming that $\rhd$ is well-founded. Let s be a $\rhd$-minimal element of T that is immortal in $>$. By Theorem 2 (with $\rhd$ for A and $\sqsupset$ for B) and Lemma 6, there must be an $s' \lhd^* s$ that is immortal in $\sqsupset$ (and, therefore, in $\gg$). By the assumption of this theorem, there must be a neighbor $s'' \lhd s'$ that is immortal in $>$, contradicting minimality of s. $\square$

This condition is captured by the following:

Definition 9 (Lifting)**.** *Relation A* lifts to *relation B if*

$$B^\infty \quad \subseteq \quad A(A+B)^\infty \,.$$

In general,

Theorem 10 ([7, Lemma 84])**.** *If relation A lifts to B and B selects A, then*

$$(A+B)^\infty = (A+B)^* A^\infty \,.$$

Corollary 11. *An abstract path ordering $>$ is well-founded if $\rhd$ lifts to $\sqsupset$.*

A minori ad maius if it lifts to $\gg$.

This applies to the nested multiset ordering [10], where $\gg$ is the multiset ordering, and to lexicographic orderings. The general case of such "lifted" definitions was first studied in [17] and was pursued further in [14, 15].

It turns out, however, that oftentimes we need a weaker alternative to lifting, in which the A-step need only take place *eventually*. Borrowing modal-logic notation (see [8] for details), this is captured by the next definition.

Definition 12 (Escaping)**.** *Relation A* escapes from *relation B if*

$$B^\infty \quad \models \quad \Diamond [\![A(A+B)^\infty]\!] B^\infty \,.$$

Here, B^∞ is being used to denote the set of all infinite B-chains and the $\Diamond$ means that some suffix looks like $[\![A(A+B)^\infty]\!] B^\infty$. The double-bracket notation turns the set (of sequences) $A(A+B)^\infty$ into the relation between those elements having immortal A-neighbors and everything. Accordingly, the definition means that there is a point in every infinite B-chain such that an A-step out of that point leads to a potentially "immortal" element in the union. Escaping is somewhat reminiscent of the "bar induction" criterion in [15].

Theorem 13. *If relation A escapes from B and B selects A, then*

$$B^\infty \quad \subseteq \quad B^* A^\infty \,.$$

So, under these conditions, B is well-founded if A is. Because we are presuming well-foundedness of $\rhd$, the inescapable conclusion is that

Corollary 14. *An abstract path ordering $>$ is well-founded if $\rhd$ escapes from $\sqsupset$.*

The multiset path ordering [4], lexicographic path ordering [17], and recursive path ordering [18, 5] are all special cases, where $\rhd$ is the proper subterm relation (so, $\rhd^+ = \rhd$), C and D are always true, and $\gg$ is a recursive lifting of $>$ to multisets, precedence (first) and (then) multisets, precedence and tuples lexicographically, and a mixture thereof, respectively.

5 Discussion

We are optimistic that the commutation-based approach taken here will likewise help for advanced path orderings, like the general path ordering [9] and higher-order recursive-path-ordering [13, 16, 2], without recourse to reducibility/computability predicates, because (as pointed out in [6]) there is an analogy between the use of reducibility predicates and the use in proofs of well-foundedness of the "constricting" derivations used in the proof of Theorem 2 cited above.

We can apply this commutation method to analyze the dependency-pair method of proving termination. (See [1]; compare [6].) We also hope to analyze minimal bad sequence arguments for well-quasi-orderings in a similar fashion. (See [19]; compare [15].)

References

[1] Thomas Arts and Jürgen Giesl, 2000, "Termination of term rewriting using dependency pairs", *Theoretical Computer Science*, vol. 236, pp. 133–178.

[2] Frédéric Blanqui, Jean-Pierre Jouannaud, and Albert Rubio, Oct. 2007, "HORPO with computability closure: A reconstruction", *Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2007)*, Yerevan, Armenia, N. Dershowitz and A. Voronkov, eds., Lecture Notes in Computer Science, vol. 4790, Springer-Verlag, Berlin, pp. 138–150.

[3] Jeremy E. Dawson and Rajeev Gore, 2007, "Termination of abstract reduction systems", *Proceedings of Computing: The Australasian Theory Symposium (CATS 2007)*, Ballarat, Australia, pp. 35–43.

[4] Nachum Dershowitz, Mar. 1982, "Orderings for term-rewriting systems", *Theoretical Computer Science*, vol. 17, no. 3, pp. 279–301.

[5] Nachum Dershowitz, 1987, "Termination of rewriting", *J. of Symbolic Computation*, vol. 3, pp. 69–116.

[6] Nachum Dershowitz, Sept. 2004, "Termination by abstraction", *Proceedings of the Twentieth International Conference on Logic Programming*, St. Malo, France, Lecture Notes in Computer Science, vol. 3132, Springer-Verlag, Berlin, pp. 1–18.

[7] Nachum Dershowitz, 2009, "On lazy commutation", *Languages: From Formal to Natural*, O. Grumberg, M. Kaminski, S. Katz, and S. Wintner, eds., Lecture Notes in Computer Science, vol. 5533, Springer-Verlag, Berlin, pp. 59–82. Available at `http://www.cs.tau.ac.il/~nachum/papers/LazyCommutation.pdf`.

[8] Nachum Dershowitz, 2010, "Notations for sets of sequences", draft. Available at `http://www.cs.tau.ac.il/~nachum/papers/notation.pdf`.

[9] Nachum Dershowitz and Charles Hoot, May 1995, "Natural termination", *Theoretical Computer Science*, vol. 142, no. 2, pp. 179–207.

[10] Nachum Dershowitz and Zohar Manna, Aug. 1979, "Proving termination with multiset orderings", *Communications of the ACM*, vol. 22, no. 8, pp. 465–476.

[11] Henk Doornbos, Roland Backhouse, and Jaap van der Woude, 1997, "A calculational approach to mathematical induction", *Theoretical Computer Science*, vol. 179, pp. 103–135.

[12] Henk Doornbos and Burghard von Karger, 1998, "On the union of well-founded relations", *Logic Journal of the IGPL*, vol. 6, no. 2, pp. 195–201.

[13] Maribel Fernández and Jean-Pierre Jouannaud, 1994, "Modular termination of term rewriting systems revisited", in E. Astesiano, G. Reggio, and A. Tarlecki, eds., *Recent Trends in Data Type Specification*, Lecture Notes in Computer Science, vol. 906, Springer-Verlag, Berlin, pp. 255–272.

[14] Maria C. F. Ferreira and Hans Zantema, 1995, "Well-foundedness of term orderings", in: N. Dershowitz, ed., *Proceedings of the 4th International Workshop on Conditional Term Rewriting Systems (CTRS '94)*, Lecture Notes in Computer Science, vol. 968, Springer-Verlag, Berlin, pp. 106–123.

[15] Jean Goubault-Larrecq, Sept. 2001, "Well-founded recursive relations", *Proceedings of the 15th International Workshop on Computer Science Logic (CSL '01)*, Paris, France, Lecture Notes in Computer Science, vol. 2142, Springer-Verlag, Berlin, pp. 484–497.

[16] Jean-Pierre Jouannaud and Albert Rubio, 1999, “Higher-order recursive path orderings”, *Proceedings 14th Annual IEEE Symposium on Logic in Computer Science (LICS)*, Trento, Italy, pp. 402–411.

[17] Sam Kamin and Jean-Jacques Lévy, February 1980, “Two generalizations of the recursive path ordering”, unpublished note, Department of Computer Science, University of Illinois, Urbana, IL. Available at `http://pauillac.inria.fr/~levy/pubs/80kamin.pdf`.

[18] Pierre Lescanne, 1990, “On the recursive decomposition ordering with lexicographical status and other related orderings”, *J. Automated Reasoning*, vol. 6, no. 1, pp. 39–49.

[19] Paul-André Melliès, 1998, “On a duality between Kruskal and Dershowitz theorems”, *Proceedings of the 25th International Conference on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science, vol. 1443, Springer-Verlag, Berlin, pp. 518–529.

Length preserving string rewriting with exponential derivation length

Hans Zantema[1,2]

[1] Department of Computer Science, TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, email: H.Zantema@tue.nl
[2] Institute for Computing and Information Sciences, Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands

Abstract. Inspired by binary counting we present and analyze a small length preserving terminating string rewriting system with several nice properties, in particular having derivations of exponential length. These observations are the key for the proof in [1] that a particular kind of RNA-editing admits a sequence of modifications that essentially requires an exponential number of steps.

1 Length preserving string rewriting

Let Σ be any set of symbols.

A *string rewrite rule* $\ell \to r$ over Σ consists of two strings $\ell, r \in \Sigma^*$.

A *string rewriting system* (SRS) over Σ is a set of string rewrite rules over Σ. We assume Σ to be finite, and only consider SRSs that are finite.

An SRS R is called *length-preserving* if the length $|\ell|$ of ℓ is equal to the length $|r|$ of r for every rule $\ell \to r \in R$.

For an SRS R its rewrite relation $\to_R$ is defined by

$$x\ell y \to_R xry$$

for every $x, y \in \Sigma^*$ and every $\ell \to r \in R$.

An SRS R is called *terminating* if $\to_R$ is terminating.

The *derivation length* $d(t, R)$ of a string t with respect to an SRS R is the maximal value of n such that t rewrites in n steps to any string. For R being terminating this is well-defined; here our finiteness assumptuion is essential.

An SRS has *exponential derivation length* if there exists $c > 1$ such that

$$\sup_{|t|=n} d(t, R) = \Theta(c^n).$$

The following proposition states that for length-preserving terminating SRSs this derivation length is the highest possible.

Proposition 1. *Let R be a length-preserving terminating SRS over Σ, and $c = |\Sigma|$. Then*

$$\sup_{|t|=n} d(t, R) = O(c^n).$$

Proof. Assume t with $|t| = n$ admits a reduction of length greater than c^n. Then all strings in this reduction are in Σ^n, due to length-preservingness. The total number of such strings is $|\Sigma^n| = c^n$, so this reduction contains double occurrences, giving rise to infinite reduction, contradicting termination. $\square$

If $\ell_1 = xy$ and $\ell_2 = yz$ for two (possibly equal) rules $\ell_1 \to r_1$ and $\ell_2 \to r_2$, and a non-empty string y, then we say that these rules *overlap*. An SRS is called *orthogonal* if there are no other overlaps than the trivial overlaps obtained by choosing twice the same rule and $\ell_1 = y = \ell_2$. It is easy to see that orthogonal string rewriting satisfies the *diamond property*, that is, if $t \to_R u_i$ for $i = 1, 2$ and $u_1 \neq u_2$, then there exists v such that $u_i \to_R v$ for $i = 1, 2$. As a consequence we have the following proposition.

Proposition 2. *Let R be a terminating and orthogonal SRS. Then every string t has a unique normal form u, and every reduction from t to u has the same length.*

For an SRS R its *string reverse* R^r is defined to consist of all rules $\ell^r \to r^r$ for $\ell \to r \in R$; here t^r denotes the string reversal of a string t. Trivially, termination, derivation lengths and orthogonality are preserved under string reversal.

However, there is also another kind of reversal. For an SRS R we define its *rewrite reverse* $R^{\leftarrow}$ to consist of all rules $r \to \ell$ for $\ell \to r \in R$. For arbitrary string rewriting the basic properties are not preserved, for instance, the single rule $aa \to a$ is terminating, but its rewrite reverse is not terminating.

Proposition 3. *Let R be a length-preserving SRS. Then R is terminating if and only if $R^{\leftarrow}$ is terminating, and R has exponential derivation length if and only if $R^{\leftarrow}$ has exponential derivation length.*

Proof. If R is not terminating then there is a string t having an infinite reduction. Let $n = |t|$. Then all strings in this reduction are in the finite set Σ^n. So there is a string u in this reduction such that $u \to_R^+ u$. By reversing arrows this yields an infinite reduction of $R^{\leftarrow}$, proving that $R^{\leftarrow}$ is not terminating. Similarly, non-termination of $R^{\leftarrow}$ yields non-termination of R as $(R^{\leftarrow})^{\leftarrow} = R$. The claim on exponential derivation lengths follows from the observation that if $t \to_R^* u$ then $|t| = |u|$, and there is a reduction from u to t of the same length with respect to $R^{\leftarrow}$. $\square$

In the next section we encode binary counting in length-preserving string rewriting, yielding a length-preserving SRS with exponential derivation length.

2 Binary counting

In n-digit binary numbers we can count in $2^n - 1$ steps from the number 0 represented by the string 0^n to the number $2^n - 1$ represented by the string 1^n. For doing so we need a carry mechanism for which we introduce a fresh symbol c. Adding one to a number ending in 0 is represented by replacing this 0 by 1, and adding one to a number ending in 1 is represented by replacing this 1 by c. Next we need rules for propagating this c: the pattern $0c$ should be replaced by 10 and c: the pattern $1c$ should be replaced by $c0$. Summarizing, this counting from 0^n to 1^n can be mimicked by the SRS consisting of the four rules

$$\begin{aligned} 0 &\to 1 \\ 1 &\to c \\ 0c &\to 10 \\ 1c &\to c0 \end{aligned}$$

This SRS is terminating, as is easily seen by dependency pair analysis. This is the most straightforward SRS representation of binary counting; it appeared before, e.g., in a presentation of Dieter Hofbauer and Johannes Waldmann on Proof Theory and Rewriting in Obergurgl, March 30, 2010.

By construction there is a reduction from 0^n to 1^n containing $2^n - 1$ applications of the first two rules, so having exponential derivation length. This reduction can even be extended: the string 1^n further reduces to the normal form c^n.

However, by applying only the first two rules there is also a reduction from 0^n to c^n in only $2n$ steps. Note that in this linear reduction the first two rules are applied on all positions, while in the exponential length reduction the first two rules are only applied on the rightmost position. To force that the first two rules are only applied on the rightmost position, we introduce a fresh symbol b, and replace the first two rules $\ell \to r$ by $\ell b \to rb$, yielding

$$\begin{aligned} 0b &\to 1b \\ 1b &\to cb \\ 0c &\to 10 \\ 1c &\to c0 \end{aligned}$$

This SRS is still terminating and admits exponential derivation length for the same reason, but now the system is orthogonal, so by Proposition 2 every reduction from $0^n b$ to $c^n b$ has the same length. As this system still mimicks the binary counting in an exponential number of steps, we conclude that every reduction from $0^n b$ to $c^n b$ has length exponential in n.

Next, note that in the reduction encoding binary counting every application of the first rule is directly followed by an application of the second rule, and the second rule is only applied directly after the first rule. So the first two rules can be combined in one rule, still keeping exponential derivation length. The result is the following SRS B:

$$B : \begin{cases} 0b \to cb \\ 0c \to 10 \\ 1c \to c0 \end{cases}$$

In the rest of this note we investigate this particular SRS B. Just like the earlier variants, termination of B immediately follows from dependency pair analysis. The SRS B has been designed in such a way that $0^n b$ rewrites to $c^n b$

in a number $f(n)$ of steps, for every n, where $f(n)$ is exponential in n. Let us analyze $f(n)$ more precisely. We have $f(0) = 0$, and by

$$0^n b \rightarrow^{f(n-1)} 0c^{n-1}b \rightarrow^{n-1} 1^{n-1}0b \rightarrow 1^{n-1}cb \rightarrow^{n-1} c0^{n-1}b \rightarrow^{f(n-1)} c^n b$$

we obtain

$$f(n) = 2f(n-1) + 2n - 1,$$

for every $n > 0$, from which one easily proves by induction that

$$f(n) = 3 * 2^n - 2n - 3$$

for every $n \geq 0$.

A remarkable property is the fact that $B^{\leftarrow}$ is a renaming of B. In this renaming the symbols 0 and c are swapped, while the symbols 1 and b are kept. By this swapping also the second and third rule swap. This expresses the fact that counting up in binary notation is exactly the same as counting down, while swapping the symbols 0 and c.

We saw that $0^{n-1}b$ reduces to $c^{n-1}b$ in $f(n-1)$ steps. The next proposition states that this is the longest possible reduction of strings of length n.

Proposition 4. *In B, no string of length n admits a reduction of length greater than $f(n-1)$.*

Proof. We do induction on n. Assume t is a string of length n of maximal reduction length. Assume it contains a b on position k for $k < n$, then the reduction steps can be split up in steps on the $n-k$ symbol right from this b, and the k symbols on position k and left from it. These steps are independent, so by the induction hypthesis the total number of steps is at most $f(k-1)+f(n-k-1)$, which is always less than $f(n-1)$. So in the remaining case there are no b's except for possibly the last position. If there are no b's at all then only the second and third rule can be applied, giving a linear reduction length (count $2\#c + \#1$). So t consists of $n-1$ symbols from $\{0, 1, c\}$ followed by a b. As t is assumed to have maximal reduction length, it is in normal form with respect to $B^{\leftarrow}$, that is, it does not contain patterns cb, 10 or $c0$. From this we conclude that either $t = 0^{n-1}b$ or t ends in $1b$. In the latter case this end pattern $1b$ will never be reduced due to the shape of the rules, so only the seond and third rule apply, yielding linear reduction length as observed before. So in the remaining case we have $t = 0^{n-1}b$. As we know that every reduction of $0^{n-1}b$ to its normal form $c^{n-1}b$ has length $f(n-1)$, this concludes the proof. $\square$

With respect to the SRS B the next proposition implies that $f(n-1)$ is not only a bound on reduction lengths of strings of length n, it is also a bound on conversion lengths.

Proposition 5. *Let R be a terminating length-preserving SRS for which both R and $R^{\leftarrow}$ are orthogonal. Let $s \leftrightarrow_R^n t$, for which no conversion from s to t exists shorter than n steps. Then there is a string u with $|u| = |s| = |t|$ having a reduction of length n.*

Proof. Applying the diamond property for $\to_R$ yields v with $s \to_R^p v$ and $t \to_R^q v$ with $p+q \leq n$, due to minimality of n we obtain $p+q = n$. Similarly, applying the diamond property for $\to_{R^\leftarrow}$ yields u with $u \to_R^{p'} s$ and $u \to_R^{q'} t$ with $p' + q' \leq n$, due to minimality of n we obtain $p' + q' = n$. From Proposition 2 we conlude that all reductions from u to v have the same length, so $p + p' = q + q'$. Now the equalities imply $p + p' = q + q' = n$, so u has a reduciton of length $p + p' = n$ to v. □

3 An application

In [1] a mutation mechanism has been investigated inspired by a particular kind of RNA-editing. One of the key results states that in this mechanism it is possible to start in some string of length n and end in another string after an exponential number of steps, while it is not possible to reach the same string in less than exponentially many steps. The key argument makes use of our results on B: it is shown that the mutation mechanism can mimick string rewriting, while starting from the right kind of strings essentially no other steps can be done than those that mimick string rewriting. For the argument the following properties were essential:

- B is orthogonal,
- $B^\to$ is orthogonal, that is, the right-hand sides of B are non-overlapping,
- all left-hand sides and right-hand sides of B have length two,
- there are strings t_i, u_i for which $|t_i| = |u_i| = i$, for all i, and t_i rewrites to u_i, but not in a number of steps less than exponential in i.

References

1. H. Zantema. Complexity of guided insertion-deletion in RNA-editing. In H. Fernau and C. Martin-Vide, editors, *4th International Conference on Language and Automata Theory and Applications (LATA 2010)*, Lecture Notes in Computer Science. Springer, 2010. to appear.

On Disproving Termination of Constrained Term Rewriting Systems [*]

Naoki Nishida
Nagoya University
Nagoya, Japan

Masahiko Sakai
Nagoya University
Nagoya, Japan

Tatsuya Hattori
Nagoya University
Nagoya, Japan

Abstract

This paper shows a sufficient condition for non-termination of constrained term rewriting systems. There is a distance in the non-termination proofs of constrained and unconstrained systems, e.g., constrained rewrite rules such that the right-hand sides are encompassments of the left-hand sides do not always cause non-termination. For such constrained rewrite rules, we characterize extra constraints that ensure non-termination caused by the constrained rewrite rules. We also show that such extra constraints sometimes can be obtained from the constraints of the constrained rewrite rules, by removing some closures from the disjunctive normal forms of the constraints.

1 Introduction

Constrained (un)conditional term rewriting systems are sets of constrained (un)conditional rewrite rules [21, 15, 18, 7, 4, 5, 12, 20]. For such systems, the constraint parts of rules are evaluated by some built-in semantics given by the membership relation, linear integer arithmetics (so-called *Presburger arithmetics*), equational theories independent on the rewrite rules, and so on. The condition parts of rules are evaluated by the rewrite rules recursively. Constrained systems have been enriched such as *constrained equational systems* (CESs, for short) [10, 11], that are rewrite systems with built-in numbers and semantic data structures. This paper deals with *constrained unconditional term rewriting systems* (constrained TRSs, for short) defined in [5, 12, 20].

Theorem proving methods for constrained (un)conditional TRSs are investigated in several ways [6, 1, 4, 9, 12, 5, 20]. In these methods, termination of given constrained systems has to be guaranteed (or proved in advance). Moreover, before starting the process of theorem proving, users have to specify reduction orders to be consistent with the reduction of the given systems. The methods proceeds, orienting equations by means of the reduction orders at each step of the expansion operation, and carrying them as hypotheses. All the combined systems of the given system and intermediate hypotheses must be consistent with the reduction orders in order to guarantee correctness of proofs obtained by the methods.

It is very difficult to know in advance which reduction orders are adequate for given constrained systems and equation sets to be proved, as well as the case of *completion procedures* [3]. To avoid this difficulty, termination provers can be used instead of reduction orders, as well as the case of completion procedures [22]. At each step of the orientation, the methods orients almost all equations temporally in a direction, combines each of them with the given system and the adopted hypotheses, and adds one of the temporally oriented equations such that the termination provers succeeds in proving termination of the corresponding combined systems, into the adopted hypothesis set. This operation guarantees that, if the process finishes successfully, then the reduction of the combined system of the given system and all the adopted hypotheses can be used as the reduction orders we expected. Especially, in the case of constrained TRSs (as shown later or in [12, 20, 11]) obtained from imperative programs with `while`-loops, the approach of employing termination provers is often helpful because any path-based order such as lexicographic path order is hardly helpful in proving termination of such constrained TRSs.

When termination provers are employed in theorem proving methods, at each step of the expansion, the provers are invoked great many times and they are sometimes applied to non-terminating systems because the methods orients almost all equations temporally in both directions until the methods find an appropriate oriented equation. In each execution, the termination provers tend to spend much time on

[*]This work is partly supported by MEXT. KAKENHI #20300010, #20500008, and #21700011.

trying to prove termination of systems even if the systems are not terminating. Moreover, if the provers fail to disprove termination, then the provers must wait for either the ending of all the processes or the timeout. For this reason, improvements of disproving power of the provers is promising for making the theorem proving methods efficient.

The DP framework [2, 14, 13], one of the termination proof techniques for unconstrained term rewriting systems (TRSs, for short), has been extended for the class of CESs [10, 11]. To prove termination of constrained TRSs, we can use the results in [10, 11], by adapting constrained TRSs to CESs. On the other hand, any sufficient condition for disproving termination of constrained systems is not known, except for the detection of unconstrained rewrite rules of the form $t \to C[t\theta]$. Of course, it would be possible to extend this method for constrained rules, e.g., the detection of constrained rules $t \to C[t\theta]$ $[\![\phi]\!]$ such that ϕ is valid. However, such an extension is not so practical because the constraints of rules are hardly valid. Thus, we need to relax the side condition that ϕ is valid.

Now, we will take a close look at the relation between non-termination and the constraints of rules, by using a running example. Consider the following constrained TRS R_{sum} obtained by transforming the following imperative program [12] (or [11], adapting CESs to constrained TRSs and adding the rules for the entry function sum and the `return`-statement):

```
int sum(int x){
    int i = 0, z = 0;
    while( x != i ){
        z += i+1;
        i ++;
    }
    return z;
}
```

$$R_{\mathsf{sum}} = \left\{ \begin{array}{rll} \mathsf{sum}(x) \to & \mathsf{eval}(x,0,0) & \\ \mathsf{eval}(x,i,z) \to & \mathsf{eval}(x,\mathsf{s}(i),\mathsf{plus}(z,\mathsf{s}(i))) & [\![x \succ i \vee i \succ x]\!] \\ \mathsf{eval}(x,i,z) \to & z & [\![\neg(x \succ i \vee i \succ x)]\!] \\ \mathsf{plus}(0,y) \to & y & \\ \mathsf{plus}(\mathsf{s}(x),y) \to & \mathsf{s}(\mathsf{plus}(x,y)) & [\![\mathsf{s}(x) \succ 0]\!] \\ \mathsf{plus}(\mathsf{p}(x),y) \to & \mathsf{p}(\mathsf{plus}(x,y)) & [\![0 \succ \mathsf{p}(x)]\!] \\ \mathsf{s}(\mathsf{p}(x)) \to & x & \\ \mathsf{p}(\mathsf{s}(x)) \to & x & \end{array} \right\}$$

Here, the function symbols 0, s, p, and plus represent linear arithmetic expressions over integers as usual, and the predicate symbol $\succ$ is interpreted by $>$ over integers. For a non-negative integer n, $\mathsf{sum}(\mathsf{s}^n(0))$ computes the summation from 0 to n, e.g., $\mathsf{sum}(\mathsf{s}^{10}(0))$ is reduced by R_{sum} to $\mathsf{s}^{55}(0)$. Terms obtained by applying the entry function sum to terms over $\{0,\mathsf{s}(),\mathsf{plus}(,)\}$ are terminating but R_{sum} is not terminating, e.g., $\mathsf{eval}(0,\mathsf{s}(0),0)$ is not terminating. On the other hand, the constrained TRS obtained from R_{sum} by replacing $x \succ i \vee i \succ x$ by $x \succ i$ is terminating. Thus, it is not true that any constrained rewrite rules of the form $t \to C[t\theta]$ $[\![\phi]\!]$ cause non-termination.

The reason why every rule $t \to C[t\theta]$ $[\![\phi]\!]$ does not always cause non-termination is that every reduction $t\sigma \to C\sigma[t\theta\sigma]$ does not provide an infinitely contiguous reductions $t\theta\sigma \to C\theta\sigma[t\theta\theta\sigma]$, $t\theta\theta\sigma \to C\theta\theta\sigma[t\theta\theta\theta\sigma], \cdots$, i.e., the validity of $\phi\sigma$ does not imply the validity of $\phi\theta\sigma$, $\phi\theta\theta\sigma, \cdots$. Thus, the validity of $\phi \Rightarrow \phi\theta$ is an alternative to the validity of ϕ. Unfortunately, this alternative is still weak. For example, consider the second rule $\mathsf{eval}(x,i,z) \to \mathsf{eval}(x,\mathsf{s}(i),\mathsf{plus}(z,\mathsf{s}(i)))$ $[\![x \succ i \vee i \succ x]\!]$ in R_{sum}, a substitution $\theta = \{i \mapsto \mathsf{s}(i), z \mapsto \mathsf{plus}(z,\mathsf{s}(i))\}$, and a substitution $\sigma = \{x \mapsto \mathsf{s}(0), i \mapsto 0, z \mapsto 0\}$. The reduction $\mathsf{eval}(x,i,z)\sigma \equiv \mathsf{eval}(\mathsf{s}(0),0,0) \to_{R_{\mathsf{sum}}} \mathsf{eval}(\mathsf{s}(0),\mathsf{s}(0),\mathsf{plus}(0,\mathsf{s}(0))) \equiv \mathsf{eval}(x,i,z)\theta\sigma$ holds but $\mathsf{eval}(x,i,z)\theta\sigma \to_{R_{\mathsf{sum}}} \mathsf{eval}(x,i,z)\theta\theta\sigma$ does not hold. In fact, $(x \succ i \vee i \succ x) \Rightarrow (x \succ i \vee i \succ x)\theta$ is not valid.

Recall what is sufficient for non-termination, i.e., the validity of $\phi\sigma$, $\phi\theta\sigma$, $\phi\theta\theta\sigma, \cdots$. This is not true for all substitutions σ satisfying ϕ but true for some of them. For example, given $\sigma' = \{x \mapsto 0, i \mapsto \mathsf{s}(0)\}$, all of the constraints $(x \succ i \vee i \succ x)\sigma'$, $(x \succ i \vee i \succ x)\theta\sigma'$, $(x \succ i \vee i \succ x)\theta\theta\sigma', \cdots$ are valid, i.e., $\mathsf{eval}(x,i,z)\sigma'$ is not terminating. Actually, this holds for any substitutions σ' such that $(i \succ x)\sigma'$, $(i \succ x)\theta\sigma'$, $(i \succ x)\theta\theta\sigma', \cdots$ are valid. Thus, for such substitutions σ', all of the constraints $(x \succ i \vee i \succ x)\sigma'$, $(x \succ i \vee i \succ x)\theta\sigma'$, $(x \succ i \vee i \succ x)\theta\theta\sigma', \cdots$ are valid because the validity of $(i \succ xi)\theta \cdots \theta\sigma'$ implies the validity of $(x \succ i \vee i \succ x)\theta \cdots \theta\sigma'$. To conclude, a constrained rule $t \to C[t\theta]$ $[\![\phi]\!]$ causes non-termination if there exist a substitution σ' and a constraint η such that $\eta\sigma'$, $\eta\theta\sigma'$, $\eta\theta\theta\sigma', \cdots$, and $\eta \Rightarrow \phi$ are valid. By replacing the existence of σ' by the satisfiability of η, the sufficient condition is translated into the one that there exists a satisfiable constraint η such that $\eta \Rightarrow \phi$ and $\eta \Rightarrow \eta\theta$ are valid: for a substitution σ' such that $\eta\sigma'$ is valid, the validity of $\eta \Rightarrow \eta\theta$ implies the validity of $\eta\theta\sigma'$, $\eta\theta\theta\sigma', \cdots$ and the validity of $\eta \Rightarrow \phi$ implies the validity of $\phi\theta\sigma'$, $\phi\theta\theta\sigma', \cdots$.

This paper shows a sufficient condition for non-termination of constrained TRSs, following the above observation. We also show a heuristics to find extra constraints (such as η above) for constrained rules of the form $t \to C[t\theta]$ $[\![\phi]\!]$ if the rules cause non-termination. For the sake of simplicity, we discuss on the simpler class than the class of CESs. Though, we believe that the results in this paper holds for more complicated classes, such as CESs.

We assume familiarity with the basic concepts and notations of term rewriting systems (TRSs, for short) [3, 19], and first-order predicate logic [16].

2 Constrained Term Rewriting Systems

In this section, we recall the definition of constrained term rewriting systems [18, 4, 5, 12, 20].

Let $\mathcal{G}$ be a signature such that $\mathcal{G}$ has at least a constant (i.e., $\mathcal{T}(\mathcal{G}) \neq \emptyset$), and $\mathcal{P}$ be a finite set of *predicate* symbols. We sometimes denote $\neg\phi_1 \vee \phi_2$ by $\phi_1 \Rightarrow \phi_2$ as usual. Let $\mathcal{M}$ be a *structure* for formulas over $(\mathcal{G}, \mathcal{P}, \mathcal{V})$, where the interpretation of closed formulas ϕ by $\mathcal{M}$, denoted by $\mathcal{M} \models \phi$, is defined as usual. We suppose that for every element a in the universe of $\mathcal{M}$, there exists a term $t \in \mathcal{T}(\mathcal{G})$ such that $(t)^{\mathcal{M}} = a$. The *validity* and *satisfiability* of formulas are defined as usual. When $\mathcal{G}$, $\mathcal{P}$, and $\mathcal{M}$ are fixed in context, we may call formulas over $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ *constraints* (w.r.t. $\mathcal{M}$). We assume that for each structure we use, there exist decision algorithms for the validity (and satisfiability) w.r.t. the structure. For example, an algorithm for the truth value of closed (quantified) formulas for linear integer arithmetics is well-known [8], that can be used for deciding the validity and satisfiability. This assumption is usual in recent frameworks of constrained systems [1, 10].

Let $\mathcal{G}$ and $\mathcal{F}$ be signatures such that $\mathcal{F} \cap \mathcal{G} = \emptyset$ and $\mathcal{G}$ contains at least a constant, $\mathcal{P}$ be a finite set of predicate symbols, and $\mathcal{M}$ be a structure for $(\mathcal{G}, \mathcal{P})$. A *constrained rewrite rule over* $(\mathcal{F}, \mathcal{G}, \mathcal{P}, \mathcal{V}, \mathcal{M})$ is a triple (l, r, ϕ), written as $l \to r$ $[\![\phi]\!]$, such that l and r are terms in $\mathcal{T}(\mathcal{F} \cup \mathcal{G}, \mathcal{V})$, l is not a variable, ϕ is a quantifier-free constraint w.r.t. $\mathcal{M}$ (i.e., a formula over $(\mathcal{G}, \mathcal{P}, \mathcal{V})$), and $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r) \cup \mathrm{fv}(\phi)$, and ϕ is satisfiable w.r.t. $\mathcal{M}$. We may write $l \to r$ instead of $l \to r$ $[\![\top]\!]$. A *constrained term rewriting system* (constrained TRS, for short) *over* $(\mathcal{F}, \mathcal{G}, \mathcal{P}, \mathcal{V}, \mathcal{M})$ is a finite set of constrained rewrite rules over $(\mathcal{F}, \mathcal{G}, \mathcal{P}, \mathcal{V}, \mathcal{M})$. The *rewrite relation* $\to_R$ is defined by $\{(C[l\sigma]_p, C[r\sigma]_p) \mid l \to r\ [\![\phi]\!] \in R,\ \mathcal{R}an(\sigma|_{\mathrm{fv}(\phi)}) \subseteq \mathcal{T}(\mathcal{G}, \mathcal{V}),\ \phi\sigma$ is valid w.r.t. $\mathcal{M}\}$.

Example 1. Let a signature $\mathcal{G}_{PA} = \{\mathsf{0}, \mathsf{s}(), \mathsf{p}(), \mathsf{plus}(,)\}$, a predicate set $\mathcal{P}_{PA} = \{\succ(,)\}$, and $\mathcal{M}_{PA}$ be a structure for $(\mathcal{G}_{PA}, \mathcal{P}_{PA})$ such that the universe of $\mathcal{M}_{PA}$ is the set $\mathbb{Z}$ of integers, $\mathsf{0}^{\mathcal{M}_{PA}} = 0$, $\mathsf{s}^{\mathcal{M}_{PA}}(x) = x+1$, $\mathsf{p}^{\mathcal{M}_{PA}}(x) = x-1$, $\mathsf{plus}^{\mathcal{M}_{PA}}(x_1, x_2) = x_1 + x_2$, and $\succ^{\mathcal{M}_{PA}} = \{(m, n) \mid m, n \in \mathbb{Z},\ m > n\}$. The constrained TRS R_{sum} in Section 1 is over $(\{\mathsf{sum}(), \mathsf{eval}(,,)\}, \mathcal{G}_{PA}, \mathcal{P}_{PA}, \mathcal{V}, \mathcal{M}_{PA})$.

In contrast to *conditional TRSs* [19], any constrained rewrite rule is not used for evaluating the truth value of instantiated constraints in the rewrite relation. For this reason, the termination is different from *operational termination* of conditional TRSs [17], that is finiteness of derivation trees.

3 Sufficient Condition for Non-Termination of Constrained TRSs

Throughout this section, we assume that R is a constrained TRS over $(\mathcal{F}, \mathcal{G}, \mathcal{P}, \mathcal{V}, \mathcal{M})$. Before showing the main result based on the observation in Section 1, we start with adapting a well-known sufficient condition for non-termination of unconstrained TRSs to constrained systems. The well-known sufficient condition is the detection of the reduction $t \to^+ C[t\theta]$. If the range of θ is over terms in $\mathcal{T}(\mathcal{G}, \mathcal{V})$, then this is also a sufficient condition of non-termination for constrained TRSs because the reduction is closed under contexts and substitutions whose ranges are over terms in $\mathcal{T}(\mathcal{G}, \mathcal{V})$: if there exists a reduction $t \to_R^+ C[t\theta]$ such that $\mathcal{R}an(\theta) \subseteq \mathcal{T}(\mathcal{G}, \mathcal{V})$, then R is not terminating.

A simple way to find the reduction $t \to^+ C[t\theta]$ is to find a rewrite rule of the form $t \to C[t\theta]$. In contrast, as shown in Section 1, this approach is not useful for the case of constrained TRSs and thus we have to find extra constraints η (as shown in Section 1) for constrained rules $t \to C[t\theta]$ $[\![\phi]\!]$ that cause non-termination. The role of η is to narrow down the original constraint ϕ.

Theorem 1. *R is not terminating if there exist a constrained rewrite rule $t \to C[t\theta]$ $[\![\phi]\!] \in R$ and a constraint η such that $\mathscr{R}an(\theta|_{\mathsf{fv}(\phi)}) \subseteq \mathscr{T}(\mathscr{G},\mathscr{V})$, η is satisfiable w.r.t. $\mathscr{M}$, and $\eta \Rightarrow \eta\theta$ and $\eta \Rightarrow \phi$ are valid w.r.t. $\mathscr{M}$.*

As the well-known sufficient condition is based on the detection of a reduction $t \to^+ C[t\theta]$, we would like to extend Theorem 1 so as to be based on the detection of reductions that cause non-termination. However, this seems impossible because constraints such as ϕ and η cannot be specified in the reduction sequences. So as to specify constraints in reduction sequences, we review the reduction under constraints. Let ψ be a constraint that is satisfiable w.r.t. $\mathscr{M}$. Then, the *constrained reduction $\to_{\psi,R}$ under ψ* is defined as follows [12, 5]: $\to_{\psi,R} = \{\, (C[l\sigma],C[r\sigma]) \mid l \to r$ $[\![\phi]\!] \in R,\ \psi \Rightarrow \phi\sigma$ is valid w.r.t. $\mathscr{M}\,\}$. Note that $\to_{\psi,R}$ and $\to_R$ are not identical. It is clear that $s \to_{\psi,R} t$ implies $s\theta \to_R t\theta$ for any substitution θ such that $\psi\theta$ is valid w.r.t. $\mathscr{M}$. Thus, to show non-termination of $\to_R$, it suffices to show non-termination of $\to_{\psi,R}$ for some ψ. Theorem 1 is extended to constrained reductions as follows.

Theorem 2. *Let η be a constraint. Then, there exists an infinite derivation of $\to_{\eta,R}$ (i.e., R is not terminating) if there exist a term t, a context $C[\ ]$, and a substitution θ such that $\mathscr{R}an(\theta|_{\mathsf{fv}(\eta)}) \subseteq \mathscr{T}(\mathscr{G},\mathscr{V})$, $t \to^+_{\eta,R} C[t\theta]$, and $\eta \Rightarrow \eta\theta$ is valid w.r.t. $\mathscr{M}$.*

A difficulty of automating the method based on Theorem 1 is how to find extra constraints η. However, the constraints of rules sometimes provide hints to find the constraints η. For the example $\mathsf{eval}(x,i,z) \to \mathsf{eval}(x,\mathsf{s}(i),\mathsf{plus}(z,\mathsf{s}(i)))$ $[\![x \succ i \vee i \succ x]\!] \in R_{\mathsf{sum}}$ in Section 1, we used the constraint $i \succ x$ as η. This constraint $i \succ x$ is a closure of the original constraint $x \succ i \vee i \succ x$ of the rule. The following theorem incorporates a heuristics to find the extra constraints η.

Theorem 3. *Let ϕ be a constraint and θ be a substitution with $\mathscr{R}an(\theta) \subseteq \mathscr{T}(\mathscr{G},\mathscr{V})$, $\eta_1 \vee \cdots \vee \eta_n$ be a disjunctive normal form (DNF, for short) of ϕ, and (V,E) be a directed graph such that $V = \{\eta_i \mid \eta_i$ is satisfiable w.r.t. $\mathscr{M}\}$ and $E = \{(\eta_i,\eta_j) \mid \eta_i,\eta_j \in V,\ \eta_i \Rightarrow \eta_j\theta)$ is valid w.r.t. $\mathscr{M}\}$, and a path $\eta'_1,\cdots,\eta'_k$ of nodes ($k > 0$) be a* cycle *in (V,E). Then, $\eta'_1 \vee \cdots \vee \eta'_k$ is satisfiable w.r.t. $\mathscr{M}$, and $(\eta'_1 \vee \cdots \vee \eta'_k) \Rightarrow \phi$ and $(\eta'_1 \vee \cdots \vee \eta'_k) \Rightarrow (\eta'_1 \vee \cdots \vee \eta'_k)\theta$ are valid w.r.t. $\mathscr{M}$.*

Since $\eta'_1,\ldots,\eta'_k$ are closures of the DNF of ϕ, the range of valuations satisfying the constraint $\eta'_1 \vee \cdots \vee \eta'_k$ obtained from the cycle is narrower than the range of ϕ, and $\eta'_1 \vee \cdots \vee \eta'_k$ plays a role of η in Theorem 1. This is the reason why DNFs are used.

Example 2. Consider the constraint $x \succ i \vee i \succ x$ again. Let ϕ be $x \succ i \vee i \succ x$ and θ be a substitution $\{i \mapsto \mathsf{s}(i), z \mapsto \mathsf{plus}(z,\mathsf{s}(i))\}$. Then, neither $x \succ i \Rightarrow x \succ \mathsf{s}(i)$, $x \succ i \Rightarrow \mathsf{s}(i) \succ x$, nor $i \succ x \Rightarrow x \succ \mathsf{s}(i)$ is valid but $i \succ x \Rightarrow \mathsf{s}(i) \succ x$ is valid. Thus, we have the graph $(\{i \succ x\},\{(i \succ x, i \succ x)\})$, that has a cycle.

Finally, we obtain the following sufficient condition for non-termination.

Theorem 4. *R is not terminating if there exist a rule $l \to C[l\theta]$ $[\![\phi]\!] \in R$ such that $\mathscr{R}an(\theta|_{\mathsf{fv}(\phi)}) \subseteq \mathscr{T}(\mathscr{G},\mathscr{V})$, and for a DNF $\eta_1 \vee \cdots \vee \eta_n$ of ϕ, the directed graph $(\{\eta_i \mid \eta_i$ is satisfiable w.r.t. $\mathscr{M}\},\{(\eta_i,\eta_j) \mid \eta_i,\eta_j \in V,\ \eta_i \Rightarrow \eta_j\theta$ is valid w.r.t. $\mathscr{M}\})$ has a cycle.*

Note that the sufficient condition in Theorem 4 is decidable if the validity of constraints over $(\mathscr{G},\mathscr{P},\mathscr{V})$ is decidable. When $n = 1$ in Theorem 4, it suffices to check the validity of $\eta_1 \Rightarrow \eta_1\theta$.

Example 3. The directed graph constructed from the rule $\mathsf{eval}(x,i,z) \to \mathsf{eval}(x,\mathsf{s}(i),\mathsf{plus}(z,\mathsf{s}(i)))$ $[\![x \succ i \vee i \succ x]\!] \in R_{\mathsf{sum}}$ in Section 1 has a cycle. Therefore, by Theorem 4, R_{sum} is not terminating.

The method in this paper to disprove termination analyzes single constrained rules of the form $l \to C[l\theta]\ [\![\psi]\!]$. For this reason, the method is not so powerful while the sufficient condition in Theorem 2 is general. To make the method more practical, we need to improve it. Moreover, we have to experiment for many non-terminating constrained TRSs in order to verify the usefulness of the method.

References

[1] A. Armando, M. Rusinowitch, and S. Stratulat. Incorporating decision procedures in implicit induction. *J. Symb. Comput.*, 34(4):241–258, 2002.

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

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

[4] A. Bouhoula and F. Jacquemard. Automated induction for complex data structures. Research Report LSV-05-11, Laboratoire Spécification et Vérification, ENS Cachan, France, July 2005. 24 pages.

[5] A. Bouhoula and F. Jacquemard. Automated induction with constrained tree automata. In *Proc. of IJCAR '08*, vol. 5195 of *LNCS*, pp. 539–554, Springer, 2008.

[6] A. Bouhoula and M. Rusinowitch. Implicit induction in conditional theories. *J. Autom. Reasoning*, 14(2):189–235, 1995.

[7] H. Comon. Completion of rewrite systems with membership constraints. part I: Deduction rules, part II: Constraint solving. *J. Symb. Comput.*, 25(4):397–419, 421–453, 1998.

[8] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for dpll(t). In T. Ball and R. B. Jones, editors, *Proc. of CAV 2006*, vol. 4144 of *LNCS*, pp. 81–94, Springer, 2006.

[9] S. Falke and D. Kapur. Inductive decidability using implicit induction. In *Proc. of LPAR '06*, *LNAI*, pp. 45–59, Springer, 2006.

[10] S. Falke and D. Kapur. Dependency pairs for rewriting with built-in numbers and semantic data structures. In *Proc. of RTA 2008*, vol. 5117 of *LNCS*, pp. 94–109, Springer, 2008.

[11] S. Falke and D. Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In *Proc. of CADE-22*, vol. 5663 of *LNCS*, pp. 277–293, Springer, 2009.

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

[13] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proc. of IJCAR 2006*, vol. 4130 of *LNCS*, pp. 281–286, Springer, 2006.

[14] J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. of LPAR '04*, vol. 3452 of *LNCS*, pp. 301–331, Springer, 2004.

[15] C. Hoot. Completion for constrained term rewriting systems. In *Proc. of CTRS-92*, vol. 656 of *LNCS*, pp. 408–423, Springer, 1992.

[16] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2000.

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

[18] C. Lynch and W. Snyder. Redundancy criteria for constrained completion. *Theor. Comput. Sci.*, 142(2):141–177, 1995.

[19] E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer-Verlag, 2002.

[20] T. Sakata, N. Nishida, T. Sakabe, M. Sakai, and K. Kusakari. Rewriting induction for constrained term rewriting systems. *IPSJ Transactions on Programming*, 2(2):80–96, Mar. 2009 (in Japanese).

[21] Y. Toyama. Confluent term rewriting systems with membership conditions. In *Proc. of CTRS-87*, vol. 308 of *LNCS*, pp. 228–241, Springer, 1987.

[22] I. Wehrman, A. Stump, and E. M. Westbrook. Slothrop: Knuth-Bendix completion with a modern termination checker. In *Proc. of RTA 2006*, vol. 4098 of *LNCS*, pp. 287–296, Springer, 2006.

A Proof of Theorem 1

Proof. It follows from the assumption that $t \to_{\eta,R} C[t\theta]$ and $\eta \Rightarrow \eta\theta$ is valid w.r.t. $\mathscr{M}$. Thus, it follows from Theorem 2 that R is not terminating. $\square$

B Proof of Theorem 2

Proof. We first show that if $s \to^{n}_{\eta,R} t$, then $s\theta \to^{n}_{\eta,R} t\theta$ for any substitution θ such that $\eta\theta$ is satisfiable w.r.t. $\mathscr{M}$ and $\eta \Rightarrow \eta\theta$ is valid w.r.t. $\mathscr{M}$. We prove this claim by induction on n. Since the case that $n = 0$ is clear, we only consider the remaining case that $n > 0$. Suppose that $s \equiv C[l\sigma] \to_{\eta,R} C[r\sigma] \to^{n-1}_{\eta,R} t$ where $l \to r$ $[\![\phi]\!] \in R$ and $\eta \Rightarrow \phi\sigma$ is valid w.r.t. $\mathscr{M}$. Then, $\eta\theta \Rightarrow \phi\sigma\theta$ is also valid because of the validity of $\eta \Rightarrow \phi\sigma$. It follows from the validity of $\eta \Rightarrow \phi\theta$ and $\eta\theta \Rightarrow \eta\sigma\theta$ that $\eta \Rightarrow \eta\sigma\theta$ is also valid. Thus, we have $(C[l\sigma])\theta \to_{\eta,R} (C[r\sigma])\theta$. By the induction hypothesis, we have $(C[r\sigma])\theta \to^{n-1}_{\eta,R} t\theta$. Therefore, we have $s\theta \to^{n}_{\eta,R} t\theta$.

Next, we prove the theorem. Suppose that $t \to^{+}_{\eta,R} C[t\theta]$. Then, by definition, η is satisfiable w.r.t. $\mathscr{M}$. Due to the above claim, we have that $t\theta \to^{+}_{\eta,R} (C[t\theta])\theta$, $t\theta\theta \to^{+}_{\eta,R} (C[t\theta])\theta\theta, \cdots$. Thus, we have an infinite derivation $t \to^{+}_{\eta,R} C[t\theta] \to^{+}_{\eta,R} C[(C[t\theta])\theta] \equiv C[C\theta[t\theta\theta]] \to^{+}_{\eta,R} C[C\theta[(C[t\theta])\theta\theta]] \to^{+}_{\eta,R} \cdots$. Since η is satisfiable, there exists a substitution δ such that $\mathrm{fv}(\eta) \subseteq \mathscr{D}om(\delta)$, $\mathscr{R}an(\delta|_{\mathrm{fv}(\eta)}) \subseteq \mathscr{T}(\mathscr{G})$, and $\mathscr{M} \models \eta\delta$. Thus, we have an infinite derivation $t\delta \to^{+}_{R} (C[t\theta])\delta \to^{+}_{R} (C[C\theta[t\theta\theta]])\delta \to^{+}_{R} (C[C\theta[(C[t\theta])\theta\theta]])\delta \to^{+}_{R} \cdots$. Therefore, R is not terminating. $\square$

C Proof of Theorem 3

Proof. By definition, $\eta'_1, \ldots, \eta'_k$ ($k > 0$) are satisfiable w.r.t. $\mathscr{M}$. Since $\eta'_1 \vee \cdots \vee \eta'_k$ is a disjunction of some closures in a DNF of ϕ, $(\eta'_1 \vee \cdots \vee \eta'_k) \Rightarrow \phi$ is valid w.r.t. $\mathscr{M}$. It follows from the construction of E and the definition of cycles that $\eta'_i \Rightarrow \eta'_{i+1}\theta$ is valid w.r.t. $\mathscr{M}$ for $1 \leq i \leq k$, where we consider $i+1$ as 1 if $i = k$. Thus, $(\eta'_1 \vee \cdots \vee \eta'_k) \Rightarrow (\eta'_1 \vee \cdots \vee \eta'_k)\theta$ is also valid w.r.t. $\mathscr{M}$. Therefore, this theorem holds. $\square$

A Purely Logical Approach to Program Termination

EXTENDED ABSTRACT

Mădălina Eraşcu* Tudor Jebelean
Research Institute for Symbolic Computation
Johannes Kepler University, Linz, Austria
{merascu,tjebelea}@risc.uni-linz.ac.at

We present our work in progress concerning the logical foundations of the analysis of termination for imperative recursive programs. The analysis is based on forward symbolic execution [13] and functional semantics. The distinctive feature of our approach is the formulation of *the termination condition as an induction principle* developed from the structure of the program with respect to iterative structures (recursive calls and `while` loops). Moreover the termination condition insures the existence and the uniqueness of the function implemented by the program. Note that the existence is not automatic, because a recursive program corresponds, logically, to an implicit definition. It is interesting that this inductive termination condition can be also used for proving the uniqueness of the function as well as the total correctness of the program. We show in this paper how to prove the existence of the implemented function in the case of single recursion programs (programs with at most one recursive call on each branch). The method can be applied however to all imperative recursive programs, where recursive calls are outside the loops. For other programs, termination analysis appears to involve co-recursive functions and it is subject to further investigation. The methods presented here are under implementation in the *Theorema* system [3].

Related work. Existing static analysis methods in the Floyd-Hoare style [8, 11] for proving termination of programs with loops consist in manually annotate the loop with a termination term [10], or to synthesize the termination term automatically using various techniques mostly from linear [integer] programming [15, 1, 2]. These approaches can be seen in the context of our work as methods for proving certain classes of such logically expressed termination conditions that we generate. A recent approach for termination of functional programs is based on the comparison of infinite paths in the control flow graph and in „size-change graphs", comparison that is reduced to the inclusion test for Büchi automata. Automated tools supporting termination analysis are e.g. Terminator [4, 9], ACL2 [12], and termination tools for term rewriting systems (`http://rewriting.loria.fr/`).

Theoretical background. Our approach is purely logic, meaning that the program correctness is provable in predicate logic, without using any additional theoretical model for program semantics or program execution, but only using the theories relevant to the predicates, constants and functions present in the program text. (By a *theory* we understand a set of formulae in the language of predicate logic with equality.) We call such theories Υ *object theories*. We consider two kinds of functions in the object theory: *i*) *basic* – they have only input conditions, but no output conditions (e.g. arithmetic operations in various number domains); and *ii*) *additional* – they are functions implemented by other programs or bounded arithmetical operations, and in the process of verification conditions generation only their specification will be used.

Additionally, we consider a *meta-theory* which includes the programming constructs (statements), the program itself, as well as the terms and the formulae from the object theory, which are *meta–terms* from the point of view of the meta-theory, and they behave like *quoted*. The programming statements are: abrupt statements (`break, return`), assignments – including recursive calls, conditionals and `while`

*Supported by Upper Austrian Government and Austrian Science Foundation (FWF) grant W1214–DK1

loops. The statements contain formulae and terms from the object theory. A program P is a tuple of statements and is annotated with pre- and postcondition, the logic formulae $I_f[\alpha]$ and $O_f[\alpha,\beta]$, respectively. It takes as input a certain number of variables and it returns a single value β. For simplicity we consider a single input variable (denoted conventionally by α). (Clearly this formalism can be easily extended to several input and output variables).

The meta-theory contains further constructs for reasoning about programs: The meta-predicate Π checks that a program is syntactically correct, that every branch contains a `return` statement, that `break` statement occurs only inside loops, and that each variable is initialized before it is used. The meta-level function Σ creates an object-level formula containing a new [second order] symbol f denoting conventionally the function defined by the program. It generates a conjunction of formulae with the shape:

$$\underset{\alpha:I_f}{\forall} \Phi \Rightarrow (f[\alpha]=t),$$

having the following meaning: the expression for f is the symbolic term t, conditioned by the object-level formula Φ – the accumulated conditions coming from the analysis of each statement on the respective path. This formula is universally quantified over the input variable α satisfying the input condition I_f of the program. (Please note that in this paper we depart from the classical notation for application of functions and predicates, in that we use the square brackets instead of the usual round brackets.) We consider this formula as being the *semantics* of the program P in the following sense: the function f implemented by the program satisfies $\Sigma[P]$, in other words $\Sigma[P]$ is the implicit definition of the function f.

Note that Σ effectively translates the original program into a *functional* program. From this point on, one could reason about the program using e.g. the Scott fixpoint theory ([14], pag. 86), however we prefer a purely logical approach.

The meta-level function Γ generates two kinds of verification conditions insuring the partial correctness of the program. *Safety conditions* are formulae with the shape $\Phi \Rightarrow I_h[t]$, where I_h is the input condition of some function h called with the current symbolic value t, and the formula Φ accumulates the conditions on the respective branch. *Functional conditions* are formulae checking that the output condition on the currently returned value is a consequence of the accumulated conditions on the respective branch.

Finally, the meta-level function Θ generates a termination condition for the recursive program and for each `while` loop.

The detailed formalization of the predicate and meta-functions are presented in [5, 6, 7].

Single recursion programs. We present in this paper the main meta-theorems concerning the programs which have *at most one recursive call on each branch*. It is quite straightforward to show that such programs can always be expressed as in (1), where Q is a predicate and S, C, and R are functions defined using the constructs present in the program text, possibly using conditionals but no recursion.

$$P: \quad f[\alpha] = \quad \underline{\texttt{if}}\ Q[\alpha]\ \ \underline{\texttt{then}}\ \ S[\alpha]\ \ \underline{\texttt{else}}\ C[\alpha, f[R[\alpha]]] \tag{1}$$

The semantics formula $\Sigma[P]$ and termination condition $\Theta[P]$ for such a program are (2) and (3), respectively.

$$\underset{\alpha:I_f}{\forall} \wedge \begin{cases} Q[\alpha] \Rightarrow (f[\alpha]=S[\alpha]) \\ \neg Q[\alpha] \Rightarrow (f[\alpha]=C[\alpha,f[R[\alpha]]]) \end{cases} \tag{2}$$

$$\underset{\alpha:I_f}{\forall} \wedge \left. \begin{cases} Q[\alpha] \Rightarrow \pi[\alpha] \\ \neg Q[\alpha] \wedge \pi[R[\alpha]] \Rightarrow \pi[\alpha] \end{cases} \right\} \Rightarrow \underset{\alpha:I_f}{\forall} \pi[\alpha] \tag{3}$$

(We use as notations: $x : I_f$ for "x satisfying $I_f[x]$" and $\wedge$ with curly brackets for conjunction of several formulae.) Note that both formulae are at object level. Also, (2) is an implicit definition of f.

In formula (3), π is a *new constant symbol*, thus in fact it behaves like a universally quantified predicate. This is why this formula is in fact an induction principle. Note also that the termination condition abstracts some details of the actual program (functions S and C), because they are in fact not important for termination.

The rationale of this formula is as follows: The left-hand side of the implications represents a property which should be fulfilled by the predicate $\pi[\alpha]$ (*"the program terminates on input α"*), property which, in case of recursive calls includes also the predicate $\pi[R[\alpha]]$ – that is the arbitrary predicate applied to the current symbolic values of the arguments of the recursive call to f. However there may be many predicates which have this property (for instance "True"). Intuitively, we consider that the predicate expressing termination is the strongest predicate obeying this property. Since the new constant π behaves like a universally quantified (second order) variable, the formula states that the input condition I_f is stronger than any such predicate, thus it is stronger than the termination predicate. Therefore, the program terminates for any values of the input variable which fulfills the input condition.

Correctness of the method. The total correctness formula for single recursion programs is expressed as: "The formula $\underset{\alpha}{\forall} I_f[\alpha] \Rightarrow O_f[\alpha, f[\alpha]]$ is a logical consequence of the object theory augmented with $\Sigma[P]$ and with the verification conditions." However, this always holds in the case that $\Sigma[P]$ is contradictory to the theory, which may happen when the program is recursive. Therefore, one proves first that the existence (and the uniqueness) of a f satisfying $\Sigma[P]$ is a logical consequence of the object theory augmented with the verification conditions. This follows from the termination condition.

We give now the main steps of the development leading to this fact. Please note that we use n, m as natural numbers, and n^+ for the successor function.

Lemma 1. *(Existence of the repetition function.) The formula*

$$\underset{h}{\forall}\, \underset{G}{\exists}\, \underset{x}{\forall} \left(G[0,x] = x \,\wedge\, \underset{n:\mathbb{N}}{\forall}(G[n^+,x] = h[G[n,x]])\right)$$

is a logical consequence of the natural number theory.

Proof. Let x be arbitrary but fixed.

One proves first $\underset{m:\mathbb{N}}{\forall}\, \underset{H}{\exists} \left(H[0] = x \,\wedge\, \underset{n<m}{\forall} H[n^+] = h[H[n]]\right)$ by natural induction on m. From here by Skolemization on H one obtains $\underset{\mathscr{H}}{\exists}\, \underset{m:\mathbb{N}}{\forall} \left(\mathscr{H}[m][0] = x \wedge \underset{n<m}{\forall} \mathscr{H}[m][n^+] = h[\mathscr{H}[m][n]]\right)$. Furthermore one can prove $\underset{n:\mathbb{N}}{\forall}\, \underset{m \geq n}{\forall} \mathscr{H}[m][n] = \mathscr{H}[n][n]$ by natural induction on n and by taking $g[n] = \mathscr{H}[n][n]$ one has (since x was arbitrary) $\underset{x}{\forall}\, \underset{g}{\exists} \left(g[0] = x \wedge \underset{n:\mathbb{N}}{\forall} g[n^+] = h[g[n]]\right)$ which by Skolemization on g gives the desired formula (with notation $G[n,x]$ instead of $G[x][n]$). □

Remark 1. The function $G[n,x]$ is usually denoted as $h^n[x]$.
Remark 2. It is straightforward to show that $h^n[h[x]] = h^{n^+}[x]$.

The subsequent properties need the theory of natural numbers, although we do not specify this explicitely.

Lemma 2. *(Existence of the recursion index.) The formula $\underset{\alpha:I_f}{\forall}\, \underset{n:\mathbb{N}}{\exists} Q[R^n[\alpha]]$ is a logical consequence of the termination condition* (3) *and the safety verification conditions.*

Proof. The proof uses the induction principle given in (3), where $\pi[\alpha]$ is $\underset{n:\mathbb{N}}{\exists} Q[R^n[\alpha]]$. One needs to use the safety conditions and the property of h^n given above. □

Remark 1. One can define now a function (the recursion index of α) $M[\alpha] = \min\{n \mid Q[R^n[\alpha]\}$ because the set is nonempty.
Remark 2. It is straightforward to show that $M[R[\alpha]]^+ = M[\alpha]$.

Theorem 1. *(Existence of the function implemented by the program.) The formula* (2) *is a logical consequence of the termination condition* (3) *and the safety verification conditions.*

Proof. The proof is similar to the one from *Lemma 1*, only that instead of the running argument n we use α with a certain recursion index.

One proves first:

$$\underset{m:\mathbb{N}}{\forall} \underset{F}{\exists} \underset{\alpha:I_f}{\forall} (M[\alpha] \leq m) \Rightarrow \big((Q[\alpha] \Rightarrow F[\alpha] = S[\alpha]) \wedge (\neg Q[\alpha] \Rightarrow F[\alpha] = C[\alpha, F[R[\alpha]]])\big) \tag{4}$$

by natural induction on m.

By Skolemizing F from (4) one obtains:

$$\underset{\mathscr{F}}{\exists} \underset{m:\mathbb{N}}{\forall} \underset{\alpha:I_f}{\forall} (M[\alpha] \leq m) \Rightarrow \big((Q[\alpha] \Rightarrow \mathscr{F}[m][\alpha] = S[\alpha]) \wedge (\neg Q[\alpha] \Rightarrow \mathscr{F}[m][\alpha] = C[\alpha, \mathscr{F}[m][R[\alpha]]])\big)$$

Furthermore one can prove $\underset{\alpha:I_f}{\forall} \underset{m:\mathbb{N}}{\forall} (m \geq M[\alpha]) \Rightarrow (\mathscr{F}[m][\alpha] = \mathscr{F}[M[\alpha]][n])$ by the induction given in the formula (3) (taking as $\pi[\alpha]$ the formula above without the quantifier for α).

Finally one takes $f[\alpha] = \mathscr{F}[M[\alpha]][\alpha]$. □

Remark. Uniqueness of f is straightforward: take f_1, f_2 satisfying (2) and use (3) with $\pi[\alpha]$ as $f_1[\alpha] = f_2[\alpha]$.

Theorem 2. *(Total correctness.) The formula* $\underset{\alpha}{\forall} I_f[\alpha] \Rightarrow O_f[\alpha, f[\alpha]]$ *is a logical consequence of the object theory augmented with the program semantics and the verification conditions.*

Proof. The proof is straightforward by taking in (3) $\pi[\alpha]$ as $O_f[\alpha, f[\alpha]]$. This is because the left-hand side of the (3) becomes identical to the functional conditions generated for partial correctness. □

Programs containing `while` loops. Our approach can be easily extended to programs containing arbitrarily nested possibly abrupt terminating `while` loops, by transforming the loop body B into a tail recursive function where the loop invariant ι represents the specification of the new function. The parameters of the virtual function are the so called *critical variables* – the variables which are modified within the loop body. The actual arguments of the call are the values of the critical variables when entering the loop. The template semantics and termination formulae are similar to those corresponding to primitive recursive functions, namely if the `while` loop is (5) then the semantics is (6) and the termination condition is (7).

$$W: \quad \underline{\texttt{while}}\ \varphi[\delta]\ \underline{\texttt{do}}\ \iota, B \tag{5}$$

$$\underset{\delta:\iota}{\forall} \wedge \begin{cases} \neg\varphi[\delta] \Rightarrow (f[\delta] = \delta) \\ \varphi[\delta] \Rightarrow (f[\delta] = f[R[\delta]]) \end{cases} \tag{6}$$

$$\underset{\delta:\iota}{\forall} \wedge \left\{ \begin{array}{l} \neg\varphi[\delta] \Rightarrow \pi[\delta] \\ \varphi[\delta] \wedge \pi[R[\delta]] \Rightarrow \pi[\delta] \end{array} \right\} \Rightarrow \underset{\delta:\iota}{\forall} \pi[\delta] \tag{7}$$

In case of nested loops, the operations from the body of the inner loops are not reflected explicitly in the clauses of the semantics and verification conditions, except if a `return` – at all levels or a `break`– in non-nested loops is encountered. It is the loop invariant that encodes them and that it is used in the further analysis of the program. The details of the formalization are found in [7].

The correctness proof proceeds similar to the case of primitive recursive functions, the only difference being the definition of the loops semantics on the recursive branch.

Conclusions and future work. During the construction of a mathematical theory, when we define a new function in an implicit way, we prove the existence of it in order to avoid introducing a contradiction in our theory. Likewise, during the construction of a software system, when we program a new function, we prove the termination of it, in order to insure the effectiveness of our system. The work presented here is a step towards showing that these two situations essentially reduce to the same logical operations.

Further work includes the treatment of programs with loops that contain abrupt termination and recursive calls, investigation of termination theory for programs with multiple recursion and with nested recursion, as well as the development of methods for proving the verification conditions by combining logical and algebraic algorithms.

References

[1] A. Bradley, Z. Manna, and H. Sipma, *Linear Ranking with Reachability*, Proc. 17^{th} Intl. Conference on Computer Aided Verification (K. Etessami and S. Rajamani, eds.), Lecture Notes in Computer Science, vol. 3576, Springer Verlag, July 2005.

[2] A. Bradley, Z. Manna, and H. Sipma, *Termination Analysis of Integer Linear Loops*, CONCUR 2005 - Concurrency Theory, Springer-Verlag, London, UK, 2005, pp. 488–502.

[3] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger, *Theorema: Towards Computer-Aided Mathematical Theory Exploration*, Journal of Applied Logic **4** (2006), no. 4, 470–504.

[4] B. Cook, A. Podelski, and A. Rybalchenko, *CFL-Termination*, Tech. Report MSR-TR-2008-160, Microsoft Research, 2008.

[5] M. Eraşcu and T. Jebelean, *Practical Program Verification by Forward Symbolic Execution: Correctness and Examples*, Austrian-Japan Workshop on Symbolic Computation in Software Science (B. Buchberger, T. Ida, and T. Kutsia, eds.), 2008, pp. 47–56.

[6] M. Eraşcu and T. Jebelean, *A Calculus for Imperative Programs: Formalization and Implementation*, Proceedings of the 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (S. Watt, V. Negru, T. Ida, T. Jebelean, D. Petcu, and D. Zaharie, eds.), IEEE, 2009, pp. 77– 84.

[7] M. Eraşcu and T. Jebelean, *A Purely Logical Approach to Imperative Program Verification*, Tech. Report 10-07, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, 2010.

[8] R. Floyd, *Assigning Meaning to Programs*, Proc. of Symposia in Appl. Math. American Mathematical Society, 1967.

[9] A. Gotsman, B. Cook, M. Parkinson, and V. Vafeiadis, *Proving that Non-blocking Algorithms don't Block*, POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), ACM, 2009, pp. 16–28.

[10] D. Gries, *The Science of Programming*, Springer, 1981.

[11] C. A. R. Hoare, *An Axiomatic Basis for Computer Programming*, Communications of the ACM **12** (1969), no. 10, 576–580.

[12] M. Kaufmann, J. Strother Moore, and P. Manolios, *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[13] J. King, *Symbolic Execution and Program Testing*, Communications of the ACM **19** (1976), no. 7, 385–394.

[14] J. Loeckx, K. Sieber, and R. Stansifer, *The Foundations of Program Verification*, John Wiley & Sons, Inc., New York, NY, USA, 1984.

[15] A. Podelski and A. Rybalchenko, *A Complete Method for the Synthesis of Linear Ranking Functions*, VMCAI, 2004, pp. 239–251.

Non-termination analysis of Logic Programs using Types

Dean Voets[*]
Dean.Voets@cs.kuleuven.be

Danny De Schreye
Danny.DeSchreye@cs.kuleuven.be

1 Introduction

In the past five years, techniques have been developed to analyze the non-termination – as opposed to termination – of logic programs. The main motivation for this work is to provide precision results for termination analyzers. The first and most well-known non-termination analyzer is *NTI* ([5]). Recently, we developed a slightly more precise analyzer, *P2P*, presented in [8].

Both non-termination analyzers contain two phases in their analysis. In a first phase, they compute a suitable, finite approximation of all the computations for the considered class of queries. In a second phase, the non-termination analyzers analyze these finite approximations of the computations, to detect loops. They prove that these loops correspond to infinite derivations in concrete computations of the program for the considered queries.

In *NTI*, the finite approximation of the computations are binary unfoldings of the given program. Detecting loops is based on a special more general than relation, called Δ-*more general*.

In [8], the finite approximations of the computations is the *moded SLD-tree*. This concept has been introduced in [7], in the context of termination and non-termination prediction. A moded SLD-tree is a symbolic approximation of concrete derivations, based on modes. It's finiteness is ensured by applying a *complete* loop check on the symbolic derivations, called *LP-check*. For its second phase, the analysis in [8] introduces a *sound* loop check on the moded SLD-tree. Each observed loop corresponds to a non-terminating computation for the concrete derivations. We illustrate this technique with an example.

Example 1. `a(f(X),s(Y)):- a(X,s(s(Y))).`

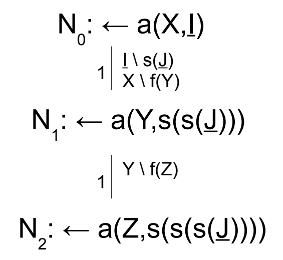

Figure 1: Moded SLD-tree of Example 1

P2P constructs the moded SLD-tree of Figure 1. In this tree, $\underline{\mathtt{I}}$ *and* $\underline{\mathtt{J}}$ *are input modes, denoting unknown ground terms. P2P proves that the path from* N_1 *to* N_2 *can be repeated infinitely often for any query* `a(X,s(`$\underline{\mathtt{J}}$`))` *with* `X` *a variable and* $\underline{\mathtt{J}}$ *a variable-free term.* □

In [8], the non-termination analyzer *P2P* was applied to all non-terminating programs of the benchmark of the annual termination competition[1]. It turned out that *P2P* was able to prove non-termination for all non-terminating programs in the benchmark.

[*]Supported by the Fund for Scientific Research - FWO-project G0561-08

[1]Available at http://www.lri.fr/~marche/termination-competition/

Our work since then has focussed on two new directions. One is to identify classes of programs for which current non-termination analyzers fail. A second is to investigate whether the inclusion of type-information, in addition to modes, may improve the power of our analyzer.

Considering the first of these questions, a limitation of both *NTI* and *P2P* is that they only detect non-terminating derivations if, within these derivations, some fixed sequence of clauses can be applied repeatedly. The following example violates this restriction.

Example 2. *The program, longer, loops for any query* `longer(L)`*, with* `L` *a non-empty list of zeros. The predicate zeros/1 checks if the list contains only zeros. At the recursive call, a zero is added to the list in the previous call.*

```
longer([0|L]):-
        zeros(L),                    zeros([]).
        longer([0,0|L]).             zeros([0|L]):- zeros(L).
```

The list in the recursive call is longer than the original one and thus, the number of applications of the recursive clause for zeros/1 increases in each recursion. Therefore, no fixed sequence of clauses can be repeated infinitely and both NTI and P2P fail to prove non-termination of this example. □

There are many variants of this class of programs. Instead of having an increasing number of applications of a same clause, we could have a predicate that always succeeds but in which alternative clauses are used for obtaining success. Again, *NTI* and *P2P* will fail to prove non-termination, because the sequence of clause applications is not repeated. We overcome this limitation by using non-failure information. *Non-failure analysis* ([2]) detects classes of goals that can be guaranteed not to fail, given mode and type information. If, in a derivation a non-failing selected atom is selected, we treat it with a special transition allowing to abstract from the exact sequence of clauses needed to reach success. To use the information provided by non-failure analysis in the non-termination analysis of [8], type information must be added to the symbolic derivation tree. We add this information using regular types ([4]).

Example 3. *For the program longer of Example 2, the type inference technique of [1] is able to automatically infer the following type definition and signatures:*

$$T_{lz} \rightarrow [\,]; [T_0 \mid T_{lz}] \qquad T_0 \rightarrow 0 \qquad longer(T_{lz}) \qquad zeros(T_{lz})$$

Non-failure analysis of [2] proves that zeros/1 is non-failing if its argument is an input mode of type T_{lz}. It cannot show that longer/1 with an argument of type T_{lz} is non-failing, because $longer([\,])$ fails. We will illustrate that we can prove non-termination on the basis of this extra information. □

The set of terms constructible from a certain type definition, is called the *denotation of the type*. We represent the denotation of type T by $Den(T)$.

Another limitation of both *NTI* and *P2P* is related to aliased variables. We illustrate this with an example from [6].

Example 4. `append([],L,L).    append([H|T],L,[H|R]):- append(T,L,R).`

The query `append(X,X,X)` *succeeds once with a computed answer substitution* `X/[]`*. The program loops after backtracking.*

Both NTI and P2P fail to prove non-termination due to the aliased variables in the query. □

Non-termination of the last example can be proven by specializing *append* for the considered query.

Example 5. *Specializing append for the query* `append(X,X,X)`*, using the ECCE specializer of [3], produces the following program:*

```
append(X,X,X):- app1(X).         app1([]).
app2([H|T]):- app2(T).           app1([A,B|C]):- app2(C).
```

Both NTI and P2P prove non-termination of this specialized program. □

2 Moded-Typed SLD-trees and the *NFG* transition by example

In this section, we extend the moded SLD-tree, intuitively introduced in Example 1, with types and a special transition to treat non-failing selected atoms. As in [8], we use the complete loop check *LP − check* to obtain finite trees.

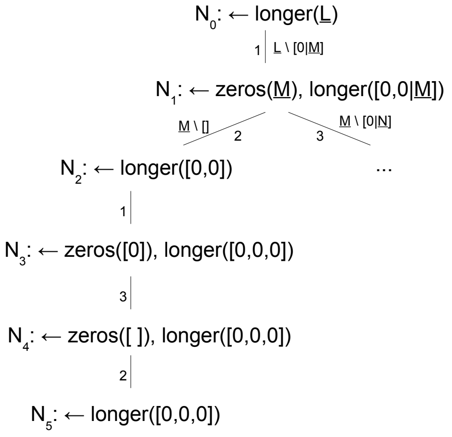
N0: ← longer(L)
1
L \ [0|M]
N1: ← zeros(M), longer([0,0|M])
M \ []
2
3
M \ [0|N]
N2: ← longer([0,0])
...
1
N3: ← zeros([0]), longer([0,0,0])
3
N4: ← zeros([ ]), longer([0,0,0])
2
N5: ← longer([0,0,0])

Figure 2: Moded-typed SLD-tree of the *longer* program

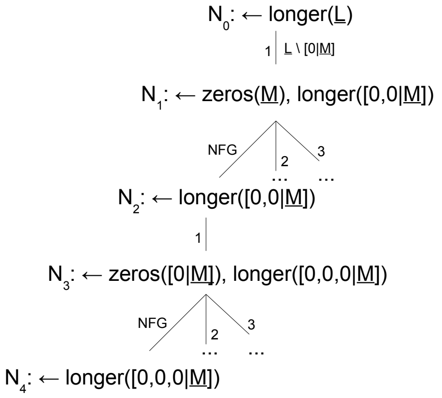

Figure 3: Moded-typed SLD-tree with *NFG* of the *longer* program

Example 6. *Figure 2 shows a part of the moded-typed SLD-tree for the longer program. The (inferred) types and signatures are also part of the tree, but are not shown here.*

Variables corresponding to input modes are denoted by underlining the variables name and are called input variables. When substituting an input variable with a term, all variables in that term also become input variables. At node N_5, the derivation is cut by $LP - check$. □

A moded-typed atom A represents a set of concrete atoms, called the *denotation* of A. This is the set of atoms obtained by replacing the input modes by arbitrary terms of their respective type. Because non-termination of an atom implies non-termination of all more general atoms, we consider a moded-typed atom A to also represent all more general atoms. We call this set the *extended denotation* of A, $Ext(A)$.

Example 7. $Ext(longer([0,0 \mid \underline{M}]))$ *contains* $longer([0,0]), longer([0,0,0|X]), longer(Y), \ldots$ □

To overcome the limitation illustrated in Example 2, we extend the moded-typed SLD-tree by adding a special transition NFG, to treat non-failing selected atoms. This transition allows to abstract from the sequence of clauses needed to solve the non-failing atom. Note that during the evaluation of a non-failing atom, normal variables may be bound to terms of their respective type. Therefore, we approximate the application of this unknown sequence of clauses by substituting all normal variables in the selected atom by fresh input variables of the correct type.

Example 8. *Figure 3 shows a derivation in the moded-typed SLD-tree with NFG of Example 2 for the query $longer(\underline{L})$. The selected atom at node N_1, $zeros(\underline{M})$, is non-failing and can be solved using an NFG transition. Since there are no normal variables in the selected atom, there are no substitutions for this transition. At node N_3, the NFG transition is applied as well.* □

Note that solving a non-failing atom corresponds to a potentially infinite sequence of clause applications. Thus, for an NFG transition $N_i : G_i \rightarrow_{NFG} N_j : G_j$, it is possible that no concrete state corresponding to G_j ever occurs in a concrete derivation. However, this will not cause any problems for our analysis. If our analysis detects and reports a non-terminating computation for a branch in the moded-typed SLD-tree which has an NFG transition, then a corresponding concrete atom either finitely succeeds or non-terminates. In both cases, reporting non-termination is correct.

3 Typed non-termination analysis with non-failing goals

In this section, we reformulate our non-termination conditions of [8] for moded-typed SLD-trees with *NFG* and show that program specialization allows to extend the applicability of the analysis.

To prove non-termination, we prove that a path between two nodes N_b and N_e in a moded-typed SLD-derivation can be repeated infinitely often. Such paths can be identified using the tree conditions in the following definition. If a path satisfies these conditions, we call it an *inclusion loop*.

Definition 1. *In a moded-typed SLD-derivation with NFG D, nodes $N_i : G_i$ and $N_j : G_j$, with A_i^1 and A_j^1 as selected atoms, are an* ***inclusion loop****, $N_i : G_i \overset{inc}{\rightarrow} N_j : G_j$, if:*

- *No substitutions on input variables occur in the path from N_i to N_j.*
- $A_i^1 \prec_{anc} A_j^1$.
- $Ext(A_j^1) \subset Ext(A_i^1)$ □

An inclusion loop $N_i : G_i \overset{inc}{\rightarrow} N_j : G_j$ corresponds to an infinite loop for every goal of the extended denotation of G_i.

In [8], we introduced a syntactic condition to check the third condition of the inclusion loop and we have shown that *input-generalizations* can be used to improve the applicability of the analysis. Due to space restrictions, these parts are omitted.

Example 9. *The moded-typed SLD-tree with NFG of Figure 3 allows to prove non-termination of the longer program. There are no substitutions on input variables between these nodes, the ancestor relation holds and $Ext(longer([0,0,0 \mid \underline{M}]))$ is a subset of $Ext(longer([0,0 \mid \underline{M}]))$.*

Every query $longer(\underline{L})$ can reach node N_2 if the substitution $\underline{L} \setminus [0 \mid \underline{M}]$ succeeds. Therefore, non-termination of all queries in $Ext(longer([0 \mid \underline{M}]))$ is proven. □

3.1 Program specialization

In Example 4, we pointed at a class of programs, related to aliased variables, for which current non-termination analyzers fail to prove non-termination. Non-termination analysis techniques have difficulties with treating queries with aliased variables. Program specialization often reduces the aliasing, due to argument filtering. So, in the context of aliasing, applying program specialization often improves the applicability of the analysis.

Program specialization can also be used in combination with the *NFG* transition. When solving a non-failing atom, all variables in the atom are substituted with new input variables. These input variables give an overestimation of the possible values after evaluating the non-failing atom. Program specialization can produce more instantiated, but equivalent clauses. These more instantiated clauses give a better approximation of the possible values after evaluating the non-failing atom. We illustrate this with an example.

Example 10. *The following program is an adaption of the longer program. In this program, $zeros/2$ generates the list containing an extra 0.*

```
longer([0|L]):-                     zeros([],[0]).
        zeros([0|L],Ln),            zeros([0|L],[0|R]):- zeros(L,R).
        longer(Ln).
```

The type definition of Example 3 is correct for this program. All arguments are of type T_{lz}. Non-failure analysis shows that zeros$/2$ is non-failing if its first argument is of type T_{lz} and the second argument is a free variable.

As in the longer program, an NFG transition is needed to prove non-termination since the number of applications of the recursive clause for zeros$/2$ increases. When treating a non-failing atom zeros$([0 \mid \underline{L}], Ln)$ with the NFG transition, the free variable Ln is replaced by a new input variable $\underline{Ln}$ of type T_{lz}, losing the information that $\underline{Ln}$ is non-empty.

Program specialization allows to generate a more instantiated version of the first clause, showing that the second argument of zeros$/2$ is bound to a non-empty list.

```
longer([0|L]):- zeros([0|L],[0|Ln]), longer([0|Ln]).
```

Proving non-termination of this specialized program, using the NFG transition to treat zeros$/2$ is straightforward. □

4 Conclusion and Future work

In this paper, we identified classes of logic programs for which current analyzers fail to prove non-termination and we extended our non-termination analysis of [8] to overcome these limitations. As in [8], non-termination is proven by constructing a symbolic derivation tree, representing all derivations for a class of queries, and then proving that a path in this tree can be repeated infinitely.

The most important class of programs for which current analyzers fail, are programs for which no fixed sequence of clauses can be repeated infinitely. We have shown that non-failure information ([2]) allows to abstract away from the exact sequence of clauses needed to solve non-failing goals.

We illustrated that program specialization ([3]) can be used to overcome another limitation of current analyzers. If non-termination can not be proven due to aliased variables, redundant argument filtering may remove these duplicated variables from the program. Program specialization can also be used in combination with the special transition for non-failing atoms, giving a better approximation of the possible values after solving the non-failing atom.

References

[1] M. Bruynooghe, J.P. Gallagher, and W. Van Humbeeck. Inference of well-typing for logic programs with application to termination analysis. In *SAS 2005*, volume 3672 of *LNCS*, pages 35–51. Springer, 2005.

[2] Saumya K. Debray, Pedro López-García, and Manuel V. Hermenegildo. Non-failure analysis for logic programs. In *ICLP*, pages 48–62, 1997.

[3] M. Leuschel. Logic program specialisation. In *Partial Evaluation*, volume 1706 of *LNCS*, pages 155–188. Springer, 1998.

[4] Manh Thang Nguyen. *Termination Analysis: Crossing Paradigm Borders*. PhD thesis, K.U.Leuven, June 2009. De Schreye, Daniel (supervisor).

[5] Étienne Payet and Frédéric Mesnard. Nontermination inference of logic programs. *ACM Transactions on Programming Languages and Systems*, 28(2):256–289, 2006.

[6] Danny De Schreye and Stefaan Decorte. Termination of logic programs: The never-ending story. *J. Log. Program.*, 19/20:199–260, 1994.

[7] Yi-Dong Shen, Danny De Schreye, and Dean Voets. Termination prediction for general logic programs. *CoRR*, abs/0905.2004, 2009.

[8] D. Voets and D. De Schreye. A new approach to non-termination analysis of logic programs. In P. M. Hill and D. S. Warren, editors, *ICLP*, volume 5649 of *LNCS*, pages 220–234. Springer, 2009.

Certification extends Termination Techniques

Christian Sternagel* (christian.sternagel@uibk.ac.at), University of Innsbruck, Austria
René Thiemann (rene.thiemann@uibk.ac.at), University of Innsbruck, Austria

1 Introduction

Termination provers for term rewrite systems (TRSs) became more and more powerful in the last years. One reason is that a proof of termination no longer is just some reduction order which contains the rewrite relation of the TRS. Currently, most provers construct a proof in the dependency pair framework (DP framework). This allows to combine basic termination techniques in a flexible way. Hence, a termination proof is a tree where at each node a specific technique is applied. Therefore, instead of just stating the precedence of some lexicographic path order or giving some polynomial interpretation, current termination provers return proof trees consisting of many different techniques and reaching sizes of several megabytes. Thus, it would be too much work to check by hand whether these trees really form a valid proof. (Also, checking by hand does not provide a very high degree of confidence.)

It is regularly demonstrated that we cannot blindly trust in the output of termination provers. Every now and then, some termination prover delivers a faulty proof. Most often, this is only detected if there is another prover giving a contradicting answer on the same problem. To improve this situation, three systems have been developed over the last few years: CiME/Coccinelle [5,6], Rainbow/CoLoR [4], and CeTA/IsaFoR [16]. These systems either certify or reject a given termination proof. Here, Coccinelle and CoLoR are libraries on rewriting for Coq (http://coq.inria.fr) and IsaFoR is our library on rewriting for Isabelle [15]. (Throughout this paper we just write Isabelle whenever we refer to Isabelle/HOL.) And indeed, using certifiers several bugs have been detected. For example, in the termination competition of the last year (November 2009), at least eight faulty proofs were spotted by certifiers.[1] (Caused by three different bugs, all of which were most likely due to some output error.)

Although many termination techniques have already been formalized—CeTA can certify termination or nontermination proofs for 1522 out of the 2132 TRSs from the TPDB version 7.0.2 which is over 70 % of the whole database—there are still several techniques that have not been formalized. So, clearly there are termination proofs that are produced by some termination tool where the certifiers have to become more powerful.

However, a similar situation also occurs in the other direction. We have formalized termination techniques in a more general setting as they have been introduced. Hence, currently we can certify proofs using techniques that no termination tool supports so far. In this paper we shortly present two of these formalizations.

(a) Polynomial orders with negative constants [12].

(b) Arctic termination [13].

Here, for (a) we were able to lift the result from the naturals as introduced in [12] to an arbitrary carrier, including matrices (Sec. 3). For (b) we have generalized the arctic semiring and the arctic semiring below zero into one semiring which subsumes both existing approaches and extends them to the rationals (Sec. 4).

Note that all the proofs that are presented (or omitted) in the following, have been formalized in our Isabelle library IsaFoR. This library and the executable certifier CeTA are available at CeTA's website:

http://cl-informatik.uibk.ac.at/software/ceta

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

[1]Result IDs http://termcomp.uibk.ac.at/termcomp/competition/resultDetail.seam?resultId=135160, 136252, 136278, 136365, 136378, 136499, 137163, and 137465

2 Preliminaries

We assume familiarity with term rewriting [2]. Still, we recall the most important notions that are used later on. A *term* t over a set of *variables* $\mathcal{V}$ and a set of *function symbols* $\mathcal{F}$ is either a variable $x \in \mathcal{V}$ or an n-ary function symbol $f \in \mathcal{F}$ applied to n argument terms $f(t_1, \dots, t_n)$.

A *rewrite rule* is a pair of terms $\ell \to r$ and a TRS $\mathcal{R}$ is a set of rewrite rules. The *rewrite relation (induced by* $\mathcal{R}$*)* $\to_{\mathcal{R}}$ is the closure under substitutions and under contexts of $\mathcal{R}$, i.e., $s \to_{\mathcal{R}} t$ iff there is a context C, a rewrite rule $\ell \to r \in \mathcal{R}$, and a substitution σ such that $s = C[\ell\sigma]$ and $t = C[r\sigma]$. A TRS $\mathcal{R}$ is terminating, written $\mathsf{SN}(\mathcal{R})$, if there is no infinite derivation $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} t_3 \to_{\mathcal{R}} \dots$.

3 Polynomial Orders with Negative Constants

Polynomial orders [14] are a well-known technique to prove termination. They are an instance of the termination technique of well-founded monotone algebras. Such algebras can be used for all termination techniques that rely on *reduction pairs* [1]. Here, a reduction pair consists of two partial orders $(\succsim, \succ)$ where $\succsim$ and $\succ$ are stable, $\succsim$ is reflexive and monotone, $\succ$ is well-founded, and $\succsim$ is compatible to $\succ$, i.e., $\succsim \circ \succ \subseteq \succ$. If additionally $\succ$ is monotone, then we call $(\succsim, \succ)$ a *monotone* reduction pair.

It is well-known that reduction pairs can be used for proving termination of TRSs within the DP framework [1, 10, 11]. Moreover, monotone reduction pairs can be used for direct termination proofs or rule removal [3, 9, 14].

To formalize polynomial orders, we first assume some semiring over which the polynomials are built.

Definition 1. *A structure $(\mathcal{U}, \oplus, \odot, \underline{0}, \underline{1})$ with universe $\mathcal{U}$, two binary operation $\oplus$ and $\odot$ on $\mathcal{U}$, and with $\underline{0}, \underline{1} \in \mathcal{U}$ is a* semiring with one-element *iff*

- *$\oplus$ and $\odot$ are associative and $\oplus$ is commutative*
- *$\underline{0} \neq \underline{1}$, $\underline{0}$ and $\underline{1}$ are neutral elements w.r.t. $\oplus$ and $\odot$, respectively, and $\underline{0} \odot x = x \odot \underline{0} = \underline{0}$*
- *$\odot$ distributes over $\oplus$: $x \odot (y \oplus z) = x \odot y \oplus x \odot z$ and $(x \oplus y) \odot z = x \odot z \oplus y \odot z$*

To obtain polynomial orders, we assume a strict and a non-strict order. Moreover, we demand the existence of a unary predicate mono where $\mathsf{mono}(x)$ indicates that multiplication with x is monotone w.r.t. the strict order.

Definition 2. *A structure $(\mathcal{U}, \oplus, \odot, \underline{0}, \underline{1}, \geq, >, \mathsf{mono})$ is an* ordered semiring *iff $(\mathcal{U}, \oplus, \odot, \underline{0}, \underline{1})$ is a semiring with one-element and additionally:*

- *$\geq$ is reflexive and transitive; $>$ and $\geq$ are compatible: $> \circ \geq \subseteq >$ and $\geq \circ > \subseteq >$*
- *$\underline{1} \geq \underline{0}$ and $\mathsf{mono}(\underline{1})$*
- *$\oplus$ is left-monotone w.r.t. $\geq$: if $x \geq y$ then $x \oplus z \geq y \oplus z$*
- *$\oplus$ is left-monotone w.r.t. $>$: if $x > y$ then $x \oplus z > y \oplus z$*
- *$\odot$ is left-monotone w.r.t. $\geq$: if $x \geq y$ and $z \geq \underline{0}$ then $x \odot z \geq y \odot z$; $\odot$ is right-monotone w.r.t. $\geq$*
- *$\odot$ is right-monotone w.r.t. $>$: if $\mathsf{mono}(x)$, $x \geq \underline{0}$, and $y > z$ then $x \odot y > x \odot z$*
- *$\{(x, y) \mid x > y \wedge y \geq \underline{0}\}$ is well-founded*

Note that using the approach of well-founded monotone algebras, every interpretation of the function symbols over some ordered semiring gives rise to a strict $(\succ)$ and a non-strict $(\succsim)$ order on terms. For example, for a polynomial interpretation $\mathcal{P}ol$ we define $s \succ_{\mathcal{P}ol} t$ iff $[s] > [t]$, and $s \succsim_{\mathcal{P}ol} t$ iff $[s] \geq [t]$

where $[s]$ is the homeomorphic extension of $\mathcal{P}ol$ to terms.

Theorem 3. *Let $\mathcal{P}ol$ be a polynomial interpretation over an ordered semiring $(\mathcal{U}, \oplus, \odot, \underline{0}, \underline{1}, \geq, >, \mathsf{mono})$ where $[f](x_1, \ldots, x_n) = f_0 \oplus f_1 \odot x_1 \oplus \cdots \oplus f_n \odot x_n$ and $f_i \geq \underline{0}$ for all $0 \leq i \leq n$ and every n-ary symbol f. Then $(\succsim_{\mathcal{P}ol}, \succ_{\mathcal{P}ol})$ is a reduction pair. If moreover, $\mathsf{mono}(f_i)$ for all $1 \leq i \leq n$ then $(\succsim_{\mathcal{P}ol}, \succ_{\mathcal{P}ol})$ is a monotone reduction pair.*

Example 4 (Ordered Semirings). *$(\mathbb{N}, +, \cdot, 0, 1, \geq, >, \geq 1)$, $(\mathbb{Z}, +, \cdot, 0, 1, \geq, >, \geq 1)$, and $(\mathbb{Q}, +, \cdot, 0, 1, \geq, >_\delta, \geq 1)$ are ordered semirings. In the last case, we assume a fixed rational number δ with $0 < \delta$, and where $>_\delta$ is defined by $x >_\delta y$ iff $x - y \geq \delta$.*

To formalize matrix-interpretations [8], we followed the approach of [7] and used a domain with an additional strict-dimension and where the elements are matrices—instead of vectors as in [8]. In detail, we have proven that if $0 < sd \leq n$ and $(\mathcal{U}, \oplus, \odot, \underline{0}, \underline{1}, \geq, >, \mathsf{mono})$ is an ordered semiring, then $(\mathcal{U}^{n\times n}, \oplus^{n\times n}, \odot^{n\times n}, \underline{0}^{n\times n}, \underline{1}^{n\times n}, \geq^{n\times n}, >^{n\times n}_{sd}, \mathsf{mono}^{n\times n}_{sd})$ is also an ordered semiring where all operations and constants are lifted to work on n-dimensional matrices, with the strict-dimension sd. Here, $\geq^{n\times n}$ compares the arguments component-wise, and $M >^{n\times n}_{sd} M'$ iff $M \geq^{n\times n} M'$ and at least one entry in the upper-left $sd \times sd$-submatrix is strictly decreasing w.r.t. $>$. Moreover, $\mathsf{mono}^{n\times n}_{sd}$ demands that for every column in the upper-left $sd \times sd$-submatrix there is at least one monotone entry.

As observed in [7], choosing $sd = 1$, is comparable to the classic definition of matrix-interpretations. Choosing $sd = n$, is always best if one does not require monotonic reduction pairs. However, to ensure monotonicity also a small value of sd might be attractive.

To lift the requirement in Thm. 3 that all f_i have to be at least $\underline{0}$, in [12], polynomial orders with negative constants have been introduced. There, the constant part can be arbitrary but the interpretation of a function is always wrapped into a $\mathsf{max}(\underline{0}, \cdot)$ operation to ensure well-foundedness. This complicates the comparison of terms, as the resulting interpretations are not pure polynomials anymore, but also contain the max-operator. To this end, approximations $[\cdot]_{left}$ and $[\cdot]_{right}$ have been introduced which interpret terms by polynomials without max, such that $[s]_{left} \leq [s] \leq [s]_{right}$.

However, the existing approximations are unsound if generalized naively. For example, in the case where the constant part is negative, it is removed. This works fine for the integers and the rationals, but not for matrices, as here some parts of the matrix may be negative, but other parts can also be positive and thus, cannot be removed. Thus, we formalized the following approximations which are equivalent to those of [12], but also work for matrices:

Definition 5. *Let $\mathsf{cp}(\cdot)$ be the constant part and $\mathsf{ncp}(\cdot)$ be the non-constant part of a polynomial.*

$$[x]_{left} = [x]_{right} = x$$

$$[f(t_1, \ldots, t_n)]_{left} = \begin{cases} \mathsf{max}(\underline{0}, \mathsf{cp}(p_{left})) & \text{if } \mathsf{ncp}(p_{left}) = \underline{0} \\ p_{left} & \text{otherwise} \end{cases}$$

$$[f(t_1, \ldots, t_n)]_{right} = \mathsf{ncp}(p_{right}) \oplus \mathsf{max}(\underline{0}, \mathsf{cp}(p_{right}))$$

$$\text{where } p_{left} = [f]([t_1]_{left}, \ldots, [t_n]_{left}) \quad \text{and} \quad p_{right} = [f]([t_1]_{right}, \ldots, [t_n]_{right})$$

Note that for Def. 5 we have to extend ordered semirings by the additional unary operation: $\mathsf{max}(\underline{0}, \cdot)$.

Definition 6. *A structure $(\mathcal{U}, \oplus, \odot, \underline{0}, \underline{1}, \geq, >, \mathsf{mono}, \mathsf{max}\underline{0})$ is an* ordered semiring with max *iff $(\mathcal{U}, \oplus, \odot, \underline{0}, \underline{1}, \geq, >, \mathsf{mono})$ is an ordered semiring and additionally:*

- $\mathsf{max}\underline{0}(x) \geq \underline{0}$ *and* $\mathsf{max}\underline{0}(x) \geq x$
- $y \geq x \geq \underline{0}$ *implies* $\mathsf{max}\underline{0}(y) \geq \mathsf{max}\underline{0}(x) = x$

Theorem 7. *Let* $(\mathcal{U},\oplus,\odot,\underline{0},\underline{1},\geq,>,\mathsf{mono},\mathsf{max}\underline{0})$ *be an ordered semiring with max and* $\mathcal{P}ol$ *be a polynomial interpretation where* $[f](x_1,\dots,x_n) = f_0 \oplus f_1 \odot x_1 \oplus \cdots \oplus f_n \odot x_n$ *and* $f_i \geq \underline{0}$ *for all* $1 \leq i \leq n$ *and every n-ary symbol f. Then* $(\succsim_{\mathcal{P}ol},\succ_{\mathcal{P}ol})$ *is a reduction pair where* $s \succ / \succsim t$ *can be approximated by* $[s]_{left} > / \geq [t]_{right}$.

Example 8. *All ordered semirings of Ex. 4 are also ordered semirings with max, where* $\mathsf{max}\underline{0}$ *is the standard operation on* $\mathbb{N}$, $\mathbb{Z}$, *and* $\mathbb{Q}$, *and* $\mathsf{max}\underline{0}$ *is performed component-wise for matrices.*

For example, for $\mathbb{Q}$ *it is now possible to use interpretations like*

$$[\mathsf{half}](x) = \frac{1}{2}\cdot x + \frac{1}{2} \qquad [\mathsf{p}](x) = x - 1 \qquad [\mathsf{s}](x) = x+1$$

where

$$[\mathsf{s}(x)]_{left} = x + 1 > \frac{1}{2}\cdot x + \frac{1}{2} = [\mathsf{p}(\mathsf{half}(\mathsf{s}(\mathsf{s}(x))))]_{right}$$

Since we are not aware of any termination tool that supports these interpretation, we would like to encourage their integration, perhaps an interpretation like

$$[\mathsf{f}](x,y) = \begin{pmatrix} \frac{1}{2} & 8 \\ \frac{7}{5} & 0 \end{pmatrix} \cdot x + y + \begin{pmatrix} -\frac{1}{3} & 3 \\ -5 & \frac{2}{9} \end{pmatrix}$$

increases the power in the next competition.

4 Arctic Semirings

In [13], the *arctic semiring* as well as the *arctic semiring below zero*, where used the first time in the well-founded monotone algebra setting.

Example 9 (Arctic Semirings)**.** *The arctic semiring* $(\mathbb{A}_{\mathbb{N}},\mathsf{max},+,-\infty,0)$, *the arctic semiring below zero* $(\mathbb{A}_{\mathbb{Z}},\mathsf{max},+,-\infty,0)$, *and the arctic rational semiring* $(\mathbb{A}_{\mathbb{Q}},\mathsf{max},+,-\infty,0)$, *are semirings with one-element as in Def. 1. The carriers are given by* $\mathbb{A}_S = S \cup \{-\infty\}$. *Furthermore, the standard operations* max *and* $+$ *are extended such that* $\mathsf{max}\{x,-\infty\} = x$ *and* $x + -\infty = -\infty + y = -\infty$, *for all x and y.*

Definition 10. *A structure* $(\mathcal{U},\oplus,\odot,\underline{0},\underline{1},\geq,>,\mathsf{pos})$ *is an ordered arctic semiring iff* $(\mathcal{U},\oplus,\odot,\underline{0},\underline{1})$ *is a semiring with one-element and additionally:*

- $\geq$ *is reflexive and transitive;* $>$ *and* $\geq$ *are compatible:* $> \circ \geq \, \subseteq \, >$ *and* $\geq \circ > \, \subseteq \, >$
- $\underline{1} \geq \underline{0}$; $\mathsf{pos}(\underline{1})$; $x > \underline{0}$; $x \geq \underline{0}$; *and if* $\underline{0} > x$ *then* $x = \underline{0}$
- $\oplus$ *is left-monotone w.r.t.* $\geq$
- $\oplus$ *is monotone w.r.t.* $>$: *if* $x > y$ *and* $x' > y'$ *then* $x \oplus x' > y \oplus y'$
- $\odot$ *is left- and right-monotone w.r.t.* $\geq$ *and left-monotone w.r.t.* $>$
- *staying positive: if* $\mathsf{pos}(x)$ *and* $\mathsf{pos}(y)$ *then* $\mathsf{pos}(x \oplus z)$ *and* $\mathsf{pos}(x \odot y)$
- $\{(x,y) \mid x > y \wedge \mathsf{pos}(y)\}$ *is well-founded*

Theorem 11. *Let* $\mathcal{P}ol$ *be a polynomial interpretation over an ordered arctic semiring* $(\mathcal{U},\oplus,\odot,\underline{0},\underline{1},\geq,>,\mathsf{pos})$ *where* $[f](x_1,\dots,x_n) = f_0 \oplus f_1 \odot x_1 \oplus \cdots \oplus f_n \odot x_n$ *and* $\mathsf{pos}(f_i)$ *for some* $0 \geq i \geq n$ *and every n-ary symbol f. Then* $(\succsim_{\mathcal{P}ol},\succ_{\mathcal{P}ol})$ *is a reduction pair where* $s \succsim / \succ t$ *is approximated by comparing* $[s]$ *and* $[t]$ *component-wise using* $> / \geq$. *(For example to compare* $a \odot x \oplus b \odot y \oplus c > d \odot x \oplus e \odot y \oplus f$ *one demands* $a > d$, $b > e$, *and* $c > f$.*)*

Moreover, if $n > 0$ and $(\mathcal{U}, \oplus, \odot, \underline{0}, \underline{1}, \geq, >, \mathsf{pos})$ is an ordered arctic semiring, then $(\mathcal{U}^{n\times n}, \oplus^{n\times n}, \odot^{n\times n}, \underline{0}^{n\times n}, \underline{1}^{n\times n}, \geq^{n\times n}, >^{n\times n}, \mathsf{pos}^{n\times n})$ is also an ordered arctic semiring where all operations and constants are lifted to work on n-dimensional matrices. Here $\geq^{n\times n}$ and $>^{n\times n}$, compare arguments componentwise and $\mathsf{pos}^{n\times n}$ checks, whether the leftmost topmost element is pos.

Example 12 (Ordered Arctic Semirings). *All arctic semirings of Ex. 9 are also ordered arctic semirings. In all three cases, we use the non-strict ordering* $x \geq y \equiv y = -\infty \vee (x \neq -\infty \wedge x \geq_{\mathbb{N}/\mathbb{Z}/\mathbb{Q}} y)$. *For* $\mathbb{A}_\mathbb{N}$ *and* $\mathbb{A}_\mathbb{Z}$, *we use the strict ordering* $x > y \equiv y = -\infty \vee (x \neq -\infty \wedge x >_{\mathbb{N}/\mathbb{Z}} y)$, *and for* $\mathbb{A}_\mathbb{Q}$, *we use the strict ordering* $x >_\delta y \equiv y = -\infty \vee (x \neq -\infty \wedge x - y \geq_\mathbb{Q} \delta)$ *for some* $\delta > 0$. *Furthermore, the check for positiveness is defined by* $\mathsf{pos}(x) \equiv x \neq -\infty \wedge x \geq_{\mathbb{N}/\mathbb{Z}/\mathbb{Q}} 0$.

Note that the ordered arctic semiring over $\mathbb{A}_\mathbb{Q}$, together with Thm. 11, unifies and extends Theorems 12 and 14 of [13]. Here, the main advantage of our approach is that we only restrict interpretations $[f](x_1,\ldots,x_n) = f_0 \oplus f_1 \odot x_1 \oplus \cdots \oplus f_n \odot x_n$ by demanding that at least one f_i is positive. This is in contrast to the theorem about the arctic semiring below zero in [13] where always the constant part f_0 has to be positive. However, Waldmann observed that for finite TRSs one can transform every polynomial order over the arctic rationals into an order over the arctic naturals by multiplication and shifting.

References

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

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

[3] A. Ben Cherifa and P. Lescanne. Termination of Rewriting Systems by Polynomial Interpretations and Its Implementation. *Science of Computer Programming*, 9(2):137–159, 1987.

[4] F. Blanqui, W. Delobel, S. Coupet-Grimal, S. Hinderer, and A. Koprowski. CoLoR, a Coq library on rewriting and termination. In *Proc. WST'06*, pages 69–73, 2006.

[5] É. Contejean, P. Courtieu, J. Forest, A. Paskevich, O. Pons, and X. Urbain. A3PAT, an approach for certified automated termination proofs. In *Proc. PEPM'10*. To appear.

[6] É. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Certification of automated termination proofs. In *Proc. FroCoS'07*, LNAI 4720, pages 148–162, 2007.

[7] P. Courtieu, G. Gbedo, and O. Pons. Improved matrix interpretation. In *Proc. SOFSEM'10*, LNCS 5901, pages 283–295, 2010.

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

[9] A. Geser. *Relative Termination*. PhD thesis, University of Passau, Germany, 1990.

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

[11] N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1-2):172–199, 2005.

[12] N. Hirokawa and A. Middeldorp. Tyrolean Termination Tool: Techniques and features. *Information and Computation*, 205(4):474–511, 2007.

[13] A. Koprowski and J. Waldmann. Arctic termination ... below zero. In *Proc. RTA'08*, LNCS 5117, pages 202–216, 2008.

[14] D. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA, 1979.

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

[16] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Proc. TPHOLs '09*, LNCS 5674, pages 452–468, 2009.

Termination Analysis of Logic Programs with Cut Using Dependency Triples*

Thomas Ströder, Jürgen Giesl
LuFG Informatik 2, RWTH Aachen University, Germany
{stroeder,giesl}@informatik.rwth-aachen.de
Peter Schneider-Kamp
IMADA, University of Southern Denmark, Denmark
petersk@imada.sdu.dk

Abstract

In very recent work, we introduced a non-termination preserving transformation from logic programs with cut to definite logic programs. In this paper we extend the transformation such that logic programs with cut are transformed into dependency triple problems instead of definite logic programs. By the implementation of our new method and extensive experiments, we show that the power of automated termination analysis for logic programs with cut is increased substantially.

1 Introduction

Automated termination analysis for logic programs has been widely studied, see, e.g., [3, 4, 5, 11, 13, 14, 17]. Still, virtually all existing techniques only prove universal termination of *definite* logic programs, which do not use the **cut** "!", while most realistic Prolog programs do so. In [16] we introduced a non-termination preserving automated transformation from logic programs with cut to definite logic programs. The transformation consists of two stages. In the first stage we construct a so-called *termination graph* for a given logic program with cut. The second stage is the generation of a definite logic program from this termination graph. In this paper, we improve the second stage of the transformation by generating dependency triple problems instead of definite logic programs from termination graphs.

Dependency triples were introduced in [12] and improved further to the so-called *dependency triple framework* in [15]. Here, the idea was to adapt the successful dependency pair framework [2, 8, 9, 10] from term rewriting to (definite) logic programming. The experiments in [15] showed that this leads to the most powerful approach for automated termination analysis of definite logic programs so far. Our aim is to benefit from this work by providing an immediate translation from termination graphs to dependency triple problems in order to obtain an analysis that preserves termination in more cases.

Example 1. *To illustrate the concepts and the contributions of this paper, we use the leading example of Fig. 1. It formulates a simplified variant of a functional program from [7, 20] with nested recursion as a logic program. The auxiliary predicate* p *is used to compute the predecessor of a natural number while* eq *is used to unify two terms. See, e.g., [1] for the basics of logic programming.*

$$\begin{array}{rcll}
\mathsf{f}(0,Y) & \leftarrow & !,\mathsf{eq}(Y,0). & (1)\\
\mathsf{f}(X,Y) & \leftarrow & \mathsf{p}(X,P),\mathsf{f}(P,U),\mathsf{f}(U,Y). & (2)\\
\mathsf{p}(0,0). & & & (3)\\
\mathsf{p}(\mathsf{s}(X),X). & & & (4)\\
\mathsf{eq}(X,X). & & & (5)
\end{array}$$

Figure 1: Example program

Note that when ignoring cuts, this logic program is not terminating for the set of queries $\mathcal{Q} = \{\mathsf{f}(t_1,t_2) \mid t_1 \textit{ is ground}\}$. *On the other hand, the program terminates if the cuts are taken into account.*

*Supported by the DFG grant GI 274/5-2, the DFG Research Training Group 1298 (*AlgoSyn*), and the Danish Natural Science Research Council.

In this paper, we first present a termination graph obtained for this example program in Sect. 2 before we apply our new transformation from termination graphs to dependency triple problems in Sect. 3. We show that this new transformation has significant practical advantages in Sect. 4 and, finally, we conclude in Sect. 5.

2 Termination Graphs

Using the method from [16] we obtain the following termination graph for Ex. 1, where we applied some simplifications to ease presentation. The states of this graph contain sequences of *abstract queries*. Here, the *abstract variables* T_i stand for arbitrary terms, whereas underlined abstract variables only stand for ground terms. A sequence of queries $Q_1 \mid Q_2 \mid Q_3 \mid \ldots$ represents the current goal Q_1 and the remaining backtracking possibilities $Q_2, Q_3, \ldots$ in the order of their execution. Sometimes we annotate states by *unification information* such as "$T_4 \not\sim 0$" meaning that T_4 only stands for terms that do not unify with 0. We start in Node A with the state $\mathsf{f}(\underline{T_1}, T_2)$ representing the set of queries $\mathcal{Q}$. Then the termination graph is constructed by a symbolic evaluation of the program. The CASE rule performs Prolog's clause selection rule by labeling the queries with the numbers of the program clauses to indicate which clause to apply next to a query. We applied this rule to the initial node A leading to a node B with two labeled copies of this query. They correspond to the two possibly applicable clauses (1) and (2) in the program. The EVAL rule then performs the resolution with Clause (1), leading to Node C. Moreover, it also produces the child node D which represents those cases where T_1 stands for a term that does not unify with 0. Here, we have to backtrack by removing the first goal from the current state. If we detect that the current goal cannot unify with the head of the corresponding program clause, we use the BACKTRACK rule which is equivalent to the second successor of the EVAL rule. The CUTALL rule drops further backtracking possibilities while the SUC rule backtracks after a successful evaluation, since we examine universal termination. Finally, the SPLIT rule separates two atoms in one query and the INSTANCE rule refers back to a state representing a superset of terms compared to the current state. The SPLIT and INSTANCE rules are needed to obtain a finite graph instead of an infinite tree. We refer to [16] for further details and explanations. In our example, the termination graph of Fig. 2 represents all possible derivations of the program for the set of queries $\mathcal{Q}$ from Ex. 1.

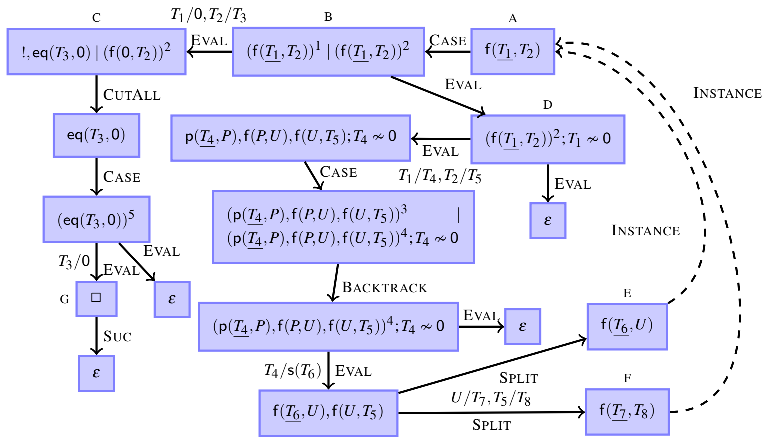
C
$T_1/0, T_2/T_3$
B
A
$!, \mathsf{eq}(T_3,0) \mid (\mathsf{f}(0,T_2))^2$
EVAL
$(\mathsf{f}(\underline{T_1},T_2))^1 \mid (\mathsf{f}(\underline{T_1},T_2))^2$
CASE
$\mathsf{f}(\underline{T_1},T_2)$
INSTANCE
EVAL
D
CUTALL
$\mathsf{eq}(T_3,0)$
$\mathsf{p}(\underline{T_4},P),\mathsf{f}(P,U),\mathsf{f}(U,T_5); T_4 \not\sim 0$
EVAL
$(\mathsf{f}(\underline{T_1},T_2))^2; T_1 \not\sim 0$
CASE
CASE
$T_1/T_4, T_2/T_5$
EVAL
$(\mathsf{eq}(T_3,0))^5$
$(\mathsf{p}(\underline{T_4},P),\mathsf{f}(P,U),\mathsf{f}(U,T_5))^3 \mid (\mathsf{p}(\underline{T_4},P),\mathsf{f}(P,U),\mathsf{f}(U,T_5))^4; T_4 \not\sim 0$
ε
INSTANCE
EVAL
$T_3/0$
EVAL
G
$\square$
ε
BACKTRACK
E
$(\mathsf{p}(\underline{T_4},P),\mathsf{f}(P,U),\mathsf{f}(U,T_5))^4; T_4 \not\sim 0$
EVAL
ε
$\mathsf{f}(\underline{T_6},U)$
SUC
ε
$T_4/\mathsf{s}(T_6)$
EVAL
SPLIT
F
$U/T_7, T_5/T_8$
$\mathsf{f}(\underline{T_6},U),\mathsf{f}(U,T_5)$
$\mathsf{f}(\underline{T_7},T_8)$
SPLIT

Figure 2: Termination Graph for Ex. 1.

3 Transformation into Dependency Triple Problems

To prove that all derivations of the example program and the set of queries $\mathcal{Q}$ are terminating, we have to show that the cycles in the termination graph from Fig. 2 cannot be traversed infinitely often when following a derivation of the original program. To this end, we synthesize a dependency triple problem [15] simulating this traversal.

The basic structure in the dependency triple framework is very similar to a clause in logic programming. A *dependency triple* (DT) [12] is a clause $H \leftarrow I, B$ where H and B are atoms and I is a sequence of atoms. Intuitively, such a DT states that a call that is an instance of H can be followed by a call that is an instance of B if the corresponding instance of I can be proven.

Here, a "derivation" is defined in terms of a chain. Let $\mathscr{D}$ be a set of DTs, $\mathscr{P}$ be the program under consideration, and $\mathscr{Q}$ be the class of queries to be analyzed.[1] A (possibly infinite) sequence $(H_0 \leftarrow I_0, B_0), (H_1 \leftarrow I_1, B_1), \ldots$ of variants from $\mathscr{D}$ is a $(\mathscr{D}, \mathscr{Q}, \mathscr{P})$*-chain* iff there are substitutions θ_i, σ_i and an $A \in \mathscr{Q}$ such that $\theta_0 = mgu(A, H_0)$ and for all i, σ_i is an answer substitution for the query $I_i\theta_i$ in the program $\mathscr{P}$, and $\theta_{i+1} = mgu(B_i\theta_i\sigma_i, H_{i+1})$. Such a tuple $(\mathscr{D}, \mathscr{Q}, \mathscr{P})$ is called a *dependency triple problem* and it is *terminating* iff there is no infinite $(\mathscr{D}, \mathscr{Q}, \mathscr{P})$-chain.

As an example, consider the DT problem $(\mathscr{D}, \mathscr{Q}, \mathscr{P})$ with $\mathscr{D} = \{d_1\}$ where $d_1 = \mathsf{p}(\mathsf{s}(X), Y) \leftarrow \mathsf{eq}(X, Z), \mathsf{p}(Z, Y)$, $\mathscr{Q} = \{\mathsf{p}(t_1, t_2) \mid t_1 \text{ is ground}\}$, and $\mathscr{P} = \{\mathsf{eq}(X, X)\}$. Now, "$d_1$, d_1" is a $(\mathscr{D}, \mathscr{Q}, \mathscr{P})$-chain. To see this, assume that $A = \mathsf{p}(\mathsf{s}(\mathsf{s}(0)), 0)$. Then $\theta_0 = \{X/\mathsf{s}(0), Y/0\}$, $\sigma_0 = \{Z/\mathsf{s}(0)\}$, and $\theta_1 = \{X/0, Y/0\}$.

The basic idea how to synthesize DT problems from termination graphs is to generate DTs for every *triple path* in the termination graph. These are paths which traverse cycles or which connect cycles to the initial state. In Fig. 2 there are two cycles already containing the initial state. As cycles must contain at least one INSTANCE edge, it is sufficient to consider triple paths from successor states of INSTANCE nodes or the initial state to INSTANCE nodes or their successors. So in our example, we have two triple paths: from A to E and from A to F. We use distinct predicate symbols for every state having all distinct variables occurring in the respective state as arguments. The only exception are INSTANCE nodes. Here we use the same predicate symbol as for the successor of the INSTANCE node. So if q is the new predicate symbol for Node A, then A is converted to the atom $\mathsf{q}(T_1, T_2)$, E is converted to $\mathsf{q}(T_6, U)$, and F is converted to $\mathsf{q}(T_7, T_8)$. To transform triple paths into DTs, we use the first node (e.g., A) as the head of the DT and the last node (e.g., E or F) as the last atom of the DT. Moreover, we apply the substitutions on the path to the head of the DT. For the path from A to E we obtain the substitution $[T_1/T_4, T_2/T_5] \circ [T_4/\mathsf{s}(T_6)]$ and, thus, the DT $\mathsf{q}(\mathsf{s}(T_6), T_5) \leftarrow \mathsf{q}(T_6, U)$.

Paths traversing the second successor of a SPLIT node may only be followed if the evaluation of the first SPLIT successor succeeds. This corresponds to the standard goal selection rule. Therefore, we add intermediate goals to the DTs. These goals correspond to the evaluation of first SPLIT successors whenever we traverse a second SPLIT successor. However, for intermediate goals we use different predicate symbols than the ones we used for the head and last body goal of the DTs. For the path from A to F we then obtain the DT $\mathsf{q}(\mathsf{s}(T_6), T_8) \leftarrow \mathsf{r}(T_6, T_7), \mathsf{q}(T_7, T_8)$ using r as the predicate for the intermediate goals.

Now, for the evaluation of intermediate goals we must additionally consider so-called *program paths*. These are paths from successors of INSTANCE nodes or first successors of SPLIT nodes to SUC nodes, INSTANCE nodes, or successors of INSTANCE nodes. However, we can exclude paths traversing other first successors of SPLIT nodes as we are interested in successful evaluations only. In Fig. 2 we have two program paths: from A to F and from A to G. For these paths we generate clauses in the same way as for the DTs with the only exception that we only take the predicate symbol r used for intermediate goals. For SUC nodes, however, we have no body goal and generate facts.

The set of queries for the resulting DT problem contains all queries for the predicate corresponding to the initial state where those positions are assumed to be ground whose corresponding variable is known to represent ground terms in the initial state.

Thus, we obtain the DT problem $(\mathscr{D}_G, \mathscr{Q}_G, \mathscr{P}_G)$ from Fig. 3 for the termination graph G of Fig. 2. This DT problem is easily shown to be terminating by our automated termination prover AProVE.

[1]For simplicity, we use a set of initial queries instead of a general call set as in [15].

$$\begin{array}{rcl} \mathcal{D}_G = \{\mathsf{q}(\mathsf{s}(T_6),T_5) & \leftarrow & \mathsf{q}(T_6,U). \\ \mathsf{q}(\mathsf{s}(T_6),T_8) & \leftarrow & \mathsf{r}(T_6,T_7),\mathsf{q}(T_7,T_8).\} \\ \mathcal{P}_G = \{\mathsf{r}(\mathsf{s}(T_6),T_8) & \leftarrow & \mathsf{r}(T_6,T_7),\mathsf{r}(T_7,T_8). \\ \mathsf{r}(0,0).\} & & \end{array}$$

$\mathcal{Q}_G$ contains all queries $\mathsf{q}(t_1,t_2)$ where t_1 is a ground term.

Figure 3: DT problem for Ex. 1

We now state the central theorem of this paper where we prove that termination of the resulting DT problem implies termination of the original logic program with cut for the set of queries represented by the root state of the termination graph. For the proof we refer to [6].

Theorem 2 (Correctness)**.** *If G is a termination graph for a logic program $\mathcal{P}$ and a set of queries $\mathcal{Q}$ such that the DT problem for G is terminating, then $\mathcal{P}$ is terminating w.r.t. $\mathcal{Q}$.*

Note that the converse of this theorem does not hold.

4 Implementation and Experiments

We implemented the new transformation in our fully automated termination prover AProVE and tested it on all 402 examples for logic programs from the Termination Problem Data Base [19] used for the annual international Termination Competition [18]. We compared the implementation of the new transformation (AProVE DT) with the implementation of the previous transformation into definite logic programs from [16] (AProVE Cut), and with a direct transformation into term rewrite systems ignoring cuts (AProVE Direct) from [14]. We ran the different versions of AProVE on a 2.67 GHz Intel Core i7 and, as in the international Termination Competition, we used a time-out of 60 seconds for each example. For all versions we give the number of examples which could be proved terminating (denoted “Successes”), the number of examples where termination could not be shown (“Failures”), the number of examples for which the timeout of 60 seconds was reached (“Timeouts”), and the total runtime (“Total”) in seconds. All details of this empirical evaluation can also be seen online and one can run AProVE on arbitrary examples via a web interface [6].

	AProVE Direct	AProVE Cut	AProVE DT
Successes	243	259	**315**
Failures	144	129	**77**
Timeouts	15	14	**10**
Total	2485.7	3288.0	**2311.6**

Table 1: Experimental results on the TPDB

As shown in Table 1, the new transformation significantly increases the number of examples that can be proved terminating. In particular, we obtain 56 additional proofs of termination compared to [16]. And indeed, for all examples where AProVE Cut succeeds, AProVE DT succeeds, too.

In addition to being more powerful, the new version using dependency triples is also more efficient than any of the two other versions, resulting in fewer timeouts and a total runtime that is less than the one of the direct version and only 70% of the version corresponding to [16].

5 Conclusion

We have shown that the termination graphs introduced in [16] can be used to obtain a transformation from logic programs with cut to dependency triple problems. Our experiments show that this new approach is both considerably more powerful and more efficient than a translation to definite logic programs as in [16]. As the dependency triple framework allows a modular and flexible combination of arbitrary termination techniques from logic programming and even term rewriting, the new transformation to dependency triples can be used as a frontend to any termination tool for logic programs (by taking the union of $\mathscr{D}_G$ and $\mathscr{P}_G$ in the resulting DT problem $(\mathscr{D}_G, \mathscr{Q}_G, \mathscr{P}_G)$) or term rewriting (by using the transformation of [15]).

References

[1] K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, London, 1997.

[2] T. Arts and J. Giesl. Termination of Term Rewriting using Dependency Pairs. *Theoretical Computer Science*, 236(1,2):133–178, 2000.

[3] M. Bruynooghe, M. Codish, J. P. Gallagher, S. Genaim, and W. Vanhoof. Termination Analysis of Logic Programs through Combination of Type-Based Norms. *ACM Transactions on Programming Languages and Systems*, 29(2):Article 10, 2007.

[4] M. Codish, V. Lagoon, and P. J. Stuckey. Testing for Termination with Monotonicity Constraints. In *ICLP '05*, volume 3668 of *LNCS*, pages 326–340, 2005.

[5] D. De Schreye and S. Decorte. Termination of Logic Programs: The Never-Ending Story. *Journal of Logic Programming*, 19,20:199–260, 1994.

[6] Empirical evaluation and proofs for "Termination Analysis of Logic Programs with Cut Using Dependency Triples". `http://aprove.informatik.rwth-aachen.de/eval/cutTriples/`.

[7] J. Giesl. Termination of Nested and Mutually Recursive Algorithms. *Journal of Automated Reasoning*, 19:1–29, 1997.

[8] J. Giesl, R. Thiemann, and P. Schneider-Kamp. The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In *LPAR '04*, volume 3452 of *LNAI*, pages 301–331, 2005.

[9] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and Improving Dependency Pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.

[10] N. Hirokawa and A. Middeldorp. Automating the Dependency Pair Method. *Information and Computation*, 199(1,2):172–199, 2005.

[11] F. Mesnard and A. Serebrenik. Recurrence with Affine Level Mappings is P-Time Decidable for CLP(R). *Theory and Practice of Logic Programming*, 8(1):111–119, 2007.

[12] M. T. Nguyen, J. Giesl, P. Schneider-Kamp, and D. De Schreye. Termination Analysis of Logic Programs based on Dependency Graphs. In *LOPSTR '07*, volume 4915 of *LNCS*, pages 8–22, 2008.

[13] M. T. Nguyen, D. De Schreye, J. Giesl, and P. Schneider-Kamp. Polytool: Polynomial Interpretations as a Basis for Termination Analysis of Logic Programs. *Theory and Practice of Logic Programming*, 2010. To appear.

[14] P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated Termination Proofs for Logic Programs by Term Rewriting. *ACM Transactions on Computational Logic*, 10(1):2:1–49, 2009.

[15] P. Schneider-Kamp, J. Giesl, and M. T. Nguyen. The Dependency Triple Framework for Termination of Logic Programs. In *LOPSTR '09*, LNCS, 2010. To appear. Preliminary version and experimental details available from `http://aprove.informatik.rwth-aachen.de/eval/PolyAproVE/`.

[16] P. Schneider-Kamp, J. Giesl, T. Ströder, A. Serebrenik, and R. Thiemann. Automated Termination Analysis for Logic Programs with Cut. In *ICLP '10*, 2010. To appear. Preliminary version and experimental details available from `http://aprove.informatik.rwth-aachen.de/eval/cut/`.

[17] A. Serebrenik and D. De Schreye. On Termination of Meta-Programs. *Theory and Practice of Logic Programming*, 5(3):355–390, 2005.

[18] The Termination Competition. `http://www.termination-portal.org/wiki/Termination_Competition`.

[19] The Termination Problem Data Base 7.0 (December 11, 2009). `http://termcomp.uibk.ac.at/status/downloads/`.

[20] C. Walther. On Proving the Termination of Algorithms by Machine. *Artificial Intelligence*, 71(1):101–157, 1994.

Outermost Termination via Contextual Dependency Pairs

Bernhard Gramlich
TU Vienna, gramlich@logic.at

Felix Schernhammer*
TU Vienna, felixs@logic.at

1 Introduction and Overview

Recently, the problem of proving outermost termination has been addressed mainly by methods relying on transformations ([4, 1, 3]). Here we describe a more direct approach inspired by the dependency pair (DP) framework of [2]. The basic idea is to enrich dependency pairs by an additional component, namely the calling context of the corresponding recursive function call. Then one can use the additional contextual information to model DP chains adhering to certain strategies (e.g. the outermost strategy). Additionally, existing methods of the ordinary DP approach can partly be reused.

Building upon this framework of contextual DPs, we describe a DP processor exploiting the additional contextual information. Basically, this processor analyzes nested contexts accumulated by consecutive DPs on DP chain candidates for (certain) redexes. If such a redex is found, the chain candidate is not a proper chain. Finally, we provide some empirical evaluation of our approach.

2 Contextual Dependency Pairs

The central observation of the (ordinary) dependency pair approach is that, given a non-terminating rewrite system $\mathscr{R}$, there exists an infinite reduction sequence (starting w.l.o.g. with a root reduction step), such that no redex contracted in this sequence contains a non-terminating proper subterm. Such reduction sequences roughly correspond to minimal dependency pair chains whose (non-)existence is analyzed in the DP framework. In the case of outermost rewriting the above observation does not hold.

Example 1. *Consider the TRS $\mathscr{R}$ consisting of the rules*

$$a \to f(a) \qquad f(x) \to g(x)\,.$$

$\mathscr{R}$ is not outermost terminating: $a \to f(a) \to g(a) \to g(f(a)) \to g(g(a)) \to \ldots$ Here, it is crucial to reduce the term $f(a)$ at the root position, although it contains the term a as proper subterm which is not outermost terminating.

Example 1 shows that in the case of outermost rewriting it is sometimes crucial to allow reductions whose redexes properly contain outermost non-terminating terms. Hence, we restrict our attention to a restricted form of outermost termination, namely outermost termination in a context.

Definition 1 (outermost termination in a context)**.** *Let $\mathscr{R}$ be a TRS. A term s is* outermost terminating in context *$C[\Box]_p$ if $C[s]_p$ does not admit an infinite outermost reduction sequence where each redex contracted occurs at, below or parallel to p and where infinitely many of these steps are at or below p.*

Indeed, given a rewrite system $\mathscr{R}$, there exists an infinite outermost reduction sequence (starting w.l.o.g. with a root reduction step) contracting only redexes not containing proper subterms that are outermost non-terminating in their respective contexts, whenever $\mathscr{R}$ is outermost non-terminating.

In order to analyze outermost termination in an adapted dependency pair framework, we enrich the dependency pairs by an additional component, namely the context in which the corresponding recursive

*The author has been supported by the Austrian Academy of Sciences under grant 22.361.

function call takes place. Informally, this amounts to an extended *contextual* version of dependency pairs which incorporates the full information of the given rules (especially the complete right-hand sides) in the form of associated contexts, but which still enables the typical DP-based reasoning.

Definition 2 (*contextual* dependency pairs)**.** *Let $(\mathscr{F}, R)$ be a TRS where the signature is partitioned into defined symbols D and constructors. The set of (extended)* contextual dependency pairs (CDPs) *$CDP(\mathscr{R})$ is given by $DP_c(\mathscr{R}) \uplus V_c(\mathscr{R}) \uplus A_c(\mathscr{R}) \uplus S_c(\mathscr{R})$, where*

$$\begin{array}{rcl} DP_c(\mathscr{R}) & = & \{l^\# \to r|_p^\# \; [c] \mid l \to r \in R, p \in Pos_D(r), c = r[\Box]_p\} \\ V_c(\mathscr{R}) & = & \{l^\# \to T(r|_p) \; [c] \mid l \to r \in R, r|_p = x \in Var, c = r[\Box]_p\} \\ A_c(\mathscr{R}) & = & \{T(f(x_1,\dots,x_{ar(f)})) \to f^\#(x_1,\dots,x_{ar(f)}) \; [\Box] \mid l \to r \in R, root(r|_p) = f \in D\} \\ S_c(\mathscr{R}) & = & \{T(f(\vec{x})) \to T(x_i)[f(\vec{x})[\Box]_i] \mid \vec{x} = x_1,\dots,x_{ar(f)}, l \to r \in R, root(r|_p) = f, i \in \{1,\dots,ar(f)\}\}\,. \end{array}$$

Here, T is a new auxiliary function symbol (the token *symbol for "shifting attention"). We call $V_c(\mathscr{R})$* variable descent *CDPs, $S_c(\mathscr{R})$* shift *CDPs and $A_c(\mathscr{R})$* activation *CDPs.* Contextual *rules of the shape $l \to r \; [c]$ can be interpreted as $l \to c[r]$ when used as rewrite rules. Slightly abusing notation, for a set $\mathscr{P}$ of such contextual rewrite rules (i.e. a* contextual TRS*) we denote by $\to_{\mathscr{P}}$ the corresponding induced ordinary rewrite relation.*

Example 2. *Consider the TRS $\mathscr{R}$ of Example 1; $CDP(\mathscr{R})$ consists of:*

$$\begin{array}{cccc} a^\# \to a^\#[f(\Box)] & a^\# \to f^\#(a)[\Box] & f^\#(x) \to T(x)[g(\Box)] & T(a) \to a^\#[\Box] \\ T(f(x)) \to f^\#(x)[\Box] & T(g(x)) \to g^\#(x)[\Box] & T(f(x)) \to T(x)[f(\Box)] & T(g(x)) \to T(x)[g(\Box)]\,. \end{array}$$

Note that the restrictions imposed by the outermost strategy are not reflected in the definition of $CDP(\mathscr{R})$. Instead the strategy plays an important role in the definition of outermost CDP chains, which in turn are based on outermost CDP problems.

Definition 3 (outermost CDP problem)**.** *An* outermost contextual dependency pair problem *(O-CDP problem) is a triple $(\mathscr{P}, \mathscr{R}, T)$ where $\mathscr{P}$ is a contextual TRS, $\mathscr{R}$ a TRS and T is a designated function symbol not occurring in the signature of $\mathscr{R}$.*

Definition 4 (outermost CDP chain)**.** *Let $(\mathscr{P}, \mathscr{R}, T)$ be an O-CDP problem. A sequence $s_1 \to t_1 \; [c_1], s_2 \to t_2 \; [c_2], \dots$ of CDPs is an outermost $(\mathscr{P}, \mathscr{R}, T)$-CDP chain if there exists a substitution σ, such that*

$$\begin{array}{rcl} s_1\sigma & \to_{\mathscr{P}} & c_1[t_1\sigma]_{p_1} = c'_1[t_1\sigma]_{p'_1} \\ \xrightarrow{\not\leq p'_1 *}_{\mathscr{R}} c''_1[s_2\sigma]_{p'_1} & \to_{\mathscr{P}} & c''_1[c_2[t_2\sigma]_{p_2}]_{p'_1} = c'_2[t_2\sigma]_{p'_2} \\ \xrightarrow{\not\leq p'_2 *}_{\mathscr{R}} c''_2[s_3\sigma]_{p'_2} & \to_{\mathscr{P}} & c''_2[c_3[t_3\sigma]_{p_3}]_{p'_2} = c'_3[t_3\sigma]_{p'_3} \; \dots \end{array}$$

where $c'_i = c''_{i-1}[c_i]$ for all $1 \leq i$ and for each single reduction $s \xrightarrow{q}_{\mathscr{P}} t$ or $s \xrightarrow{q}_{\mathscr{R}} t$ $erase(s)$ does not contain a redex above q. Here $erase(s)$ is obtained by replacing all marked dependency pair symbols $f^\#$ by their unmarked versions f and by replacing terms $T(s')$ by s'[1]*. Moreover, the O-CDP chain is* minimal *if for all $i \geq 1$ the $\mathscr{R}$-reduction $c'_i[t_i\sigma]_{p'_i} \xrightarrow{\not\leq p'_i *}_{\mathscr{R}} c''_i[s_{i+1}\sigma]_{p'_i}$ is empty (i.e., if $c'_i[t_i\sigma]_{p'_i} = c''_i[s_{i+1}\sigma]_{p'_i}$) whenever $root(t_i) = T$ (i.e., the token symbol), and if for every $i \geq 0$ every subterm of $c'_i[t_i\sigma]_{p'_i}$ at position $q > p'_i$ is outermost terminating in its context (here: $s_1\sigma = c''_0[s_1\sigma]_{p'_0}$ with $p'_0 = \varepsilon$, $c''_0[\Box] = \Box$).*

We say an O-CDP problem is *finite* if it does not admit an infinite minimal O-CDP chain.

Theorem 1. *A TRS $\mathscr{R}$ is outermost terminating iff the O-CDP problem $(CDP(\mathscr{R}), \mathscr{R}, T)$ is finite.*

[1]Formally, this definition of *erase* is not compatible with the strict modularity of the DP framework. However, to restore full modularity the *erase* function could be part of an O-CDP problem. We refrain from doing so for notational simplicity.

Example 3. *Consider the TRS $\mathscr{R}$ from Example 1 ($CDP(\mathscr{R})$ is given in Example 2) and the corresponding O-CDP problem $P = (CDP(\mathscr{R}), \mathscr{R}, T)$. P admits an infinite O-CDP chain:*

$$a^{\#} \to f^{\#}(a)\ [\Box], f^{\#}(x) \to T(x)\ [g(\Box)], T(a) \to a^{\#}\ [\Box], \ldots$$

In the following we use the notions of sound resp. complete CDP processors analogously to corresponding notions of [2].

3 Analyzing Contexts

In this section we develop a method to prove the absence of minimal O-CDP chains (for a given O-CDP problem $(\mathscr{P}, \mathscr{R}, T)$) by inspecting the nested contexts of consecutive CDPs of candidates for infinite O-CDP chains.

The main problem here is that these contexts are not constant according to Definition 4. However, the (nested) contexts are stable modulo reduction parallel to the position of the hole, i.e. they are only altered through reductions parallel to the hole position. Hence, if the nested contexts contain redexes (strictly) above the hole position that are oblivious to this kind of parallel reductions, then the corresponding sequence of CDPs does not form an O-CDP chain.

To characterize (or rather approximate) these redexes we consider only those $\mathscr{R}$-rules whose left-hand sides are linear and not overlapped by any other $\mathscr{R}$-rule (strictly) below the root. This left-linear overlay sub-system $\mathscr{R}_{ioc}$ of $\mathscr{R}$ will be used for Invalidating (potential) Outermost CDP Chains.

Given an O-CDP problem $(\mathscr{P}, \mathscr{R}, T)$ and a sequence of *CDPs* S: $s_1 \to t_1[c_1], \ldots, s_n \to t_n[c_n]$, if $c_1[\ldots c_n[erase(t_n)]_{p_n} \ldots]_{p_1}$ contains a redex strictly above $p_1. \cdots .p_n$ w.r.t. $\mathscr{R}_{ioc}$, S is not an O-CDP chain.

Example 4. *Consider the TRS of Example 1. There is a CDP $\alpha : a^{\#} \to a^{\#}[f(\Box)]$. Here, $\mathscr{R}_{ioc} = \mathscr{R}$. Considering the sequence of CDPs consisting only of α (i.e. a CDP-sequence of length 1), the term $f(erase(a^{\#})) = f(a)$ is matched by the lhs $f(x)$ (strictly) above position 1 (namely at ε) and hence this sequence of CDPs (i.e. the CDP α) cannot be part of any infinite O-CDP chain.*

Now, in order to effectively check whether the nested contexts of a sequence of CDPs contain a redex above the hole position, we describe the possible nested contexts by a tree automaton. This is to say that, given a sequence of dependency pairs $s_1 \to s_n[c_1], \ldots, s_n \to t_n[c_n]$, we construct a tree automaton $\mathscr{A}$ accepting all terms $c_1[c_2[\ldots c_n[erase(t_n)] \ldots]]\sigma$ where σ is some substitution.

Moreover, we construct another tree automaton $\mathscr{B}$ that accepts *all* terms that are matched by any rule from $\mathscr{R}_{ioc}$. Then, the idea roughly is to check whether the language accepted by $\mathscr{A}$ is a sublanguage of $\mathscr{B}$ and thus to conclude that the sequence of CDPs is not a proper O-CDP chain.

We discuss the construction of the involved automata in more detail, starting with the one that accepts a term if some subterm of it is matched by the left-hand side of some rewrite rule. Starting from an O-CDP problem $(\mathscr{P}, \mathscr{R}, T)$, the signature Σ' of the automaton we are going to construct is the signature of $\mathscr{R}$ plus $\{H, A\}$ where H is a new function symbol of arity one and A is a new constant. The role of the symbol H is to indicate the positions in accepted terms where no reductions may take place (because there is a more outer redex), and A is needed since the automata we use work on ground terms only.

Definition 5 (FP automaton)**.** *Let $\mathscr{R}$ be a left-linear TRS over a signature Σ and let $\Sigma' = \Sigma \uplus \{H, A\}$ where H is unary and A is a constant. The* forbidden pattern automaton *$FPA(\mathscr{R})$ is given by $\bigcup_{l \to r \in \mathscr{R}} \mathscr{A}^l$ where $\mathscr{A}^l$ is the automaton accepting ground terms having subterms of the shape $l\sigma[H(l\sigma|_p)]_p$ ($p > \varepsilon$).*

Example 5. *Consider the TRS $\mathscr{R}$ given by*

$$a \to f(b) \qquad b \to g(b) \qquad b \to g(a) \qquad f(g(x)) \to \bot\,.$$

$FPA(\mathscr{R})$ consists of the following transitions.

$$\begin{array}{ccccccc} a \to q_1 & A \to q_1 & b \to q_1 & f(q_1) \to q_1 & g(q_1) \to q_1 & H(q_1) \to q_2 & f(q_2) \to q_2 \\ g(q_2) \to q_2 & g(q_2) \to q_3 & g(q_1) \to q'_3 & H(q'_3) \to q_3 & f(q_3) \to q_{end} & f(q_{end}) \to q_{end} & g(q_{end}) \to q_{end} \end{array}$$

The state q_{end} is the unique end state. Here, state q_1 corresponds to arbitrary terms without any occurrence of H and q_2 corresponds to arbitrary terms with an occurrence of H. Moreover, q_3 either corresponds to g-rooted terms containing an H or instances of $H(g(x))$ while q'_3 corresponds to g-rooted terms not containing an H.

Next, we discuss the construction of an automaton accepting terms obtained by nesting contexts of subsequent CDPs in sequences of CDPs.

Definition 6 (CDP automaton)**.** *Let $(\mathscr{P},\mathscr{R},T)$ be an O-CDP problem for a given finite $\mathscr{R} = (\Sigma, R)$ and let $S\colon s_1 \to t_1,[c_1]\dots,s_n \to t_n[c_n]$ be a sequence of CDPs. The CDP automaton $DPA((c_1,\dots,c_n),\Sigma,t_n)$ corresponding to this CDP sequence is given by $\mathscr{A}_{c_1[\dots c_n[H(erase(t_n))]]}$. Here, $\mathscr{A}_t$ is the automaton accepting ground terms $t\sigma$ over the signature $\Sigma \cup \{H,A\}$ where $x\sigma = A$ for all $x \in Var(t)$.*

Replacing variables by just one new constant is justified, since we are only interested in matches by the left-hand sides of $\mathscr{R}_{ioc}$ (which is left-linear).

Theorem 2 (analyzing nested contexts)**.** *Let $(\mathscr{P}, \mathscr{R}, T)$ be an O-CDP problem for a given finite $\mathscr{R} = (\Sigma,R)$ and let $S\colon s_1 \to t_1,[c_1]\dots,s_n \to t_n[c_n]$ be a sequence of CDPs. If $L(DPA((c_1,\dots,c_n),\Sigma,t_n)) \subseteq L(FPA(\mathscr{R}_{ioc}))$, then S cannot be part of an infinite O-CDP chain.*

In order to use Theorem 2 in termination proofs and in particular in the DP framework, we have to consider candidates for CDP chains in a complete way. This is to say that for one CDP π we consider all potential sequences of CDPs that start from π and eventually use it again. If no such sequence can be part of an infinite CDP chain, then it is sound to delete π.

The rest of this section is concerned with describing the (in general infinitely many) candidate chains, or, rather, the corresponding nested contexts, in a finite way. Starting from existing dependency graph approximations, we identify cycles in this graph and describe them as combinations of *minimal cycles*. Here, a cycle is a sequence of nodes with identical start and end node, which is minimal if each node except the start node occurs at most once and the start node occurs only at the beginning and at the end.

By $node(C)$ we denote the start node of cycle C. Arbitrary cycles are combinations of minimal cycles. These combinations can be represented by trees called *minimal cycle combinations*.

Definition 7 (minimal cycle combination)**.** *Let $\mathscr{G}$ be a (finite) graph and let $C_1,\dots,C_n$ be the set of minimal cycles of $\mathscr{G}$. A* minimal cycle combination *(MCC) is a tree whose nodes are minimal cycles and whose edges $C_i - C_j$ are labeled by a positive integer i from $\{1,\dots,|C_i|\}$ if C_j is a (minimal) cycle for the i^{th} node of C_i. A minimal cycle combination is* hierarchical *if $node(C_i) \notin C_j$ whenever C_j is (in the tree) below C_i.*

We say an MCC *is for node n* if its root node is a cycle with start node n. The depth of a hierarchical MCC is bounded by the number of nodes in the graph, and thus finite in most interesting cases when considering DP graph approximations. Every MCC corresponds to a set $Cyc(M)$ of cycles obtained by recursively combining the contained minimal cycles.

Example 6. *Consider the TRS $\mathscr{R}$ of Example 5 We consider the subset $\{\alpha\colon a^\# \to b^\#[f(\Box)], \beta\colon b^\# \to b^\#[g(\Box)], \gamma\colon b^\# \to a^\#[g\Box]\}$ of $CDP(\mathscr{R})$. A dependency graph approximation might contain the edges (α,β), (α,γ), (β,γ), (β,β), (γ,α). The minimal cycles in this graph are $C_1 = (\alpha,\beta,\gamma,\alpha)$, $C_2 = (\alpha,\gamma,\alpha)$ and $C_3 = (\beta)$ (modulo rotations). For α there are two MCCs, namely M_1 consisting of only one (root) node C_2 and M_2 consisting of the node C_1 with one child C_3 connected by an edge with label 2. We have $Cyc(M_1) = \{C_2\}$ and $Cyc(M_2) = \{(\alpha,\beta^n,\gamma,\alpha) \mid n \geq 1\}$.*

For a finite graph $\mathscr{G}$ there is only a finite number of hierarchical MCCs having different sets of associated cycles. Moreover, for every given cycle C, there is a hierarchical MCC M such that $C \in Cyc(M)$. Since a cycle in a DP graph approximation corresponds to a sequence of CDPs, it also corresponds to a sequence of nested contexts. We denote the set of nested contexts corresponding to the cycles $Cyc(M)$ of an MCC M as $ctxs(M)$. Slightly abusing notation we write $ctxs(M)[t]$ for the set of terms given by $\{c[t] \mid c \in ctxs(M)\}$. Now, given a hierarchical MCC M w.r.t. a CDP graph and a term t, we can construct a tree automaton $MCCA(M,t)$ accepting all terms from $ctxs(M)[H(t)]$ (where the variables have been replaced by A). This is the crucial step of describing the infinite set $ctxs(M)[H(t)]$ finitely by $MCCA(M,t)$.

Example 7. *Consider the TRS, CDPs and MCCs of Example 6. $MCCA(M_2,a)$ consists of the following transitions (the contexts of $ctxs(M_2)[a]$ have the shape $f(g^+(g(f(a))))$ where the end state is q_{end}.*

$$\begin{array}{ccccc} A \to q_1 & a \to q_1 & H(q_1) \to q_2 & f(q_2) \to q_3 & g(q_3) \to q_4 \\ g(q_4) \to q_5 & g(q_5) \to q_5 & f(q_5) \to q_{end} & & \end{array}$$

Theorem 3 (Context Processor)**.** *Let $\Pi = (\{s \to t\ [c]\} \uplus \mathscr{P}, \mathscr{R}, T)$ be a CDP problem with $\mathscr{R} = (\Sigma, R)$ finite. If for every hierarchical MCC M for $s \to t$ in some CDP graph approximation we have $L(MCCA(M),t) \subseteq L(FPA(\mathscr{R}_{ioc}))$, then Π is finite if and only if $(\mathscr{P}, \mathscr{R}, T)$ is finite.*

Example 8. *Consider the TRS $\mathscr{R}$, CDPs and MCCs of Example 6, the automata $MCCA(M_2,a)$ and $MCCA(M_1,a)$ (the former is given in Example 7) and the automaton $FPA(\mathscr{R}_{ioc})$ from Example 5 (note that here $\mathscr{R}_{ioc} = \mathscr{R}$). We have $L(MCCA(M_i),a) \subseteq L(FPA(\mathscr{R}_{ioc}))$ for $i \in \{1,2\}$. Hence, it is sound to delete α. Note that β cannot be deleted and, indeed, the infinite sequence of CDPs consisting of only β CDPs is an infinite O-CDP chain.*

4 Evaluation and Conclusion

In a prototypic implementation of the CDP framework we performed some benchmark tests on the TRSs of the outermost section of the TPDB. In addition to the context processor of Section 3 we also used various standard DP processors not relying on minimality of DP problems, such as the polynomial ordering processors without usable rules. Out of 133 potentially terminating TRSs in the outermost category of the TPDB our implementation was able to prove outermost termination of 60 systems. To put this in perspective, in the termination competition of 2009 the three participating tools obtained 72 (Jambox), 46 (TrafO) and 27 (AProVE) successful proofs of outermost termination.

The modularity of the CDP framework is inherited from the ordinary DP framework. Hence, also in this extended framework new processors based on new ideas and tailored for specific classes of problems can be modularly added. Regarding future work, among generalizations of standard DP processors for the adapted framework an outermost usable rules criterion based on the restricted minimality property of minimal O-CDP chains would be very interesting.

References

[1] J. Endrullis and D. Hendriks. From outermost to context-sensitive rewriting. In *Proc. 20th International Conference on Rewriting Techniques and Applications, RTA'09*, pages 305–319, 2009. Springer-Verlag.

[2] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *J. Autom. Reason.*, 37(3):155–203, 2006.

[3] M. Raffelsieper and H. Zantema. A transformational approach to prove outermost termination automatically. *Electron. Notes Theor. Comput. Sci.*, 237:3–21, 2009.

[4] R. Thiemann. From outermost termination to innermost termination. In *Proc 35th Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM'09*, pages 533–545, 2009. Springer-Verlag.

Scaling termination proofs by a characterisation of cycles in CHR

Paolo Pilozzi
paolo.pilozzi@cs.kuleuven.be [*]

Danny De Schreye
danny.deschreye@cs.kuleuven.be

CHR, short for Constraint Handling Rules, is a rule-based programming language, with similarities to Term Rewrite Systems (TRS) and Logic Programming (LP). In termination analysis of rule-based languages, the concept of a cycle is often useful. Informally, a cycle is a finite sequence of rules such that these rules can potentially be repeatedly executed during a computation. Above, we write "potentially" because cycles usually involve approximations. In cycles, whether one rule can actually activate the next is only verified locally (e.g. through matching or unification). But whether all these conditions hold simultaneously — so that the full cycle can be repeatedly traversed — is unknown.

There can be several advantages of using cycles in termination analysis. They are mostly due to the fact that an analysis based on cycles is more modular. One advantage is reduced complexity. Some termination analysis techniques do not scale well for large programs. By performing the proof for separate cycles, sizes remain reduced. Another potential advantage is precision. When proving termination of a program as a whole, one typically selects a specific well-founded order. In the context of cycles, one can use different orders for different cycles. As a result, techniques based on cycles can be very successful (e.g. [4]).

We have studied the introduction of cycles in CHR. However, we were confronted with the fact that, due to the multi-headed rules and a multiset semantics in CHR, it is hard to define a suitable notion of "cycle". Consider a CHR program with one rule, $a,c \Leftrightarrow a$. Although one could consider it to be a cycle, in an actual computation it cannot result in an infinite loop, since every application removes a $c/0$ constraint. Similarly, a CHR program with one rule, $a,c,c \Leftrightarrow a,a,c$ is terminating and the rule would better not be characterised as a cycle. Considering a program with two rules, $a,a \Leftrightarrow b,b$ and $b \Leftrightarrow a$, we should identify a cycle. However, this cycle should contain the first rule together with two copies of the second rule.

One of our initial attempts to define cycles was very intuitive and rather precise, but it turned out that only for specific uses of a CHR program a concept of minimality existed, resulting in general in an infinite number of different minimal cycles. This is of course unacceptable. In this paper, we provide a less precise but useful definition of a cycle yielding a finite number of minimal cycles, and discuss its successful use in CHR termination analysis.

1 CHR syntax and semantics

First, we introduce the syntax and semantics of CHR [3]. The atoms of a CHR program, called *constraints*, are first-order relations on terms, collected in a *constraint store*. There are two kinds of constraints: *built-in constraints*, pre-defined in the host-language (CT, for Constraint Theory), and user-defined *CHR constraints*.

We assume presence of only one kind of CHR rule, that is, a *simplification rule*:

$$R_i \ @ \ H_{(i,1)},\ldots,H_{(i,n_i)} \Leftrightarrow G_{(i,1)},\ldots,G_{(i,u_i)} \mid B_{(i,1)},\ldots,B_{(i,v_i)},C_{(i,1)},\ldots,C_{(i,m_i)}.$$

Here, $H_{(i,1)},\ldots,H_{(i,n_i)}$ are head constraints, representing the CHR constraints from the store that can be used to fire the rule. For rule application, given *matching* constraints for the heads, the built-in constraints $G_{(i,1)},\ldots,G_{(i,u_i)}$ need to be satisfiable. Upon rule application, new built-in constraints, $B_{(i,1)},\ldots,B_{(i,v_i)}$, and CHR constraints $C_{(i,1)},\ldots,C_{(i,m_i)}$ are added to the constraint store, replacing the head constraints.

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

The constraint store of a CHR program represents a conjunction of constraints in which multiple occurrences of the same constraint are allowed. The constraint store thus behaves as a *multiset*.

Definition 1 (Multiset)**.** *A* multiset *is a tuple, $M_S = \langle S, m_S \rangle$, where S is a regular set, called the* underlying set *of elements, and m_S a* multiplicity function*, mapping the elements, e, of S to strict positive integer numbers, $m_S(e) \in \mathbb{N}_0$, representing the number of occurrences of e in M_S.* □

We may define m_S as its graph, i.e. the set of ordered pairs $\{(s, m_S(s)) : s \in S\}$. Therefore, the multiset written as $[\![a,a,b]\!]$ is defined as $\langle \{a,b\}, \{(a,2),(b,1)\} \rangle$, and the multiset $[\![a,b]\!]$ as $\langle \{a,b\}, \{(a,1),(b,1)\} \rangle$. For adding multiset together, we introduce *multiset join* and denote it by $\uplus$.

Since we do not consider propagation, the state transition system can be given by the *abstract CHR semantics*, where a *CHR state S* is a *constraint store* and only two kinds of transitions are defined:

1. For any state $[\![b]\!] \uplus S$, where b is a built-in constraint satisfiable in the host-language with substitution θ, a transition $([\![b]\!] \uplus S, S\theta)$ exists.

2. For any state $[\![h_1, \ldots, h_n]\!] \uplus S$, where $h_1, \ldots, h_n$ are CHR constraints matching with the head of a rule $H_1, \ldots, H_n \Leftrightarrow G_1, \ldots, G_k \mid B_1, \ldots, B_l, C_1, \ldots, C_m$ resulting in a substitution σ, such that the built-in constraints $G_1\sigma, \ldots, G_k\sigma$ are satisfiable in the host-language (CT) with substitution θ, a transition $([\![h_1, \ldots, h_n]\!] \uplus S, ([\![B_1, \ldots, B_l, C_1, \ldots, C_m]\!] \uplus S)\sigma\theta)$ exists.

We assume that any call to the host-language can only introduce variable bindings and must do so in finite time. If a built-in constraint cannot be solved, the CHR program fails. CHR is furthermore a committed choice language. Therefore, whenever a rule is applied — which is a non-deterministic choice between all possible applications of the CHR rules of the program — we commit on this choice.

2 CHR cycles

The rules of a CHR program relate CHR constraints. The head constraints of a rule represent the constraints *required* for rule application. The added CHR constraints represent the constraints *available* after rule application. To denote these sets of constraints, we introduce *abstract CHR constraints*.

Definition 2 (Abstract CHR constraint)**.** *An abstract CHR constraint $\mathscr{C} = (C, B)$ is a pair, where C is a CHR constraint and B a conjunction of built-in constraints. Given a mapping $\wp : (C, B) \mapsto \{C\sigma \mid \exists\theta : CT \models B\sigma\theta\}$, an abstract CHR constraint represents a set of CHR constraints.* □

Two kinds of abstract CHR constraints are present in a CHR program. An *abstract input constraint* $in_{(i,j)} = (H_{(i,j)}, G_i)$, where $G_i = G_{(i,1)} \wedge \ldots \wedge G_{(i,u_i)}$, represents a set of CHR constraints matching with the head $H_{(i,j)}$, possibly resulting in rule application. An *abstract output constraint* $out_{(i,j)} = (C_{(i,j)}, G_i)$ represents the CHR constraints which become available after rule application.

Example 1 (Greatest Common Divisor)**.** *The following CHR program, P, implements the Euclidian algorithm for computing the greatest common divisor (GCD).*

$$R_1 \ @ \ gcd(0) \Leftrightarrow true.$$
$$R_2 \ @ \ gcd(M), gcd(N) \Leftrightarrow N \geq M, M > 0 \mid L \ is \ N - M, gcd(M), gcd(L).$$

The first rule removes $gcd(0)$ constraints, while the second replaces two $gcd/1$ constraints by simpler $gcd/1$ constraints. In the following, we denote by $\mathscr{I}n_{R_i}$ and $\mathscr{O}ut_{R_i}$ the multisets of abstract input constraints and output constraints of a rule R_i, and by $\mathscr{I}n_P$ and $\mathscr{O}ut_P$, the multisets of abstract input and output constraints of a set of rules, P. For the GCD program we obtain:

$$\begin{array}{rcl}\mathscr{I}n_P & = & \mathscr{I}n_{R_1} \uplus \mathscr{I}n_{R_2} = [\![in_{(1,1)}, in_{(2,1)}, in_{(2,2)}]\!], \text{ where} \\ & & in_{(1,1)} = (gcd(0), true) \\ & & in_{(2,1)} = (gcd(M), N \geq M \wedge M > 0) \\ & & in_{(2,2)} = (gcd(N), N \geq M \wedge M > 0) \\ \mathscr{O}ut_P & = & \mathscr{O}ut_{R_1} \uplus \mathscr{O}ut_{R_2} = [\![out_{(2,1)}, out_{(2,2)}]\!], \text{ where} \\ & & out_{(2,1)} = (gcd(M), N \geq M \wedge M > 0) \\ & & out_{(2,2)} = (gcd(L), N \geq M \wedge M > 0)\end{array} \qquad \square$$

The rules of a CHR program relate abstract inputs to abstract outputs.

Definition 3 (Rule transition relation)**.** *A rule transition of a CHR program P is an ordered pair $T_i = (\mathscr{I}n_{R_i}, \mathscr{O}ut_{R_i})$, relating the multiset of abstract input constraints $[\![in_{(i,1)}, \ldots, in_{(i,n_i)}]\!] = \mathscr{I}n_{R_i}$ of a rule $R_i \in P$ to the multiset of abstract output constraints $[\![out_{(i,1)}, \ldots, out_{(i,m_i)}]\!] = \mathscr{O}ut_{R_i}$ of R_i. The rule transition relation $\mathscr{T} = \{T_i \mid R_i \in P\}$ of a CHR program P is the set of rule transitions T_i of a program P.* □

Example 2 (GCD cont.)**.** *The GCD program has two rules and thus two rule transitions:*

$$\mathscr{T} = \{T_1, T_2\}, \text{ where } T_1 = ([\![in_{(1,1)}]\!], [\![]\!]) \text{ and } T_2 = ([\![in_{(2,1)}, in_{(2,2)}]\!], [\![out_{(2,1)}, out_{(2,2)}]\!]). \qquad \square$$

Abstract outputs relate to inputs by a *match transition relation*, given by a dependency analysis.

Definition 4 (Match transition relation)**.** *A match transition of a CHR program P is an ordered pair $M_{(i,j,k,l)} = (out_{(i,j)}, in_{(k,l)})$, expressing a dependency between an output $out_{(i,j)} = (C_{(i,j)}, G_i)$ of $\mathscr{O}ut_P$ and an input $in_{(k,l)} = (C_{(k,l)}, G_k)$ of $\mathscr{I}n_P$, for which $\exists \theta : CT \models ((C_{(i,j)} = C_{(k,l)}) \wedge G_i \wedge G_k)\theta$ holds. The match transition relation $\mathscr{M}$ is the set of all match transitions $M_{(i,j,k,l)}$ in P.* □

Notice that a match transition exists if the output and input it connects represent sets of constraints with a non-empty intersection: $\wp(out_{(i,j)}) \cap \wp(in_{(k,l)}) \neq \emptyset$.

Example 3 (GCD cont.)**.** *We obtain for the GCD program the following match transition relation:*

$$\mathscr{M} = \{M_{(2,1,2,1)}, M_{(2,1,2,2)}, M_{(2,2,1,1)}, M_{(2,2,2,1)}, M_{(2,2,2,2)}\}.$$

Note that the first output of the second rule does not match with the input of the first rule as their intersection $\wp(out_{(2,1)}) \cap \wp(in_{(1,1)}) = \emptyset$. □

For a program P, we call $\mathscr{N}_P = \langle \mathscr{I}n_P, \mathscr{O}ut_P, \mathscr{T}_P, \mathscr{M}_P \rangle$ its *CHR net*.

2.1 CHR cycles

To discuss cycles, we need an abstraction for computations, one for which a concept of minimality exists. For this purpose, we introduce *CHR constructs*. A CHR construct is an ordered pair of multisets. The first multiset represents the number of applications of some rule in a subcomputation of a program. The second multiset represents matches within the subcomputation. Note that if a *universe* U, in which the elements of the multiset $M_S = \langle S, m_S \rangle$ must live, is specified, that we can replace m_S by a function $\mu_S : U \to \mathbb{N}$, obtained by extending m_S to U with values 0 outside S.

Definition 5 (Cyclic CHR construct)**.** *Let $\mathscr{N}_P = \langle \mathscr{I}n_P, \mathscr{O}ut_P, \mathscr{T}_P, \mathscr{M}_P \rangle$ be the CHR net of P. Then, a* CHR construct *is a pair of multisets $((P, \mu), (\mathscr{M}_P, \mu'))$: a multiset of rules (P, μ) and a multiset of match transitions $(\mathscr{M}_P, \mu')$; such that:*

$$\forall in_{(i,j)} \in \mathscr{I}n_P : \sum_{M_{(k,l,i,j)} \in \mathscr{M}_P} \mu'(M_{(k,l,i,j)}) \leq \mu(R_i) \qquad \forall out_{(i,j)} \in \mathscr{O}ut_S : \sum_{M_{(i,j,k,l)} \in \mathscr{M}_P} \mu'(M_{(i,j,k,l)}) \leq \mu(R_i)$$

A cyclic CHR construct *or* CHR cycle *is a CHR construct where additionally*

$$\forall in_{(i,j)} \in \mathscr{I}n_C : \sum_{M_{(k,l,i,j)} \in \mathscr{M}_P} \mu'(M_{(k,l,i,j)}) = \mu(R_i).$$

We say that a CHR construct is traversed, if every rule in it was applied once according to the match transitions of the construct. □

Notice that for a CHR construct to correspond to an actual computation, we need to constrain the number of outgoing match transitions for a given output to the number of times its rule occurs in the multiset of rule applications. Similarly, we need to constrain the number of incoming match transitions for a given input to the number of times its rule occurs in the multiset of rule applications. This becomes clear when regarding the next example computation for the GCD program from Example 1.

Example 4 (GCD cont.)**.** *Given a query $I = [gcd(6), gcd(3)]$, the following sequence represents a computation of P for I.*

$$[gcd(6), gcd(3)] \mapsto_{R_2} [gcd(3), gcd(3)] \mapsto_{R_2} [gcd(0), gcd(3)] \mapsto_{R_1} [gcd(3)]$$

A different representation of this computation uses concepts of CHR nets, as is shown in Figure 1. The

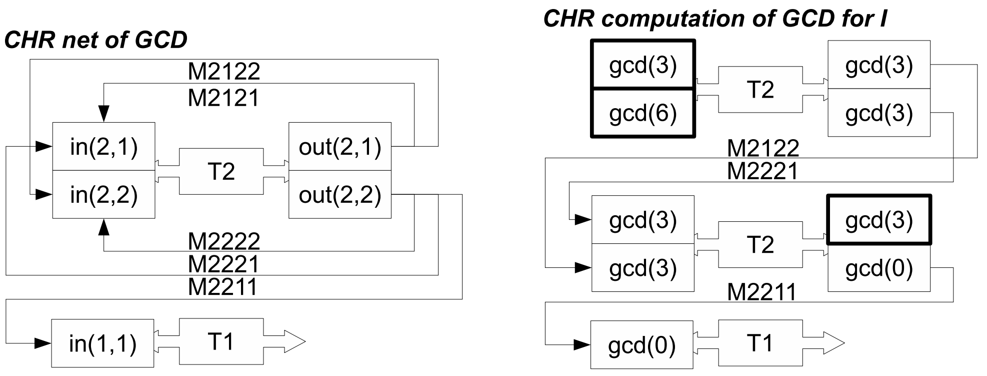

Figure 1: CHR net of the GCD program and a possible computation of GCD for *I*.

inputs without an incoming match transition, in the CHR construct representing a computation of P for I, represent the constraints from the query . Outputs without an outgoing match transition represent the constraints part of the answer. □

Finally, a CHR construct is cyclic if all inputs are provided for from within the construct. Thus, a cyclic construct replaces the constraints which are removed when traversing the construct.

The set of all possible (cyclic) CHR constructs can be represented using a system of linear inequalities. Such a system has a unique computable basis of solutions, called a *Hilbert basis*. From it, we can compute any solution to the system by positive linear combinations of the solution vectors.

Example 5 (GCD cont.)**.** *For the GCD program, we obtain the following system of inequalities, describing the cyclic CHR constructs of the program. Note that m_{R_i} represents $\mu(R_i)$, the number of rule applications of R_i in the construct, and that $m_{(i,j,k,l)}$ represents $\mu'(M_{(i,j,k,l)})$ the number of match transitions $M_{(i,j,k,l)}$ in the construct.*

$$\begin{array}{ll} m_{(2,2,1,1)} = m_{R_1} & m_{(2,1,2,1)} + m_{(2,1,2,2)} \le m_{R_2} \\ m_{(2,1,2,1)} + m_{(2,2,2,1)} = m_{R_2} & m_{(2,2,2,1)} + m_{(2,2,2,2)} \le m_{R_2} \\ m_{(2,1,2,2)} + m_{(2,2,2,2)} = m_{R_2} & \end{array}$$

By adding slack variables we obtain a system of equations with positive integer solutions, which allows us to compute the following Hilbert basis.

$$([\![R_2]\!], [\![M_{(2,1,2,2)}, M_{(2,2,2,1)}]\!]) \qquad ([\![R_2]\!], [\![M_{(2,1,2,1)}, M_{(2,2,2,2)}]\!])$$

We can obtain every cyclic construct of a CHR program by joining elements from the basis (Figure 2). □

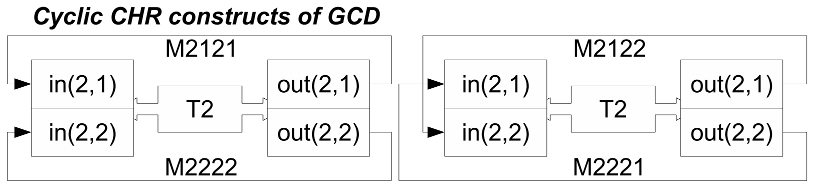

Figure 2: Cyclic CHR constructs of the GCD program.

By considering now the cycles in the basis (See Figure 2), we can derive the subsets of rules that demonstrate cyclic behaviour. For the GCD example, there is only one such subset, $\{R_2\}$, that is cyclic.

3 Evaluation and conclusion

We described an approach for analysing the cycles of a CHR program, as such safely over-approximating the loops of the program. For termination, it is sufficient to analyse only cycles and not the entire program, resulting in faster and more accurate termination proofs. We have developed an implementation for deriving the CHR net of a program, for performing a Strongly Connected Component (SCC) analysis on the CHR net of the program, and for generating systems of linear equations for each of the SCCs.

Our approach for cycle analysis in CHR is useful, however, only when regarding sufficiently large programs, consisting of multiple cyclic subsets of rules. Our approach scales well, using first a SCC analysis to make an initial estimate of the possible cycles of a program. Then, only afterwards for the remaining components, a classification is obtained by computing the basis [5] of solutions. If empty, the component is not a cycle, otherwise, it is a cycle and is analysed for termination.

Our representation of cycles can be used for more refined approaches to termination analysis of CHR as well, e.g. approaches concerning dependency pairs [1] and Ramsey's theorem [2]. Future work will involve investigating whether these approaches to termination analysis can be ported to the CHR context.

References

[1] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Electronic Notes in Theoretical Computer Science*, 236(1-2):133–178, 2000.

[2] Nachum Dershowitz, Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. A General Framework for Automatic Termination Analysis of Logic Programs. *CoRR*, cs.PL/0012008, 2000.

[3] Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.

[4] Manh Thang Nguyen, Jürgen Giesl, Peter Schneider-Kamp, and Danny De Schreye. Termination analysis of logic programs based on dependency graphs. pages 8–22, 2007.

[5] Dmitrii V. Pasechnik. On computing the Hilbert base via the Elliott-MacMahon algorithm. *Electronic Notes in Theoretical Computer Science*, 263(1-2):37–46, 2001.

A New Path Order for Exponential Time[*]

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

Naohi Eguchi
School of Information Science
Japan Advanced Institute of Science and Technology, Japan
n-eguchi@jaist.ac.jp

Abstract

In this note we present the *Exponential Path Order* EPO*. Inspired by a novel term rewriting characterisation of the exponential time functions FEXP, this order is carefully trimmed so that we believe that compatibility of TRSs implies exponentially bounded runtime complexity. Moreover, the order is complete in the sense that every exponential time function can be expressed by a TRS compatible with an instance of EPO*.

The work on EPO* is still unfinished, but we strongly believe that above mentioned results are correct.

1 Introduction

Bellantoni and Cook [4] define the class $\mathcal{B}$ as the least class of functions containing certain initial functions, and which is closed under the schemes of *safe recursion on notation* and *safe composition*. In this spirit, exactly the *polytime computable functions* FP are generated, i.e., the class $\mathcal{B}$ coincides with the class FP. Unlike previous definitions of FP (for instance, the recursion-theoretic characterisation given by Cobham [5]), the class $\mathcal{B}$ is defined without explicitly referring to any externally imposed resource bounds. Instead, the strength of the recursion scheme is broken by a syntactic separation of arguments positions into *safe* and *normal* ones. To highlight this separation, we write $f(\vec{x};\vec{y})$ instead of $f(\vec{x},\vec{y})$ for normal arguments $\vec{x}$ and safe arguments $\vec{y}$. Suppose functions g, h_0 and h_1 as well as functions h, $\vec{r}$ and $\vec{s}$ are definable in $\mathcal{B}$. Then a new function f is defined either by *safe recursion on notation* via the equations

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

or by *safe composition* via the equation

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

The purpose of the separation is to disallow recursion on recursively computed results: The recursion parameter in (SNR) is taken from a normal argument position, whereas the recursively computed result $f(z,\vec{x};\vec{y})$ is substituted into a safe argument position of the stepping function h_i. For instance, it is not possible to define an exponentiation like function exp via the equations

$$\mathsf{double}(\epsilon) = \epsilon \qquad \mathsf{double}(zi) = \mathsf{double}(z)ii \qquad \mathsf{exp}(\epsilon) = 1 \qquad \mathsf{exp}(zi) = \mathsf{double}(\mathsf{exp}(z))\ .$$

Since the function double is defined by recursion, the single argument of double needs to be normal. Consequently, the function double cannot be used in the definition of exp. The additional restrictions on argument positions imposed by scheme (SNC) ensures that safe arguments cannot influence normal ones.

Inspired by the results of Bellantoni and Cook, Arai and the second author define in [1] the class $\mathcal{N}$ as the least class containing the initial functions of $\mathcal{B}$ and that is closed under the (modified) scheme of *safe composition*

$$f(\vec{x};\vec{y}) = h(x_{i_1},\dots,x_{i_k};\vec{s}(\vec{x};\vec{y}))\ , \tag{SNC$_2$}$$

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

and *safe nested recursion on notation*

$$\begin{aligned} f(\vec{\epsilon},\vec{x};\vec{y}) &= g(\vec{x};\vec{y}) \\ f(\vec{z},\vec{x};\vec{y}) &= h_{\tau(\vec{z})}(\vec{v_1},\vec{x};\vec{y},f(\vec{v_1},\vec{x};\vec{t}_{\tau(\vec{z})}(\vec{v_2},\vec{x};\vec{y},f(\vec{v_2},\vec{x};\vec{y})))) . \end{aligned} \tag{SNRN}$$

In the recursion scheme, recursion is preformed simultaneously on multiple arguments. The functions $h_{\tau(\vec{z})}$ and $\vec{t}_{\tau(\vec{z})}$ are previously defined functions, chosen in terms of $\tau(\vec{z}) \in \Sigma_0^k$ where k is the length of $\vec{z}$ (Σ_0^k refers to the set of binary strings of length k). Further, $\vec{v_1}$ and $\vec{v_2}$ are unique predecessors of $\vec{z}$ defined in terms of τ, satisfying $\vec{v_1} <_{\mathsf{lex}} \vec{z}$ and $\vec{v_2} <_{\mathsf{lex}} \vec{z}$. Observe that the scheme (SNRN) is a syntactic extension of the scheme (SNR). Note also, that the modification of safe composition is necessary as FEXP is, opposed to FP, *not* closed under composition. For missing details on the definition of the scheme (SNRN) we kindly refer the reader to [1]. Instead, we give an illustrating example. The simplest exponentially growing function definable by safe nested recursion on notation is

$$f(\epsilon;y) = y1 \qquad\qquad f(xi;y) = f(x;f(x;y)) \quad (i = 0,1) .$$

Then it can be verified that $f(x;y) = y1^{2^{|x|}}$ where 1^n denotes n times concatenation of the symbol 1. Let FEXP denote the class of *functions computable in exponential time*. In [1] it is proved that the class $\mathcal{N}$ coincides with FEXP [1]. As a consequence, we conclude $f \in \mathsf{FEXP}$ for above defined function f.

Hofbauer has shown that *multiset path orders (MPOs for short)* induce primitive recursive bounds on the length of derivations (see [7]). The schemes of safe recursion on notation suitably tames primitive recursion so that only polytime computable functions are generated. Combining those two observation, Moser and the first author have shown that the separation of safe and normal argument positions can suitable tame MPO so that the induced *innermost runtime complexity* is polynomially bounded (see [2] for the polynomial path order POP*, a restriction of MPO). Here the runtime complexity of a TRS measures the maximal number of rewrite steps as a function in the size of the initial term, where the initial terms are *basic* terms, i.e., of the form $f(t_1,\dots,t_n)$ for constructor terms t_i. More precisely, define the *derivation height* of a terminating term t with respect to a finitely branching and well-founded relation $\to$ as

$$\mathrm{dh}(t,\to) = \max\{\ell \mid t \to t_1 \to \cdots \to t_\ell\} .$$

Let $\xrightarrow{\mathsf{i}}_{\mathcal{R}}$ denote the innermost rewrite relation as induced by a TRS $\mathcal{R}$. Then the *innermost runtime complexity* of a terminating TRS $\mathcal{R}$ is defined as

$$\mathrm{rc}^{\mathsf{i}}_{\mathcal{R}}(n) = \max\{\mathrm{dh}(t,\xrightarrow{\mathsf{i}}_{\mathcal{R}}) \mid t \text{ is basic and } |t| \leqslant n\} .$$

In this paper we present the *exponential path order* EPO*, a miniaturisation of the *lexicographic path order (LPO for short)*. We hope that the same idea exploited in [2] carries over to this miniaturisation, in the sense that by enforcing the scheme of safe nested recursion on notation the multiple recursive bound on derivation lengths induced by LPOs (see [8]) can be broken down to exponential bounds on innermost derivations, whenever the starting term is basic. Hence we aim for an order that

1. induces exponential bounds on the innermost runtime complexity of compatible TRSs, and

2. that is complete in the sense that every exponential time function is expressible by a TRS compatible with EPO*.

Up to now, we are able to verify the second property. The first property still needs further investigations.

[1]A function may even be defined by safe nested recursion on notation with more than two nested recursive calls. In fact, at least three layers are needed to capture the class FEXP, compare [1]. We avoid this complication to simplify the presentation.

2 Exponential Path Order EPO*

In this section we present the *exponential path order* EPO*. We fix a finite but else arbitrary signature $\mathcal{F}$, partitioned into defined symbols $\mathcal{D}$ and constructors $\mathcal{C}$. We use $\succsim = \succ \uplus \approx$ to denote an *admissible* quasi-precedence, i.e. a quasi-precedence where constructors are minimal. The separation of safe and normal argument positions is taken into account by the notion of *safe mapping*. A safe mapping safe is a function that associates with every n-ary function symbol f the set of *safe argument positions*. For constructors $f \in \mathcal{C}$ we require that all argument positions are safe. The argument positions not included in $\mathsf{safe}(f)$ are called *normal* and denoted by $\mathsf{nrm}(f)$. To simplify the presentation, we write $f(t_{i_1},\dots,t_{i_k};t_{j_1},\dots,t_{j_l})$ for the term $s = f(t_1,\dots,t_n)$ with $\mathsf{safe}(f) = \{i_1,\dots,i_k\}$ and $\mathsf{nrm}(f) = \{j_1,\dots,j_l\}$. We use $\approx_\mathsf{s}$ to denote term equivalence as induced by $\succsim$. Moreover, we suppose $\approx_\mathsf{s}$ respects the separation of argument positions, compare [3].

The *exponential path order* EPO* $>_{\mathsf{epo}\star}$ is based on an auxiliary order $\sqsupset_{\mathsf{epo}\star}$ defined as follows.

$$(1)\quad \frac{s_i \sqsupseteq_{\mathsf{epo}\star} t}{f(s_1,\dots,s_l;s_{l+1},\dots,s_m) \sqsupset_{\mathsf{epo}\star} t}\ f \in \mathcal{C} \text{ and some } 1 \leqslant i \leqslant m$$

$$(2)\quad \frac{s_i \sqsupseteq_{\mathsf{epo}\star} t}{f(s_1,\dots,s_l;s_{l+1},\dots,s_m) \sqsupset_{\mathsf{epo}\star} t}\ f \in \mathcal{D} \text{ and some } 1 \leqslant i \leqslant l$$

Here $\sqsupseteq_{\mathsf{epo}\star} := \sqsupset_{\mathsf{epo}\star} \cup \approx_\mathsf{s}$. The split into two orders is necessary, as we must carefully control composition and recursion of functions according to the schemes (SNC_2) and (SNRN). Note that due to the restrictive definition of case (2), one can show $f(\vec{x};\vec{y}) \sqsupset_{\mathsf{epo}\star} x_i$, but one cannot show $f(\vec{x};\vec{y}) \sqsupset_{\mathsf{epo}\star} y_i$. Based on the auxiliary order $\sqsupset_{\mathsf{epo}\star}$, we define for $s = f(s_1,\dots,s_l;s_{l+1},\dots,s_m)$ the *exponential path order* EPO* $>_{\mathsf{epo}\star}$ as follows.

$$(1)\quad \frac{s_i \geq_{\mathsf{epo}\star} t}{f(s_1,\dots,s_l;s_{l+1},\dots,s_m) >_{\mathsf{epo}\star} t}\ \text{for some } 1 \leqslant i \leqslant m$$

$$(2)\quad \frac{s \sqsupset_{\mathsf{epo}\star} t_1 \ \cdots\ s \sqsupset_{\mathsf{epo}\star} t_k \quad s >_{\mathsf{epo}\star} t_{k+1} \ \cdots\ s >_{\mathsf{epo}\star} t_n}{f(s_1,\dots,s_l;s_{l+1},\dots,s_m) >_{\mathsf{epo}\star} g(t_1,\dots,t_k;t_{k+1},\dots,t_n)}\ \text{for } f \succ g$$

$$(3)\quad \frac{s_1 = t_1 \ \cdots\ s_1 = t_{i-1} \quad s_i \sqsupset_{\mathsf{epo}\star} t_i \quad s \sqsupset_{\mathsf{epo}\star} t_{i+1} \ \cdots\ s \sqsupset_{\mathsf{epo}\star} t_l \quad s >_{\mathsf{epo}\star} t_{k+1} \ \cdots\ s >_{\mathsf{epo}\star} t_n}{f(s_1,\dots,s_l;s_{l+1},\dots,s_m) >_{\mathsf{epo}\star} g(t_1,\dots,t_k;t_{k+1},\dots,t_n)}$$

for $f \approx g$ and some $1 \leqslant i \leqslant \min(l,k)$

It is easy to see that $\sqsupset_{\mathsf{epo}\star} \subseteq >_{\mathsf{epo}\star} \subseteq >_{\mathsf{lpo}}$, hence compatibility of a TRS with $>_{\mathsf{epo}\star}$ implies termination, and moreover, a multiply recursive bound on the length of derivations [8]. Note that in order to show $s >_{\mathsf{epo}\star} g(t_1,\dots,t_k;t_{k+1},\dots,t_n)$ by case (2), we need to prove $s \sqsupset_{\mathsf{epo}\star} t_i$ for $1 \leqslant i \leqslant k$ instead of $s >_{\mathsf{epo}\star} t_i$. By the above observation on $>_{\mathsf{epo}\star}$, we can only compare normal arguments of s with t_i. This is in accordance with the scheme (SNC_2).

In [6], the second author introduces the *exponential path order* EPO, a restriction of LPO that induce exponential bounds on the innermost runtime complexity of TRSs. Although the order is complete for FEXP in principle, its application is very restricted on naturally formulated TRSs. The idea of the current research on EPO* is to lift this limitation. Besides the order EPO, a term rewriting characterisation $\mathcal{R}_\mathcal{N}$ of the class FEXP is presented [6]. This characterisation is inspired by the schemes from [1], we use it below to show completeness of EPO*. The scheme of rewrite rules $\mathcal{R}_\mathcal{N}$ consists of the rules drawn below. For clarification of this scheme of rewrite rules, we kindly refer the reader to [6]. Binary words are formed from the constructor symbols $\varepsilon, \mathrm{S}_0$ and S_1. The function symbols $\mathrm{O}^{k,l}, \mathrm{I}^{k,l}_r, \mathrm{P}, \mathrm{C}$ correspond to the initial functions of the class $\mathcal{N}$. The function symbols $\mathrm{SUB}[g,i_1,\dots,i_k,\vec{h}]$ are used to denote the function obtained by composing functions g and $\vec{h}$ according to the scheme (SNC_2). Finally, function symbols $\mathrm{SNRN}[g,h_{w'},\vec{t_{w'}},\vec{s_{w'}}\,(w' \in \Sigma_0^k)]$ correspond to the functions defined by safe nested recursion on notation in accordance to scheme (SNRN). We highlight the separation of safe and normal argument positions directly in the rules.

$$
\begin{array}{rll}
(1) & \mathrm{O}^{k,l}(\vec{x};\vec{y}) \to \epsilon & \\
(2) & \mathrm{I}_r^{k,l}(\vec{x};\vec{y}) \to x_r & \text{for } 1 \leqslant r \leqslant k \\
(3) & \mathrm{I}_r^{k,l}(\vec{x};\vec{y}) \to y_{r-k} & \text{for } k < r \leqslant l+k \\
(4) & \mathrm{P}(;\epsilon) \to \epsilon & \\
(5) & \mathrm{P}(;\mathrm{S}_i(;x)) \to x & \\
(6) & \mathrm{C}(;\epsilon,y_0,y_1) \to y_0 & \\
(7) & \mathrm{C}(;\mathrm{S}_i(;x),y_0,y_1) \to y_1 & \\
(8) & \mathrm{SUB}[g,i_1,\dots,i_k,\vec{h}](\vec{x};\vec{y}) \to g(x_{i_1},\dots,x_{i_k};\vec{h}(\vec{x};\vec{y})) & \\
(9) & \mathrm{SNRN}[g,h_{w'},\vec{t_{w'}},\vec{s_{w'}}\,(w' \in \Sigma_0^k)](\vec{\epsilon},\vec{x};\vec{y}) \to g(\vec{x};\vec{y}) & \\
(10) & \mathrm{SNRN}[g,h_{w'},\vec{t_{w'}},\vec{s_{w'}}\,(w' \in \Sigma_0^k)](\mathrm{S}_{i_1}(;z_1),\dots,\mathrm{S}_{i_k}(;z_k),\vec{x};\vec{y}) \to & \\
 & \quad h_w(\vec{v_1},\vec{x};\vec{y},\mathrm{SNRN}[g,h_{w'},\vec{t_{w'}},\vec{s_{w'}}\,(w' \in \Sigma_0^k)](\vec{v_1},\vec{x};\vec{a})) & \\
 & \quad\quad [\vec{t_w}(\vec{v_2},\vec{x};\vec{y},\mathrm{SNRN}[g,h_{w'},\vec{t_{w'}},\vec{s_{w'}}\,(w' \in \Sigma_0^k)](\vec{v_2},\vec{x};\vec{b}))/\vec{a}] & \\
 & \quad\quad [\vec{s_w}(\vec{v_3},\vec{x};\vec{y},\mathrm{SNRN}[g,h_{w'},\vec{t_{w'}},\vec{s_{w'}}\,(w' \in \Sigma_0^k)](\vec{v_3},\vec{x};\vec{y}))/\vec{b}] \quad i_j \in \{0,1\}, & \text{for } 1 \leqslant j \leqslant k
\end{array}
$$

In (10) we use $\vec{v_1}$ and $\vec{v_2}$ for specific predecessors of the arguments to $\mathrm{SNRN}[g,h_{w'},\vec{t_{w'}},\vec{s_{w'}}\,(w' \in \Sigma_0^k)]$, compare the scheme ($\mathsf{SNRN}$). By the results of [1], it follows that for each $f \in \mathsf{FEXP}$ there exists a finite restriction $\mathcal{R}_f \subseteq \mathcal{R}_{\mathcal{N}}$ such that $\mathcal{R}_f$ computes the function f (compare [6]).

Theorem 1. *Let $\mathcal{R}_f$ be a finite restriction of $\mathcal{R}_{\mathcal{N}}$. Then $\mathcal{R}_f \subseteq >_{\mathsf{epo}\star}$ for some instance of* EPO*.

Proof. Define $\mathrm{lh}(g)$ for symbol g appearing in $\mathcal{R}_f$ as follows. Set $\mathrm{lh}(g) = 1$ for $g \in \{\mathrm{O}^{k,l},\mathrm{I}_r^{k,l},\mathrm{P},\mathrm{C}\}$. Define

$$\mathrm{lh}(\mathrm{SUB}[g,i_1,\dots,i_k,\vec{h}]) = \max\{\mathrm{lh}(g),\mathrm{lh}(\vec{h})\} + 1$$

and

$$\mathrm{lh}(\mathrm{SNRN}[g,h_{w'},\vec{t_{w'}},\vec{s_{w'}}\,(w' \in \Sigma_0^k)]) = \max\{\mathrm{lh}(g),\mathrm{lh}(\vec{t_{w'}}),\mathrm{lh}(\vec{s_{w'}}) \mid w' \in \Sigma_0^k\} + 1 \ .$$

Define the safe mapping safe as indicated by the schemata $\mathcal{R}_{\mathcal{N}}$, and define $f > g$ in the precedence if $\mathrm{lh}(f) > \mathrm{lh}(g)$. Then it can be shown that $R_f \subseteq >_{\mathsf{epo}\star}$. We show the most interesting case, namely we orient the final rule. The general case easily follows from this. For notational reasons, we only consider two levels of nestings, that is we show

$$\mathsf{f}(\mathrm{S}_{i_1}(;z_1),\dots,\mathrm{S}_{i_k}(;z_k),\vec{x};\vec{y}) >_{\mathsf{epo}\star} h_w(\vec{v_1},\vec{x};\vec{y},\mathsf{f}(\vec{v_1},\vec{x};\vec{t_w}(\vec{v_2},\vec{x};\vec{y},\mathsf{f}(\vec{v_2},\vec{x};\vec{y}))))$$

where f abbreviate $\mathrm{SNRN}[g,h_{w'},\vec{t_{w'}},\vec{s_{w'}}\,(w' \in \Sigma_0^k)]$. Set $u := \mathsf{f}(\mathrm{S}_{i_1}(;z_1),\dots,\mathrm{S}_{i_k}(;z_k),\vec{x};\vec{y})$. By one application of rule (1) in the definition of $>_{\mathsf{epo}\star}$ we obtain $u >_{\mathsf{epo}\star} y_i$ for $y_i \in \vec{y}$, similar one application of rule (2) of $\sqsupset_{\mathsf{epo}\star}$ yields $u \sqsupset_{\mathsf{epo}\star} x_i$ for $x_i \in \vec{x}$. Recall that terms $\vec{v_3}$ encode $<_{\mathsf{lex}}$-predecessors of words corresponding to terms $\vec{z}$ (compare the remarks below scheme (SNRN)). From this observation we see that for $\vec{v_3} = v_1,\dots,v_k$ and some $1 \leqslant i \leqslant k$,

$$\mathrm{S}_{i_1}(z_1) = v_1,\dots,\mathrm{S}_{i_{i-1}}(z_{i-1}) = v_{i-1},\ \mathrm{S}_{i_i}(z_i) \sqsupset_{\mathsf{epo}\star} v_i \text{ and } \mathrm{S}_{i_{i+1}}(z_{i+1}) \sqsupseteq_{\mathsf{epo}\star} v_{i+1},\dots,\mathrm{S}_{i_{i+1}}(z_k) \sqsupseteq_{\mathsf{epo}\star} v_k \ .$$

Thus by one application of rule (3) of $>_{\mathsf{epo}\star}$ we are able to conclude $u >_{\mathsf{epo}\star} \mathsf{f}(\vec{v_2},\vec{x};\vec{y})$. Note that by the above inequalities, also $u \sqsupset_{\mathsf{epo}\star} v_i$ for $1 \leqslant i \leqslant k$, and thus due to rule (2) of $>_{\mathsf{epo}\star}$ we further obtain

$$u >_{\mathsf{epo}\star} \vec{t_w}(\vec{v_2},\vec{x};\vec{y},\mathsf{f}(\vec{v_2},\vec{x};\vec{y})) \ .$$

Carrying over the observations on $\vec{v_2}$ to $\vec{v_1}$, the latter inequality gives us

$$u >_{\mathsf{epo}\star} \mathsf{f}(\vec{v_1},\vec{x};\vec{t_w}(\vec{v_2},\vec{x};\vec{y},\mathsf{f}(\vec{v_2},\vec{x};\vec{y})))$$

by another application of rule (3). We conclude with a final application of rule (2). $\square$

We want to point out that compatibility with EPO* *does not* induce exponential runtime complexity. Consider the TRS $\mathcal{R}$ consisting of the rules

$$\mathsf{d}(;x) \to \mathsf{c}(;x,x) \qquad \mathsf{f}(0;y) \to y \qquad \mathsf{f}(\mathsf{s}(;x);y) \to \mathsf{f}(x;\mathsf{d}(;\mathsf{f}(x;y))) \ .$$

Then $\mathcal{R} \subseteq >_{\mathsf{epo}\star}$ for the precedence $\mathsf{f} > \mathsf{d}$ and safe mapping as indicated in the definition of $\mathcal{R}$. Still, we conjecture the following:

Conjecture 1. *Suppose $\mathcal{R}$ is a constructor TRS compatible with $>_{\mathsf{epo}\star}$. Then the innermost runtime complexity $\mathrm{rc}^{\mathsf{i}}_{\mathcal{R}}(n)$ is bounded by an exponential.*

Our belief in this conjecture is based on the proof of completeness of exponential path orders EPO as put forward in [6]. Completeness of the order is shown by embedding derivations of finite restrictions of $\mathcal{R}_f \subset \mathcal{R}_{\mathcal{N}}$ into EPO via suitable term interpretations. The definition of the employed term interpretation takes the separation of safe and normal argument positions into account. By the close correspondence between the scheme $\mathcal{R}_{\mathcal{N}}$ with the ordering constraints imposed by $>_{\mathsf{epo}\star}$, compatibility $\mathcal{R} \subseteq >_{\mathsf{epo}\star}$ should give enough information to embed $\mathcal{R}$ derivations into instances of EPO in a similar spirit. Whether this truly holds is subject to further research.

References

[1] Arai, T., Eguchi, N.: A new Function Algebra of EXPTIME Functions by Safe Nested Recursion. TCL 10(4) (2009)

[2] Avanzini, M., Moser, G.: Complexity Analysis by Rewriting. In: Proc. of 9th FLOPS 2008. LNCS, vol. 4989, pp. 130–146. Springer Verlag (2008)

[3] Avanzini, M., Moser, G.: Dependency Pairs and Polynomial Path Orders. In: Proc. of 20th RTA 2009. LNCS, vol. 5595, pp. 48–62. Springer Verlag (2009)

[4] Bellantoni, S., Cook, S.: A new Recursion-Theoretic Characterization of the Polytime Functions. CC 2(2), 97–110 (1992)

[5] Cobham, A.: The Intrinsic Computational Difficulty of Functions. In: Proc. 1964 LMPS 1964. pp. 24–30 (1964)

[6] Eguchi, N.: A Lexicographic Path Order with Slow Growing Derivation Bounds. MLQ 55(2), 212–224 (2009)

[7] Hofbauer, D.: Termination Proofs by Multiset Path Orderings Imply Primitive Recursive Derivation Lengths. TCS 105(1), 129–140 (1992)

[8] Weiermann, A.: Termination Proofs for Term Rewriting Systems with Lexicographic Path Ordering Imply Multiply Recursive Derivation Lengths. TCS 139, 355–362 (1995)

The Derivational Complexity of the Bits Function and the Derivation Gap Principle*

Harald Zankl and Martin Korp
Institute of Computer Science
University of Innsbruck
6020 Innsbruck, Austria
{harald.zankl,martin.korp}@uibk.ac.at

Abstract

In this note we present two proofs that the derivational complexity of the bits function is linear. The first proof is intuitive but not very suitable for implementation while the second one has been found automatically. Using the second proof idea allows the complexity tool CaT to show linear derivational complexity of the bits function for which no other current contemporary analyzer can infer a polynomial upper bound. In the second part of this note we generalize the weight gap principle of (Hirokawa and Moser, 2008).

1 Introduction

Hofbauer and Lautemann [5] consider the length of derivations as a measurement for the complexity of terminating rewrite systems. The resulting notion of *derivational complexity* relates the length of a rewrite sequence to the size of its starting term. Thereby it is, e.g., a suitable metric for the complexity of deciding the word problem for a given confluent and terminating rewrite system (since the decision procedure rewrites terms to normal form).

To show (feasible) upper bounds on the derivational complexity currently few techniques are known. Typically termination criteria are restricted such that polynomial upper bounds can be inferred. The early work by Hofbauer and Lautemann [5] considers polynomial interpretations, suitably restricted, to admit quadratic derivational complexity. Match-bounds [2] and arctic matrix interpretations (AMIs) [7, 8] induce linear derivational complexity and triangular matrix interpretations (TMIs) [9] admit polynomially long derivations (the dimension of the matrices yields the degree of the polynomial). All these methods share the property that until now they have been used directly only, meaning that a single termination technique has to orient all rules in one go. However, using direct criteria exclusively is problematic due to their restricted power.

In the sequel we consider the TRS $\mathcal{R}_{\mathsf{bits}}$ (nontermin/AG01/#4.28 from [10])

$$\begin{aligned} \mathsf{half}(\mathsf{0}) &\to \mathsf{0} & \mathsf{bits}(\mathsf{0}) &\to \mathsf{0} \\ \mathsf{half}(\mathsf{s}(\mathsf{0})) &\to \mathsf{0} & \mathsf{bits}(\mathsf{s}(x)) &\to \mathsf{s}(\mathsf{bits}(\mathsf{half}(\mathsf{s}(x)))) \\ \mathsf{half}(\mathsf{s}(\mathsf{s}(x))) &\to \mathsf{s}(\mathsf{half}(x)) \end{aligned}$$

and present two proofs that the derivational complexity of the TRS $\mathcal{R}_{\mathsf{bits}}$ is linear, i.e., a term of size n admits at most derivations of length $\mathcal{O}(n)$. This result is somehow surprising due to the last rule. (Note that already a single rule $\mathsf{f}(\mathsf{g}(x)) \to \mathsf{g}(\mathsf{f}(x))$ causes the derivational complexity to be quadratic.) The first proof is presented in Section 3 and is based on a low level reasoning exploiting the structure of the TRS $\mathcal{R}_{\mathsf{bits}}$. Although the reasoning is intuitive it is hard to automate. The second proof given in Section 4 builds on a recent result by the authors that allows to combine different complexity criteria for a single TRS [11]. This approach is very suitable for implementation and linear derivational complexity of the TRS $\mathcal{R}_{\mathsf{bits}}$ can be inferred completely automatically by our complexity analyzer CaT.[1] Note that

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

[1] http://cl-informatik.uibk.ac.at/software/cat

currently no other tool can show linear (not even polynomial) derivational complexity of the TRS $\mathcal{R}_{\mathsf{bits}}$.

After the presentation of the bits example we focus on the weight gap principle introduced in [4]. We show in Section 5 how it can be generalized such that in principle arbitrary techniques can be plugged in.

2 Preliminaries

We assume familiarity with rewriting [1]. A *relative* TRS $\mathcal{R}/\mathcal{S}$ consists of two TRSs $\mathcal{R}$ and $\mathcal{S}$ with the rewrite relation $\rightarrow_{\mathcal{R}/\mathcal{S}} = \rightarrow^*_{\mathcal{S}} \cdot \rightarrow_{\mathcal{R}} \cdot \rightarrow^*_{\mathcal{S}}$. The *derivation length* of a term t with respect to a relation $\rightarrow$ and the set of terms is defined as follows: $\mathsf{dl}(t, \rightarrow) = \max\{m \mid \exists u\; t \rightarrow^m u\}$. The *derivational complexity* computes the maximal derivation length of all terms up to a given size, i.e., $\mathsf{dc}(n, \rightarrow) = \max\{\mathsf{dl}(t, \rightarrow) \mid |t| \leqslant n\}$. Sometimes we say that $\mathcal{R}$ ($\mathcal{R}/\mathcal{S}$) has linear, quadratic, etc. derivational complexity if $\mathsf{dc}(n, \rightarrow_{\mathcal{R}})$ ($\mathsf{dc}(n, \rightarrow_{\mathcal{R}/\mathcal{S}})$) can be bounded by a linear, quadratic, etc. polynomial in n. TMIs of dimension one are called SLIs. For the remainder of this note we assume that TRSs are finite.

3 Hand-Made Proof

To simplify the presentation of the first proof of the linear derivational complexity of the TRS $\mathcal{R}_{\mathsf{bits}}$ we abbreviate $s^n(0)$ by n and drop parentheses if no confusion can arise. At first we show that each derivation induced by a term of the form $\mathsf{bits}\; n$ has at most length $3n$. To this end we need the following technical lemma.

Lemma 1. *Let* $t = \mathsf{bits}\; n$. *If* $n = 0$ *then* $t \rightarrow 0$. *If* $n > 0$ *then* $t \rightarrow^{2+\frac{n}{2}} \mathsf{s}\;\mathsf{bits}\;\frac{n}{2}$.

Proof. We have $t \rightarrow \mathsf{s}\;\mathsf{bits}\;\mathsf{half}\; n \rightarrow^{n/2} \mathsf{s}\;\mathsf{bits}\;\frac{n}{2}\mathsf{half}\; m \rightarrow \mathsf{s}\;\mathsf{bits}\;\frac{n}{2}$ where $m = 0$ or $m = 1$. □

Lemma 2. *Let* $t = \mathsf{bits}\; n$. *Then* $\mathsf{dl}(t, \rightarrow_{\mathcal{R}}) \leqslant 3n$.

Proof. We restrict our attention to a special reduction. Since $\mathcal{R}$ is terminating (see [3, Example 5]) and has the diamond property this is fine. Using Lemma 1 we obtain the finite rewrite sequence

$$t \rightarrow^{2+\frac{n}{2}} \mathsf{s}\;\mathsf{bits}\;\frac{n}{2} \rightarrow^{2+\frac{n}{4}} \mathsf{s}\;\mathsf{s}\;\mathsf{bits}\;\frac{n}{4} \rightarrow^{2+\frac{n}{8}} \cdots \rightarrow^{2+\frac{n}{2^k}} \mathsf{s}^k\mathsf{bits}\;\frac{n}{2^k}$$

where $k = \lceil \log(n) \rceil$. Since $k \leqslant n$ we have

$$\sum_{1 \leqslant i \leqslant k} (2 + \frac{n}{2^i}) \leqslant 2n + n \cdot \sum_{1 \leqslant i \leqslant n} \frac{1}{2^i} \leqslant 3n$$

and hence $\mathsf{dl}(t, \rightarrow_{\mathcal{R}}) \leqslant 3n$ as desired □

At next we prove that each term of the form $\mathsf{bits}^m n$ admits at most linear derivation length. To prove this claim we need the property that $\log^i(n) \leqslant \frac{n}{2^i}$ if $\log^i(n) > 0$ for all $n > 3$ and $i \in \mathbb{N}$. This is show below.

Lemma 3. *For any* $n > 3$ *and* $i \in \mathbb{N}$ *with* $\log^i(n) > 0$ *we have* $\log^i(n) \leqslant \frac{n}{2^i}$.

Proof. We perform induction on i. If $i = 0$ then $n \leqslant \frac{n}{2^0} = n$. For the step case we claim that $\log(n) \leqslant \frac{n}{2}$ for all $n > 3$. Using the claim as well as monotonicity of log (on $n > 0$) we obtain:

$$\log^{i+1}(n) = \log(\log^i(n)) \leqslant \log(\frac{n}{2^i}) \leqslant \frac{n}{2^{i+1}}$$

It remains to show that the claim is correct. We have $\log(n) \leqslant \frac{n}{2} \Leftrightarrow 2\log(n) \leqslant n \Leftrightarrow 2^{2\log(n)} \leqslant 2^n \Leftrightarrow 2^{\log(n)}2^{\log(n)} \leqslant 2^n \Leftrightarrow n \cdot n \leqslant 2^n \Leftrightarrow n^2 \leqslant 2^n$. The latter can be shown by an easy induction on n. □

Lemma 4. *Let $n > 3$ and $t = \mathsf{bits}^m\, n$. Then $\mathsf{dl}(t, \to_{\mathcal{R}}) \leqslant 6n + m$.*

Proof. First we assume that n is large enough such that all $\log^i(n)$ are positive. By Lemma 2 we can then estimate a sequence as follows:

$$t \to^{\leqslant 3n} \mathsf{s}\,\mathsf{bits}^{m-1} \log(n) \to^{\leqslant 3\log(n)} \mathsf{s}\,\mathsf{s}\,\mathsf{bits}^{m-2} \log^2(n) \to^{\leqslant 3\log^2(n)} \cdots \to^{\leqslant 3\log^{m-1}(n)} \mathsf{s}^{\,m} \log^m(n)$$

Using Lemma 3 we conclude that this sequence admits at most

$$3n + \sum_{1 \leqslant i \leqslant m} 3\log^i(n) \leqslant 3n + \sum_{1 \leqslant i \leqslant m} 3\frac{n}{2^i} \leqslant 3n + 3n \sum_{1 \leqslant i \leqslant m} \frac{1}{2^i} \leqslant 6n$$

steps. In the other case there is a $k \leqslant m$ such that $t \to^{\leqslant 6n} \mathsf{s}^k\, \mathsf{bits}^{m-k}\, 0 \to^{m-k} \mathsf{s}^k\, 0$. Putting things together finishes the proof. □

Since $\mathsf{bits}\ 4$ admits longer derivations as $\mathsf{bits}\ 3$, by Lemma 4 we obtain the following corollary.

Corollary 5. *Let $t = \mathsf{bits}^m\, n$. Then $\mathsf{dl}(t, \to_{\mathcal{R}}) \leqslant 6n + m + 24$.* □

Since the terms considered in Corollary 5 have longest derivations we obtain the next result.

Corollary 6. *The derivational complexity of the TRS $\mathcal{R}$ is linear.* □

4 Automated Proof

Contemporary methods for proving the derivational complexity of TRSs (automatically) comprise match-bounds [2], AMIs [7,8] and TMIs [9]. These criteria can be preceded by (complexity-preserving) transformations. Usually one method must orient all rewrite rules strictly to deduce an upper bound on the derivational complexity of a TRS. Our experiments revealed that none of the above methods can conclude polynomial derivational complexity of the TRS $\mathcal{R}_{\mathsf{bits}}$ on its own. Recently [11] introduced a modular approach that allows to combine different criteria for complexity analysis by relative rewriting. Suddenly a polynomial (even linear!) upper bound on the derivational complexity can be inferred automatically. We recall the main results that allow to handle the TRS $\mathcal{R}_{\mathsf{bits}}$. The first theorem presented is one of the main results from [11].

Theorem 7. *Let $(\mathcal{R}_1 \cup \mathcal{R}_2)/\mathcal{S}$ be a relative TRS and let t be a term such that t terminates with respect to $(\mathcal{R}_1 \cup \mathcal{R}_2)/\mathcal{S}$. Then $\mathcal{O}(\mathsf{dl}(t, \to_{(\mathcal{R}_1 \cup \mathcal{R}_2)/\mathcal{S}})) = \max\{\mathcal{O}(\mathsf{dl}(t, \to_{\mathcal{R}_1/\mathcal{S}_1})), \mathcal{O}(\mathsf{dl}(t, \to_{\mathcal{R}_2/\mathcal{S}_2}))\}$.* □

The next theorem generalizes the *weight gap principle* introduced in [4] to relative rewriting.

Theorem 8. *Let $(\mathcal{R}_1 \cup \mathcal{R}_2)/\mathcal{S}$ be a relative TRS, $\mathcal{R}_1$ be non-duplicating, and let $\mathcal{M}$ be an SLI such that $\mathcal{R}_2 \subseteq \succ_{\mathcal{M}}$ and $\mathcal{S} \subseteq \succeq_{\mathcal{M}}$. Then for any $\mathcal{R}_1$ and $\mathcal{M}$ there exist constants K and L such that*

$$K \cdot \mathsf{dl}(t, \to_{\mathcal{R}_1/(\mathcal{R}_2 \cup \mathcal{S})}) + L \cdot |t| \geqslant \mathsf{dl}(t, \to_{(\mathcal{R}_1 \cup \mathcal{R}_2)/\mathcal{S}})$$

whenever t is terminating on $(\mathcal{R}_1 \cup \mathcal{R}_2)/\mathcal{S}$. □

Using the above theorems it is easy to show that the TRS $\mathcal{R}_{\mathsf{bits}}$ admits linear derivational complexity.

Theorem 9. *The derivational complexity of the TRS $\mathcal{R}_{\mathsf{bits}}$ is at most linear.*

Proof. Obviously $\mathsf{dl}(t, \to_{\mathcal{R}})$ is equal to $\mathsf{dl}(t, \to_{\mathcal{R}/\varnothing})$. By Theorem 8 with an SLI that counts the symbols $\mathsf{bits}, \mathsf{half}, \mathsf{s}$ further progress is achieved and the derivational complexity of $\mathcal{R}_{\mathsf{bits}}/\varnothing$ can be estimated by analyzing the complexity of the rule $\mathsf{bits}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{bits}(\mathsf{half}(\mathsf{s}(x))))$ relative to the other rules. For the last step there is an arctic interpretation of dimension three, i.e.,

$$\mathsf{bits}_{\mathcal{A}}(\vec{x}) = \begin{pmatrix} 0\,0\,0 \\ 0\,0\,3 \\ \$\,0\,0 \end{pmatrix} \vec{x} \qquad \mathsf{half}_{\mathcal{A}}(\vec{x}) = \begin{pmatrix} 0\,\$\,\$ \\ 0\,\$\,\$ \\ 0\,\$\,\$ \end{pmatrix} \vec{x} \qquad \mathsf{s}_{\mathcal{A}}(x) = \begin{pmatrix} 0\,\$\,0 \\ 0\,1\,2 \\ 2\,\$\,0 \end{pmatrix} x \qquad \mathsf{0}_{\mathcal{A}} = \begin{pmatrix} 0 \\ \$ \\ \$ \end{pmatrix}$$

that orients the problematic rule strictly and the remaining rules weakly. Hence the derivational complexity of the TRS $\mathcal{R}_{\mathsf{bits}}$ is at most linear. □

Note that the proof of Theorem 9 has been constructed automatically by $\mathsf{C\!T}$ within a few seconds. Furthermore we stress that the involved reasoning is on a higher level compared to the handmade proof. This also eases the task of (automated) certification.

5 Derivation Gap Principle

Based on the success of the modular complexity approach using relative rewriting, in this section we aim to generalize the weight gap principle from [4] into the so-called *derivation gap principle*. Provided that $\mathcal{R}$ satisfies some property, the new result allows us to establish an upper bound on the derivational complexity of $\mathcal{R} \cup \mathcal{S}$ based on the derivational complexities of $\mathcal{R}/\mathcal{S}$ and $\mathcal{S}$.

Theorem 10. *Let $\mathcal{R} \cup \mathcal{S}$ be a TRS and t be terminating on $\mathcal{R} \cup \mathcal{S}$. If there exists a constant Δ such that for any $t \to_{\mathcal{R}} s$ we have* $\mathsf{dl}(t, \to_{\mathcal{S}}) + \Delta \geqslant \mathsf{dl}(s, \to_{\mathcal{S}})$ *then* $\mathsf{dc}(n, \to_{\mathcal{R}\cup\mathcal{S}}) \in \mathcal{O}(\mathsf{dc}(n, \to_{\mathcal{R}/\mathcal{S}}) + \mathsf{dc}(n, \to_{\mathcal{S}}))$.

Proof. We show that under the above assumptions there exists a constant K such that $\mathsf{dl}(s, \to_{\mathcal{R}\cup\mathcal{S}}) \leqslant K \cdot \mathsf{dl}(s, \to_{\mathcal{R}/\mathcal{S}}) + \mathsf{dl}(s, \to_{\mathcal{S}})$. Consider a maximal derivation in $\mathcal{R} \cup \mathcal{S}$, written as follows:

$$s = s_0 \to_{\mathcal{S}}^{k_0} t_0 \to_{\mathcal{R}} s_1 \to_{\mathcal{S}}^{k_1} t_1 \to_{\mathcal{R}} \cdots \to_{\mathcal{R}} s_m \to_{\mathcal{S}}^{k_m} t_m$$

Since the derivation is maximal we have $\mathsf{dl}(s_0, \to_{\mathcal{R}\cup\mathcal{S}}) \leqslant \mathsf{dl}(s_0, \to_{\mathcal{R}/\mathcal{S}}) + \sum_{0 \leqslant i \leqslant m} k_i$. Obviously we have $\mathsf{dl}(s_0, \to_{\mathcal{S}}) \geqslant \mathsf{dl}(t_0, \to_{\mathcal{S}}) + k_0$. Because $\mathsf{dl}(t_0, \to_{\mathcal{S}}) \geqslant \mathsf{dl}(s_1, \to_{\mathcal{S}}) - \Delta$ by assumption we obtain $\mathsf{dl}(s_0, \to_{\mathcal{S}}) \geqslant \mathsf{dl}(s_1, \to_{\mathcal{S}}) - \Delta + k_0$. An easy induction proof shows $\mathsf{dl}(s_0, \to_{\mathcal{S}}) + m \cdot \Delta \geqslant \sum_{0 \leqslant i \leqslant m} k_i$. Because $m \leqslant \mathsf{dl}(s_0, \to_{\mathcal{R}/\mathcal{S}})$ we have $\mathsf{dl}(s_0, \to_{\mathcal{R}\cup\mathcal{S}}) \leqslant \mathsf{dl}(s_0, \to_{\mathcal{R}/\mathcal{S}}) + \mathsf{dl}(s_0, \to_{\mathcal{S}}) + \Delta \cdot \mathsf{dl}(s_0, \to_{\mathcal{R}/\mathcal{S}})$ which can be simplified to $\mathsf{dl}(s_0, \to_{\mathcal{R}\cup\mathcal{S}}) \leqslant (\Delta + 1) \cdot \mathsf{dl}(s_0, \to_{\mathcal{R}/\mathcal{S}}) + \mathsf{dl}(s_0, \to_{\mathcal{S}})$. Taking $K = \Delta + 1$ concludes the proof. □

To implement the above theorem the question arises how to check that $s \to_{\mathcal{R}} t$ implies the desired $\mathsf{dl}(s, \to_{\mathcal{S}}) + \Delta \geqslant \mathsf{dl}(t, \to_{\mathcal{S}})$ for some constant Δ. Here the idea is to test $\mathsf{dl}(l, \to_{\mathcal{S}}) + \Delta \geqslant \mathsf{dl}(r, \to_{\mathcal{S}})$ for any $l \to r \in \mathcal{R}$ and demand that additionally $C[l] + \Delta \geqslant C[r]$ and $l\sigma + \Delta \geqslant r\sigma$ holds for all contexts C and substitutions σ. Note that $\mathsf{dl}(l, \to_{\mathcal{S}})$ can always be under-approximated by 0 while $\mathsf{dl}(r, \to_{\mathcal{S}})$ can be over-approximated, e.g., by some interpretation of r.

As we know from [4] SLIs can be used to get a concrete instance of Theorem 10. Below we give counterexamples that TMIs, AMIs, and match-bounds do not adhere to the derivation gap principle without further ado. As future work we plan to restrict these criteria accordingly such that they become applicable for Theorem 10. The TRS $\mathcal{R} \cup \mathcal{S}$ in the next example was proposed by Hofbauer [6].

Example 11. Consider the following two TRSs

$$\mathcal{R} = \{\mathsf{c}(\mathsf{L}(x)) \to \mathsf{R}(x)\} \qquad \mathcal{S} = \{\mathsf{R}(\mathsf{a}(x)) \to \mathsf{b}(\mathsf{b}(\mathsf{R}(x))), \mathsf{R}(x) \to \mathsf{L}(x), \mathsf{b}(\mathsf{L}(x)) \to \mathsf{L}(\mathsf{a}(x))\}$$

We observe that the derivational complexity of $\mathcal{R} \cup \mathcal{S}$ is exponential because $\mathsf{c}^n(\mathsf{L}(\mathsf{a}(x))) \to^* \mathsf{L}(\mathsf{a}^{2^n}(x))$.

Furthermore the derivational complexity of $\mathcal{R}/\mathcal{S}$ is linear (count c's). Since the TMI $\mathcal{M}_T$ with

$$\mathsf{a}_{\mathcal{M}}(\vec{x}) = \begin{pmatrix}1&0\\0&1\end{pmatrix}\vec{x} + \begin{pmatrix}0\\1\end{pmatrix} \quad \mathsf{b}_{\mathcal{M}}(\vec{x}) = \begin{pmatrix}1&0\\0&0\end{pmatrix}\vec{x} + \begin{pmatrix}1\\0\end{pmatrix} \quad \mathsf{R}_{\mathcal{M}}(\vec{x}) = \begin{pmatrix}1&3\\0&0\end{pmatrix}\vec{x} + \begin{pmatrix}2\\0\end{pmatrix} \quad \mathsf{L}_{\mathcal{M}}(\vec{x}) = \begin{pmatrix}1&0\\0&0\end{pmatrix}\vec{x}$$

orients all rules in $\mathcal{S}$ strictly—and hence gives a quadratic upper bound on $\mathsf{dc}(n, \to_{\mathcal{R}\cup\mathcal{S}})$—(in general) TMIs cannot adhere to Theorem 10. The problem is that although there exists a Δ with $\mathsf{dl}(l, \to_{\mathcal{S}}) + \Delta \geqslant \mathsf{dl}(r, \to_{\mathcal{S}})$ for all $l \to r \in \mathcal{R}$ this property is not closed under contexts; if arbitrary TMIs are considered. Similarly the AMI $\mathcal{M}_A$ (inducing at most linear derivational complexity of $\mathcal{S}$) with

$$\mathsf{a}_{\mathcal{M}}(\vec{x}) = \begin{pmatrix}0&\$\\3&3\end{pmatrix}\vec{x} \qquad \mathsf{b}_{\mathcal{M}}(\vec{x}) = \begin{pmatrix}1&2\\\$&0\end{pmatrix}\vec{x} \qquad \mathsf{R}_{\mathcal{M}}(\vec{x}) = \begin{pmatrix}1&3\\0&2\end{pmatrix}\vec{x} \qquad \mathsf{L}_{\mathcal{M}}(\vec{x}) = \begin{pmatrix}0&\$\\\$&\$\end{pmatrix}\vec{x}$$

violates the same requirement in Theorem 10 as $\mathcal{M}_T$ above. A similar reasoning holds for match-bounds; one easily verifies that the TRS $\mathcal{S}$ is match-bounded by 2.

Summary and Conclusion

In this note we gave two proofs that the derivational complexity of the TRS $\mathcal{R}_{\mathsf{bits}}$ is linear. The first proof gives much insight into the reductions this system admits but the argument is hard for automation. The second proof has been generated automatically, is formal but does not give much insight into the system. The second part of the note generalized the weight gap principle of Hirokawa and Moser [4]. As future work we plan to investigate restrictions of TMIs, AMIs and match-bounds such that they adhere to Theorem 10.

References

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

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

[3] Hirokawa, N., Middeldorp, A.: Polynomial interpretations with negative coefficients. In: AISC 2004. LNCS (LNAI), vol. 3249, pp. 185–198 (2004)

[4] Hirokawa, N., Moser, G.: Automated complexity analysis based on the dependency pair method. In: IJCAR 2008. LNCS, vol. 5195, pp. 364–379 (2008)

[5] Hofbauer, D., Lautemann, C.: Termination proofs and the length of derivations (preliminary version). In: RTA 1989. LNCS, vol. 355, pp. 167–177 (1989)

[6] Hofbauer, D., Waldmann, J.: Complexity bounds from relative termination proofs. Talk at the Workshop on Proof Theory and Rewriting, Obergurgl (2006). Available from `http://www.imn.htwk-leipzig.de/~waldmann/talk/06/rpt/rel/main.pdf`

[7] Koprowski, A., Waldmann, J.: Arctic termination ... below zero. In: RTA 2008. LNCS, vol. 5117, pp. 202–216 (2008)

[8] Koprowski, A., Waldmann, J.: Max/plus tree automata for termination of term rewriting. AC 19(2), 357–392 (2009)

[9] Moser, G., Schnabl, A., Waldmann, J.: Complexity analysis of term rewriting based on matrix and context dependent interpretations. In: FSTTCS 2008. LIPIcs, vol. 2, pp. 304–315 (2008)

[10] People, V.: Termination problem data base (2010). See `http://termination-portal.org/`

[11] Zankl, H., Korp, M.: Modular complexity analysis via relative complexity. In: RTA 2010. LIPIcs. (2010). To appear.

Lazy abstraction for size-change termination[*]

M. Codish
Dept. of Computer Science
Ben-Gurion University
Beer Sheva, Israel

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

P. Schneider-Kamp
IMADA
University of Southern Denmark
Odense, Denmark

Size-change termination [15] is a widely used means [2, 5, 14, 16, 17, 19, 20] of showing termination of programs after an abstract interpretation to size-change graphs has been applied.

While the use of program abstractions in termination analysis is widely considered indispensable for reasons of performance, it inevitably also comes with a price: Loss of precision. That means that although the concrete program is actually terminating, its abstraction (which over-approximates the behavior of the program) can be non-terminating, or at least its termination can be too hard to show automatically.

Even worse, there is a plethora of possible abstractions, of which we decide on a single one before attempting the actual termination proof. While in some cases, such a "one size fits all" choice of abstraction actually suffices for a successful termination proof, e.g., using size-change termination, this approach often is far too coarse. It would be significantly better if one had an oracle that automatically selected an abstraction that is tailor-made for the currently used proof method in order to succeed on the program in question.

Such an approach is actually rather common for automated termination proofs of term rewrite systems using the modular DP framework [1, 9, 11, 12]. Here for each proof step, a different abstraction can be chosen, e.g., polynomial interpretations [8], matrix interpretations [7, 13], Knuth-Bendix-orders [21], or recursive path orders [4, 6, 18]. In the last years, the automation of the search for suitable abstractions has made huge steps forwards by solving the arising search problems via carefully crafted reductions to SAT. This is a feasible approach since in practice these problems are all in the complexity class NP.

In size-change termination, however, an additional challenge is that the question whether a program is size-change terminating even for a *given* abstraction is already PSPACE-complete. Hence, one cannot expect the combined problem of *finding* a suitable abstraction that allows to conclude size-change termination to be any less difficult. However, in recent work [3], Ben-Amram and Codish present a method of automating an NP-complete fragment (called *SCNP*) of this PSPACE-complete termination criterion efficiently using SAT solving, which suffices to capture most practically interesting problem instances.

By coupling these ideas, we can combine the search for an abstraction and for the size-change termination argument by encoding the two problems into a single SAT instance. In this way, the problem of choosing the right abstraction for concluding size-change termination is solved *en passant* by the SAT solver.

We show that for the setting of term rewriting, the integration of this approach into the DP framework works very smoothly. Indeed, the combination of size-change termination (resp. SCNP) and a SAT encoding for a class of abstractions gives rise to a new class of reduction pairs which we call *size-change reduction pairs*. A major benefit of this is that we can now use size-change termination as a modular ingredient of the reduction pair processor, which is the main proof engine of the DP framework and has been refined numerous times [1, 9, 11, 12] over the last years.

We have implemented size-change reduction pairs in the termination prover AProVE [10]. Our experiments indicate that using size-change reduction pairs leads to significant benefits over the earlier attempt of porting size-change termination to the realm of dependency pairs for term rewriting [20].

This is not surprising since [20] only supported considering different abstractions via two-stage generate-and-test approaches, which tend to be hopelessly inefficient in many cases. Moreover, [20] also required that size-change termination be used as the final step even when used in the modular DP framework, i.e., a modular deletion of only some dependency pairs was not supported in this setting. This restriction is also present in the *certification problem format (CPF)*, an XML format used to represent

[*]Supported by the G.I.F. grant 966-116.6, the DFG grant GI 274/5-2, and the Danish Natural Science Research Council.

termination proofs in a machine-readable way such that they can be certified via a dedicated theorem prover. To date, also here size-change termination is only supported as a final step in an otherwise modular termination proof and it cannot be employed as an intermediate step to allow for other methods afterwards. With the contribution of size-change reduction pairs, these two major disadvantages of this earlier adaption of size-change termination to term rewriting have now been overcome.

Our contribution is not restricted to term rewriting, though: Also for other settings like termination proofs for logic programs or for imperative programs, current advances in SAT and SMT solving now allow for applying abstractions only when and where they are really needed, as opposed to premature abstraction, which is still customary in many approaches.

References

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

[2] J. Avery. Size-change termination and bound analysis. In *Proc. FLOPS '06*, LNCS 3945, pages 192–207, 2006.

[3] A. M. Ben-Amram and M. Codish. A SAT-based approach to size change termination with global ranking functions. In *Proc. TACAS '08*, LNCS 4963, pages 218–232, 2008.

[4] 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.

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

[6] M. Codish, V. Lagoon, and P. Stuckey. Solving partial order constraints for LPO termination. *Journal on Satisfiability, Boolean Modeling and Computation*, 5:193–215, 2008.

[7] J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. *Journal of Automated Reasoning*, 40(2-3):195–220, 2008.

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

[9] J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. LPAR '04*, LNAI 3542, pages 301–331, 2005.

[10] 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.

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

[12] N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1,2):172–199, 2005.

[13] D. Hofbauer and J. Waldmann. Termination of string rewriting with matrix interpretations. In *Proc. RTA '06*, LNCS 4098, pages 328–342, 2006.

[14] N. D. Jones and N. Bohr. Termination analysis of the untyped lambda calculus. In *Proc. RTA '04*, LNCS 3091, pages 1–23, 2004.

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

[16] N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. TermiLog: A system for checking termination of queries to logic programs. In *Proc. CAV '97*, LNCS 1254, pages 444–447, 1997.

[17] P. Manolios and D. Vroon. Termination analysis with calling context graphs. In *Proc. CAV '06*, LNCS 4144, pages 401–414, 2006.

[18] P. Schneider-Kamp, R. Thiemann, E. Annov, M. Codish, and J. Giesl. Proving termination using recursive path orders and SAT solving. In *Proc. FroCoS '07*, LNAI 4720, pages 267–282, 2007.

[19] D. Sereni and N. D. Jones. Termination analysis of higher-order functional programs. In *Proc. APLAS ’05*, LNCS 3780, pages 281–297, 2005.

[20] R. Thiemann and J. Giesl. The size-change principle and dependency pairs for termination of term rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 16(4):229–270, 2005.

[21] H. Zankl, N. Hirokawa, and A. Middeldorp. KBO orientability. *Journal of Automated Reasoning*, 43(2):173–201, 2009.

Dependency Graphs, Relative Rule Removal, the Subterm Criterion and Derivational Complexity*

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

Andreas Schnabl
University of Innsbruck
Innsbruck, Austria
andreas.schnabl@uibk.ac.at

Abstract

We study the derivational complexity induced by the dependency pair method, enhanced with the dependency graph refinement and the subterm criterion, allowing relative removal of rules, for base orders inducing linear derivational complexity. If relative rule removal is allowed, we get a multiply recursive upper bound, otherwise the bound is primitive recursive.

1 Introduction

We assume familiarity with term rewriting (see [3, 2, 16]) and the dependency pair method (cf. [1, 6, 7, 8, 17]). Let R be a finite TRS over the set of terms $\mathrm{T}(F,V)$ built over the signature F and variables V. Let $D \subseteq F$ denote the set of defined symbols. We use $\mathsf{DP}(R)$ and $\mathsf{DG}(R)$ to denote the set of dependency pairs of R and the dependency graph of R, respectively. We use DP problems as defined in [6, 17], and we write (P,R) to abbreviate DP problems of the shape $(P,\emptyset,R,\mathbf{m})$ (in the notation of [6, 17]).

In this extended abstract, we extend our recent results from [13, 14]. We investigate the derivational complexity of TRSs whose termination is shown by a restricted set of proof methods. Recall the definition of the derivation height of a terminating term t with respect to a finitely branching rewrite relation $\to$, and the derivational complexity function of a terminating TRS R:

$$\mathsf{dh}(t,\to) = \max\{n \mid \exists t\; s \to^n t\} \qquad \mathsf{dc}_R(n) = \max\{\mathsf{dh}(t,\to_R) \mid |t| \leqslant n\}$$

We show upper bounds on the derivational complexity of TRSs whose termination is proved by the following, well-known results:

Proposition 1 ([1]). *R is terminating iff for each strongly connected component (SCC) P of $\mathsf{DG}(R)$, the DP problem (P,R) is finite.*

Proposition 2 ([6]). *A DP problem (P,R) is finite iff there exist $P' \subset P$ and a reduction pair $(\gtrsim,\succ)$ such that $P \setminus P' \subseteq \succ$, $P' \cup R \subseteq \gtrsim$, and the DP problem (P',R) is finite.*

Proposition 3 ([8, 17]). *A DP problem (P,R) is finite if there exist $P' \subset P$ and a simple projection π such that $\pi(l) \rhd \pi(r)$ for each dependency pair $l \to r$ in $P \setminus P'$, $\pi(l) \unrhd \pi(r)$ for each dependency pair $l \to r$ in P', and the DP problem (P',R) is finite.*

To motivate this study, we give two examples that provide a lower-bound on the derivational complexity induced by Propositions 1 and 2, or Propositions 1 and 3.

Example 4. Consider the following TRS R, taken from [11]:

$$\mathsf{i}(x) \circ (y \circ z) \to x \circ (\mathsf{i}(\mathsf{i}(y)) \circ z)$$
$$\mathsf{i}(x) \circ (y \circ (z \circ w)) \to x \circ (z \circ (y \circ w))$$

It is shown in [11] that dc_R is not primitive recursive. As already stated in [4], termination can be easily proved as follows: first, we consider the reduction pair induced by the polynomial algebra A with $\circ^{\sharp}_A(x,y) = y$, $\circ_A(x,y) = y+1$ and $\mathsf{i}_A(x) = 0$, which allows us to remove three of the five dependency pairs of R by Proposition 2. Next, we apply the reduction pair induced by the polynomial algebra B with $\circ^{\sharp}_B(x,y) = x$, $\circ_B(x,y) = 0$, and $\mathsf{i}_B(x) = x+1$ and conclude termination of R.

*Partly supported by FWF (Austrian Science Fund) project P20133-N15.

We write $\succ/\gtrsim$ to abbreviate the relation $\gtrsim^* \cdot \succ \cdot \gtrsim^*$ (compare [5]). Note that the reduction pairs employed in Example 4 induce linear derivational complexity, that is, the function $\mathsf{dh}(t^\sharp, \succ/\gtrsim)$ grows only linar in $|t|$.

Example 5. Consider the following family of TRSs, denoted as $R(k)$, parametrised by k ($k \geqslant 2$), which encode the k-ary Ackermann function Ack_k.

$$\begin{aligned}
\mathsf{Ack}_k(\mathsf{0},\dots,\mathsf{0},n) &\to \mathsf{S}(n)\\
\mathsf{Ack}_k(l_1,\dots,l_{k-2},\mathsf{S}(m),\mathsf{0}) &\to \mathsf{Ack}_k(l_1,\dots,l_{k-2},m,\mathsf{S}(\mathsf{0}))\\
\mathsf{Ack}_k(l_1,\dots,l_{k-2},\mathsf{S}(m),\mathsf{S}(n)) &\to \mathsf{Ack}_k(l_1,\dots,l_{k-2},m,\mathsf{Ack}_k(l_1,\dots,l_{k-2},\mathsf{S}(m),n))\\
\mathsf{Ack}_k(l_1,\dots,l_{i-1},\mathsf{S}(l_i),\mathsf{0},\dots,\mathsf{0},n) &\to \mathsf{Ack}_k(l_1,\dots,l_i,n,\mathsf{0},\dots,\mathsf{0},n)
\end{aligned}$$

Termination of $R(k)$ can be shown by k applications of Proposition 3 with $\pi_i(\mathsf{Ack}_k^\sharp) = i$, where π_i is the simple projection used in the ith application of Proposition 3.

These examples show that Propositions 1, 2 and 3 admit much higher derivational complexities than the basic dependency pair method, which (as shown in [13]) only admits a triple exponential upper bound (for reduction pairs which induce linear derivational complexity).

Still, we can show that the derivational complexity of TRSs whose termination is proved using these propositions is bounded by a multiply recursive function. In the next section, we give a sketch of this proof for the case that only Propositions 1 and 2 are applied, where we assume that the reduction pairs employed induce at most multiple-recursive complexity. Note, however that our approach can be extended to termination proofs which additionally employ Proposition 3.

2 Proof of the Upper Bound

We start by ordering the (trivial and nontrivial) SCCs of $\mathsf{DG}(R)$ by assigning a *rank* to each of them. Let P, Q denote two SCCs. We call Q *reachable* from P if there exist nodes $u \in P$, $v \in Q$ and a path in $\mathsf{DG}(R)$ from u to v. For the remainder of this paper, let k be the number of SCCs in $\mathsf{DG}(R)$, A the maximum arity of any function symbol occurring in R, and $C := \max\{2\} \cup \{\mathsf{dp}(r) \mid l \to r \in R\}$. Consider the set of all bijective mappings from the set of SCCs of $\mathsf{DG}(R)$ to $\{1,\dots,k\}$ with the property that $\mathsf{rk}(P) > \mathsf{rk}(Q)$ whenever Q is reachable from P in $\mathsf{DG}(R)$. We fix an arbitrary one of these mappings to be rk. We call $\mathsf{rk}(P)$ the *rank* of an SCC P. The rank of a position p in a term u such that $(u|_p)^\sharp \notin \mathsf{NF}(P/R)$ for some SCC P is defined by $\mathsf{rk}(u,p) := \max\{\mathsf{rk}(s \to t) \mid \exists \sigma\ (u|_p)^\sharp \to_R^* s\sigma\}$, where the rank of a dependency pair $s \to t$, denoted by $\mathsf{rk}(s \to t)$, is the rank of P such that $s \to t \in P$.

In the following we write P_i for the unique SCC with rank i. Let ℓ be a natural number (the maximum number of applications of Proposition 2 for any SCC), and split each SCC P_i of $\mathsf{DG}(R)$ into ℓ (possibly empty) disjoint parts $P_{i,1},\dots,P_{i,\ell}$. The intuition behind this split is that $P_{i,j}$ contains the dependency pairs removed by the jth application of Proposition 2 within SCC P_i. Finally let g be a monotone function over $\mathbb{N}$ with $g(0) > 0$ and

$$\mathsf{dh}(t^\sharp, \to_{P_{i,j}/R \cup \bigcup_{j'=j+1}^{\ell} P_{i,j'}}) \leqslant g(|t|)\,,$$

where $1 \leqslant i \leqslant k$, $1 \leqslant j \leqslant \ell$. Note that the function g can be inferred from the used reduction pairs.

In the sequel of this section, we show that dc_R is multiply recursive whenever the function g is multiply recursive. To this end, we make use of a simulating TRS R' (depending on k, A, C, and the function g). The idea behind R' is that it can simulate any derivation of R, and can thus be employed to show that dc_R is multiply recursive. We define the mapping $\mathsf{dh}^\sharp \colon \mathrm{T}(F,V) \to \mathbb{N} \times \mathbb{N}^k$ as follows:

$$\mathsf{dh}^\sharp(t,p) = \begin{cases} (i, \mathsf{dh}(t^\sharp, \to_{(P_{i,1}/R \cup \bigcup_{j=2}^{\ell} P_{i,j})}), \ldots, \mathsf{dh}(t^\sharp, \to_{(P_{i,\ell}/R)})) & \text{if } t^\sharp \notin \mathsf{NF}(P_i/R) \land \mathsf{rk}(t,p) = i\,, \\ (0,\ldots,0,1) & \text{if } t^\sharp \in \mathsf{NF}(\mathsf{DP}(R)/R) \land \mathsf{rt}(t|_p) \in D\,, \\ (0,\ldots,0) & \text{otherwise}\,. \end{cases}$$

Note that, if $\mathsf{dh}^\sharp(t,p) = (i, i_1, \ldots, i_\ell)$ with $i > 0$, then $\max\{i_1,\ldots,i_\ell\} > 0$, as well. We give some intuition for this definition. Consider an arbitrary SCC P_i, a term t and a position p in t such that $\mathsf{rk}(t|_p) = i$. Suppose that $t|_p$ is not in normal form with respect to P_i/R. Then for once, we need to estimate $\mathsf{dh}(t^\sharp, \to_{(P_{i,j}/R \cup \bigcup_{j'=j+1}^{\ell} P_{i,j'})})$ for all $j \in \{1,\ldots,\ell\}$. On the other hand the case that $t|_p \in \mathsf{NF}(\mathsf{DP}(R)/R)$ has to be accounted for. For any rewrite step applied to t with redex position p, and any position p' "created" by that step we have $\mathsf{dh}^\sharp(t,p) >^{\mathrm{lex}} \mathsf{dh}^\sharp(t,p')$ in both cases.

The simulating TRS R' is based on a mapping tr such that $s \to_R t$ implies $\mathsf{tr}(s) \to_{R'}^{+} \mathsf{tr}(t)$. Given a term t, tr essentially assigns to each position p of t an $A+\ell$-ary function symbol f_i. The index i of f_i and the first ℓ arguments are used to represent $\mathsf{dh}^\sharp(t,p)$.

Definition 6. *Let $t = f(t_1,\ldots,t_n)$ and $(i,i_1,\ldots,i_\ell) = \mathsf{dh}^\sharp(t^\sharp)$. Then the mapping $\mathsf{tr} : \mathrm{T}(R) \to \mathrm{T}(R')$ is defined as follows:* $\mathsf{tr}(t) = \mathsf{f}_i(\mathsf{S}^{i_1}(\mathsf{0}),\ldots,\mathsf{S}^{i_\ell}(\mathsf{0}),\mathsf{tr}(t_1),\ldots,\mathsf{tr}(t_n),\mathsf{b},\ldots,\mathsf{b})$.

Here we write $\mathsf{S}^n(\mathsf{0})$ as abbrevation for $\mathsf{S}(\cdots(\mathsf{S}(\mathsf{0})))$ (n times S). Note that if f is a constant, that is, $t = f$, then $\mathsf{tr}(t) = \mathsf{f}_i(\mathsf{S}^{i_1}(\mathsf{0}),\ldots,\mathsf{S}^{i_\ell}(\mathsf{0}),\mathsf{b},\ldots,\mathsf{b})$. To simplify the presentation, we compress sequences of b to $\overline{\mathsf{b}}$. Similarly, we use $\overline{x}$ for sequences of x.

The main tool for achieving the simulation of a rewrite step $s \to_R t$ are rules which create the progenies (as defined in [13]) of the redex position p of the step. The set of progenies of p contains at most A^{C+1} many elements, and each proper subterm of $t|_p$ may be duplicated at most that many times. In order to write down the right hand sides of these rules concisely for arbitrary A and C, we make use of some abbreviations. We define the shorthand $\vec{x}$ for $(x_1,\ldots,x_A)$. Moreover, we define $M_i^C(n_1,\ldots,n_\ell,\vec{x})$ as follows:

$$M_i^0(n_1,\ldots,n_\ell,\vec{x}) := \mathsf{f}_i(n_1,\ldots,n_\ell,\vec{x})$$
$$M_i^{j+1}(n_1,\ldots,n_\ell,\vec{x}) := \mathsf{f}_i(n_1,\ldots,n_\ell,M_i^j(n_1,\ldots,n_\ell,\vec{x}),\ldots,M_i^j(n_1,\ldots,n_\ell,\vec{x}))$$

We also define following abbreviation $X(\vec{x})$: $X(\vec{x}) = \mathsf{g}(\mathsf{size}(\mathsf{0},\ldots,\mathsf{0},\vec{x}))$.

Finally, we are in the position to present the rules of the simulating TRS R'.

Definition 7. *Consider the following (schematic) TRS R', where $i \in \{0,\ldots,k\}$, $i' \in \{1,\ldots,k\}$, $j \in \{1,\ldots,\ell\}$, and $j' \in \{1,\ldots,A\}$.*

$$\begin{array}{rr} 1_{i,j}: & \mathsf{f}_i(n_1,\ldots,n_{j-1},\mathsf{S}(n_j),n_{j+1},\ldots,n_\ell,\vec{x}) \to M_i^C(n_1,\ldots,n_j,X(\vec{x}),\ldots,X(\vec{x}),\vec{x}) \\ 2_{i'}: & \mathsf{f}_{i'}(n_1,\ldots,n_\ell,\vec{x}) \to f_{i'-1}(X(\vec{x}),\ldots,X(\vec{x}),\vec{x}) \\ 3_{i,j'}: & \mathsf{size}(\mathsf{f}_i(n_1,\ldots,n_\ell,\vec{x})) \to \mathsf{d}_A(\mathsf{size}(x_{j'})) \\ 4: & \mathsf{size}(\mathsf{b}) \to \mathsf{S}(\mathsf{0}) \\ 5: & \mathsf{d}_A(\mathsf{S}(x)) \to \mathsf{S}^A(\mathsf{d}_A(x)))) \\ 6: & \mathsf{d}_A(\mathsf{0}) \to \mathsf{0} \\ 7: & \mathsf{f}_0(n_1,\ldots,n_\ell,\vec{x}) \to \mathsf{b} \\ 8_{i,j'}: & \mathsf{f}_i(n_1,\ldots,n_\ell,\vec{x}) \to x_{j'} \\ 9: & \mathsf{f}(x) \to \mathsf{f}_k(X(\overline{x}),\ldots,X(\overline{x}),\overline{x}) \\ 10: & \mathsf{z} \to \mathsf{f}_k(X(\overline{\mathsf{b}}),\ldots,X(\overline{\mathsf{b}}),\overline{\mathsf{b}}) \end{array}$$

These rules are augmented by rules defining the function symbol g*, that is, we choose some finite TRS* R'' *(employing disjoint defined function symbols with the exception of* g*) such that* S *and* $\mathsf{0}$ *are constructor symbols, and the unique normal form of* $\mathsf{g}(\mathsf{S}^n(\mathsf{0}))$ *is* $\mathsf{S}^{g(n)}(\mathsf{0})$ *(with respect to* R''*).*

Recall that it is not difficult to define a suitable TRS R'', whenever g is computable. Moreover, whenever g is a primitive recursive function, it is possible to give such a TRS R'', whose termination can be shown by the lexicographic path order (LPO).

We motivate the rules of R'. The rules $1_{i,j}$ are the main rules for the simulation of the effects of a single step $s \to_R t$ in R'. One rewrite step in s with redex position p creates at most A^{C+1} many new positions p' with $\mathsf{dh}^\sharp(t,p) >^{\text{lex}} \mathsf{dh}^\sharp(t,p')$, and the subterms of $t|_p$ are duplicated at most A^{C+1} times. Observe that a decrease in the jth argument of f_i may "reset" the value of arguments $j+1$ to ℓ, which explains the occurrence of the values $X(\vec{x})$ on the right hand side of these rules. The rules $2_{i'}$ simulate that each of the new positions q created by $s \to t$ might be of rank j ($i' > j$). The rules $3_{i,j'}$–6 define the function symbol size, that is, $\mathsf{size}(s)$ reduces to a numeral $\mathsf{S}^l(0)$ such that $l \geqslant |s|$. The rules 7–$8_{i,j'}$ make sure that any superfluous positions and copies of subterms created by the rules of type $1_{i,j}$ can be deleted. Finally, the rules 9 and 10 guarantee that the simulating derivation can be started with a small enough initial term (see also the second part of Lemma 8 below). Recall that the function $\mathsf{dc}_{R'}$ bounds the relationship between $|t|$ and $\mathsf{dh}(t, \to_{R'})$, and in general it is not the case that $|\mathsf{tr}(s)| \leqslant |s|$, hence we need the second part of Lemma 8.

Lemma 8. *For any ground terms s and t,* $s \to_R t$ *implies* $tr(s) \to^+_{R'} tr(t)$*. Moreover, for any ground term t, we have* $\mathsf{f}^{|t|-1}(\mathsf{z}) \to^+_{R'} tr(t)$*.*

Lemma 8 yields that the length of any derivation in R can be estimated by the maximal derivation length with respect to R'. It remains to verify whether R' has multiply recursive derivational complexity. Whenever g is a multiply recursive function (hence also whenever g is a linear function), it can be encoded by a set of LPO-orientable rules, and then R' can be oriented by LPO. Then by [19] $\mathsf{dc}_{R'}$ is bounded by a multiply recursive function.

Theorem 9. *Let R be a TRS whose termination is provable by Propositions 1 and 2. Moreover, assume that for any reduction pair* $(\gtrsim, \succ)$ *used in any application of Proposition 2, there exists a multiply recursive function g such that* $\mathsf{dh}(t, \succ/\gtrsim) \leqslant g(|t|)$ *for all terms t. Then* dc_R *is bounded by a multiply recursive function.*

For the special case that $\ell = 1$ (which means that in each SCC of $\mathsf{DG}(R)$, exactly one application of Proposition 2 is needed, which removes all dependency pairs in that SCC at once), the *general ramified lexicographic path order* (GRLPO), a restricted version of LPO which has been introduced by Weiermann in [18], can be used instead of LPO in the above argument. Then by [18], $\mathsf{dc}_{R'}$ is bounded by a primitive recursive function.

With some modifications, the considerations made in this paper can be extended to additionally allow the use of Proposition 3 in the termination proof of R. Our main result remains the same for this setting:

Theorem 10. *Let R be a TRS whose termination is provable by Propositions 1, 2 and 3. Moreover, assume that for any reduction pair* $(\gtrsim, \succ)$ *used in any application of Proposition 2, there exists a multiply recursive function g such that* $\mathsf{dh}(t, \succ/\gtrsim) \leqslant g(|t|)$ *for all terms t. Then* dc_R *is bounded by a multiply recursive function.*

This concludes that termination by the restricted version of the DP framework outlined by Propositions 1, 2 and 3 induces multiply recursive derivational complexity. This constitutes a first, but important, step towards the analysis of the (derivational) complexity induced by the DP Framework. In this context it seems important to emphasise that currently no reduction pairs are known which induce non-multiply recursive complexities for finite TRSs.

This leads us to the conjecture that for termination proofs (within the DP framework) based on any of the currently known DP processors the induced derivational complexity of the initial TRS will be multiple-recursive. As a corollary to the conjecture we would obtain a proof that none of the techniques underlying current termination provers is in theory powerful enough to prove termination of Dershowitz's system `TRS/D33-33`, aka the Hydra battle rewrite system (see [3, 12]).

Future work will concentrate on this conjecture as well as an analysis of the DP framework from the point of view of runtime complexity analysis (see [9, 10, 15]).

References

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

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

[3] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 245–319. Elsevier Science, 1990.

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

[5] A. Geser. *Relative Termination*. PhD thesis, Universität Passau, 1990.

[6] J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. 11th LPAR*, volume 3452 of *LNAI*, pages 301–331, 2005.

[7] N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1,2):172–199, 2005.

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

[9] N. Hirokawa and G. Moser. Automated complexity analysis based on the dependency pair method. In *Proc. 4th IJCAR*, volume 5195 of *LNCS*, pages 364–379, 2008.

[10] Nao Hirokawa and Georg Moser. Complexity, graphs, and the dependency pair method. In *Proc. of 15th LPAR*, volume 5330 of *LNCS*, pages 652–666, 2008.

[11] D. Hofbauer. *Termination Proofs and Derivation Lengths in Term Rewriting Systems*. PhD thesis, Technische Universität Berlin, 1992.

[12] G. Moser. The Hydra Battle and Cichon's Principle. *AAECC*, 20(2):133–158, 2009.

[13] G. Moser and A. Schnabl. The derivational complexity induced by the dependency pair method. In *Proc. 20th RTA*, volume 5595 of *LNCS*, pages 255–269, 2009.

[14] G. Moser and A. Schnabl. The derivational complexity induced by the dependency pair method, 2010. Draft. Available at `http://cl-informatik.uibk.ac.at/users/aschnabl`.

[15] Georg Moser. Proof theory at work: Complexity analysis of term rewrite systems. *CoRR*, abs/0907.5527, 2009.

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

[17] R. Thiemann. *The DP Framework for Proving Termination of Term Rewriting*. PhD thesis, University of Aachen, 2007.

[18] A. Weiermann. A termination ordering for primitive recursive schemata, 1995. Preprint.

[19] A. Weiermann. Termination proofs for term rewriting systems with lexicographic path orderings imply multiply recursive derivation lengths. *Theoretical Computer Science*, 139(1,2):355–362, 1995.

Polynomial Termination over $\mathbb{N}$ and $\mathbb{R}$ does not imply Polynomial Termination over $\mathbb{Q}$

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

Aart Middeldorp
Institute of Computer Science
University of Innsbruck, Austria
aart.middeldorp@uibk.ac.at

Abstract

Polynomial interpretations are a useful technique for proving termination of term rewrite systems. They come in various flavors: polynomial interpretations over $\mathbb{N}$, $\mathbb{Q}$ and $\mathbb{R}$. In this note we show that there are term rewrite systems that are polynomially terminating over $\mathbb{N}$ and $\mathbb{R}$, but not over $\mathbb{Q}$.

1 Introduction

Polynomial interpretations are a simple yet useful technique for proving termination of term rewrite systems (TRSs, for short). They were introduced in 1979 by Lankford [3], who considered polynomial algebras over the well-founded domain of the natural numbers $\mathbb{N}$, and in the same year, Dershowitz [1] proposed an alternative approach based on polynomial algebras over the real numbers $\mathbb{R}$. However, as the real numbers equipped with the standard order $>_{\mathbb{R}}$ are not well-founded, a subterm property is explicitly required to ensure well-foundedness. In 2005 this limitation was overcome, when Lucas [4] presented his framework for proving polynomial termination over the real and rational numbers. Thus, one can distinguish three variants of polynomial interpretations, polynomial interpretations over $\mathbb{N}$, $\mathbb{Q}$ and $\mathbb{R}$, and the obvious question is how they are related when it comes to termination proving power. Combining the results of Lucas [5] and our recent results in [6] we arrive at Figure 1. The remaining question is whether the area depicted in red is inhabited, i.e., are there TRSs that are polynomially terminating over $\mathbb{N}$ and $\mathbb{R}$, but not over $\mathbb{Q}$? In this note we give an affirmative answer to this question.

2 Preliminaries

We assume familiarity with the basics of term rewriting and polynomial interpretations. Given some $N \in \{\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$ and $m \in N$, $>_N$ denotes the standard order of the respective domain and $N_m := \{x \in N \mid x \geq m\}$. For any ring R (e.g., $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$), we denote the associated *polynomial ring* in *n indeterminates* $x_1, \ldots, x_n$ by $R[x_1, \ldots, x_n]$. In the special case $n = 1$, a polynomial $P \in R[x]$ can be written as follows:

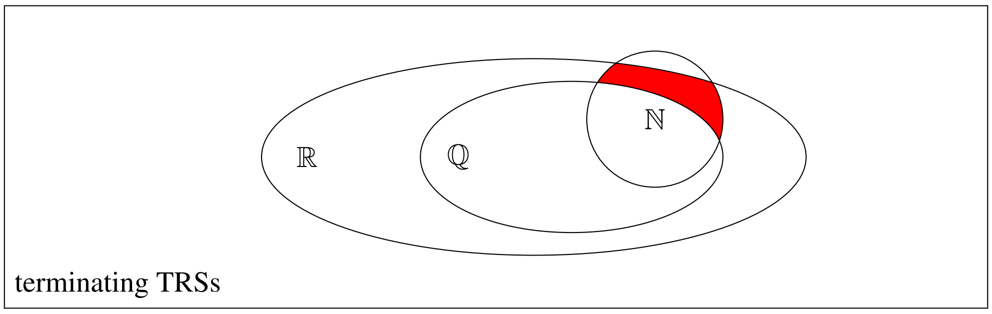

Figure 1: Comparison.

$P(x) = \sum_{k=0}^{d} a_k x^k$. For the largest k such that $a_k \neq 0$, we call $a_k x^k$ the *leading term* of P, a_k its *leading coefficient* and k its *degree*, which is denoted by $\deg(P) = k$. A polynomial $P \in R[x]$ is said to be *linear* if $\deg(P) = 1$, and *quadratic* if $\deg(P) = 2$.

The key concept for establishing (direct) termination of TRSs via polynomial interpretations is the notion of well-founded monotone algebras as they induce reduction orders on terms. Let $\mathscr{F}$ be a signature, i.e., a set of function symbols equipped with fixed arities. A *(well-founded) monotone $\mathscr{F}$-algebra* $(\mathscr{A}, >_A)$ is a non-empty algebra $\mathscr{A} = (A, \{f_A\}_{f \in \mathscr{F}})$ together with a (well-founded) order $>_A$ on the *carrier A* of $\mathscr{A}$ such that every algebra operation f_A is strictly monotone in all arguments, i.e., if $f \in \mathscr{F}$ has arity $n \geq 1$ then $f_A(a_1, \ldots, a_i, \ldots, a_n) >_A f_A(a_1, \ldots, b, \ldots, a_n)$ for all $a_1, \ldots, a_n, b \in A$ and $i \in \{1, \ldots, n\}$ with $a_i >_A b$. Moreover, every function symbol $f \in \mathscr{F}$ is said to be *interpreted* by its associated *interpretation function f_A*. Given a monotone algebra $(\mathscr{A}, >_A)$, we define the relations $\succsim_{\mathscr{A}}$ and $\succ_{\mathscr{A}}$ on terms as follows: $s \succsim_{\mathscr{A}} t$ if $[\alpha]_{\mathscr{A}}(s) \geq_A [\alpha]_{\mathscr{A}}(t)$ and $s \succ_{\mathscr{A}} t$ if $[\alpha]_{\mathscr{A}}(s) >_A [\alpha]_{\mathscr{A}}(t)$, for all assignments α of elements of A to the variables in s and t ($[\alpha]_{\mathscr{A}}(\cdot)$ denotes the usual evaluation function associated with the algebra $\mathscr{A}$). If $(\mathscr{A}, >_A)$ is a well-founded monotone algebra, then $\succ_{\mathscr{A}}$ is a reduction order. It is well-known that well-founded monotone algebras provide a complete characterisation of termination: A TRS $\mathscr{R}$ is terminating if and only if it is compatible with a well-founded monotone algebra $(\mathscr{A}, >_A)$, that is, $l \succ_{\mathscr{A}} r$ for every rewrite rule $l \to r \in \mathscr{R}$.

A *polynomial interpretation over* $\mathbb{N}$ for a signature $\mathscr{F}$ consists of a polynomial $f_{\mathbb{N}} \in \mathbb{Z}[x_1, \ldots, x_n]$ for every n-ary function symbol $f \in \mathscr{F}$, such that the following two properties hold:

1. *well-definedness*: $f_{\mathbb{N}}(x_1, \ldots, x_n) \in \mathbb{N}$ for all $x_1, \ldots, x_n \in \mathbb{N}$,

2. *strict monotonicity* of $f_{\mathbb{N}}$ in all arguments with respect to $>_{\mathbb{N}}$, the standard order on $\mathbb{N}$.

Now $(\mathbb{N}, \{f_{\mathbb{N}}\}_{f \in \mathscr{F}}, >_{\mathbb{N}})$ is a well-founded monotone algebra, and we say that a polynomial interpretation over $\mathbb{N}$ is *compatible* with a TRS $\mathscr{R}$ if the well-founded monotone algebra $(\mathbb{N}, \{f_{\mathbb{N}}\}_{f \in \mathscr{F}}, >_{\mathbb{N}})$ is compatible with $\mathscr{R}$; in this case, $\mathscr{R}$ is said to be *polynomially terminating over* $\mathbb{N}$.

If one wants to extend the notion of polynomial interpretations to the rational or real numbers, the main problem one is confronted with is the non-well-foundedness of these domains with respect to the standard orders $>_{\mathbb{Q}}$ and $>_{\mathbb{R}}$. In [2, 4], this problem is overcome by replacing these orders with new non-total orders $>_{\mathbb{R},\delta}$ and $>_{\mathbb{Q},\delta}$, the first of which is defined as follows: given some fixed positive real number δ, $x >_{\mathbb{R},\delta} y :\iff x - y \geq_{\mathbb{R}} \delta$ for all $x, y \in \mathbb{R}$. Analogously, one defines $>_{\mathbb{Q},\delta}$ on $\mathbb{Q}$. Thus, $>_{\mathbb{R},\delta}$ ($>_{\mathbb{Q},\delta}$) is well-founded on subsets of $\mathbb{R}$ ($\mathbb{Q}$) that are bounded from below.

A *polynomial interpretation over* $\mathbb{R}$ for a signature $\mathscr{F}$ consists of a polynomial $f_{\mathbb{R}} \in \mathbb{R}[x_1, \ldots, x_n]$ for every n-ary function symbol $f \in \mathscr{F}$ and some positive real number $\delta >_{\mathbb{R}} 0$, such that:

a) *well-definedness*: $f_{\mathbb{R}}(x_1, \ldots, x_n) \in \mathbb{R}_0$ for all $x_1, \ldots, x_n \in \mathbb{R}_0$,

b) *strict monotonicity* of $f_{\mathbb{R}}$ in all arguments with respect to $>_{\mathbb{R}_0,\delta}$, the restriction of $>_{\mathbb{R},\delta}$ to $\mathbb{R}_0$.

Analogously, one defines polynomial interpretations over $\mathbb{Q}$ by the obvious adaptation of the definition above. Again, $(\mathbb{R}_0, \{f_{\mathbb{R}}\}_{f \in \mathscr{F}}, >_{\mathbb{R}_0,\delta})$ and $(\mathbb{Q}_0, \{f_{\mathbb{Q}}\}_{f \in \mathscr{F}}, >_{\mathbb{Q}_0,\delta})$ are well-founded monotone algebras, and we say that a TRS is *polynomially terminating over* $\mathbb{R}$ ($\mathbb{Q}$) if it is compatible with such an algebra.

3 Main Result

In this section we present a TRS that is polynomially terminating over both $\mathbb{N}$ and $\mathbb{R}$, but not over $\mathbb{Q}$. In order to motivate the construction underlying this TRS, let us consider the following quantified polynomial inequality

$$\forall x \quad (2x^2 - x) \cdot P(a) \geqslant 0 \tag{$*$}$$

where $P \in \mathbb{Z}[a]$ is a polynomial with integer coefficients, all of whose roots are irrational and which is positive for some non-negative integer value of a. To be concrete, let us take $P(a) = a^2 - 2$ and try to satisfy $(*)$ in $\mathbb{N}$, $\mathbb{Q}_0$ and $\mathbb{R}_0$, respectively. To this end, we observe that $a := \sqrt{2}$ is a satisfying assignment in $\mathbb{R}_0$. Moreover, $(*)$ is also satisfiable in $\mathbb{N}$ by assigning $a := 2$, for example, and observing that the polynomial $2x^2 - x$ is non-negative for all $x \in \mathbb{N}$. However, $(*)$ cannot be satisfied in $\mathbb{Q}_0$ as non-negativity of $2x^2 - x$ does not hold for all $x \in \mathbb{Q}_0$ and P has no rational roots. To sum up, $(*)$ is satisfiable in $\mathbb{N}$ and $\mathbb{R}_0$, but not in $\mathbb{Q}_0$. Thus, the basic idea now is to design a TRS containing some rewrite rule whose compatibility constraint reduces to a polynomial inequality similar in nature to $(*)$. To this end, we rewrite the inequality $(2x^2 - x) \cdot (a^2 - 2) \geqslant 0$ to

$$2a^2x^2 + 2x \geq 4x^2 + a^2x$$

because now both the left- and right-hand side can be viewed as a composition of several functions, each of which is strictly monotone and well-defined. In particular, we identify the following constituents: $\mathsf{h}(x,y) := x + y$, $\mathsf{r}(x) := 2x$, $\mathsf{p}(x) := x^2$ and $\mathsf{k}(x) := a^2x$. Thus, the above inequality can be written in the form

$$\mathsf{h}(\mathsf{r}(\mathsf{k}(\mathsf{p}(x))), \mathsf{r}(x)) \geq \mathsf{h}(\mathsf{r}(\mathsf{r}(\mathsf{p}(x))), \mathsf{k}(x)) \qquad (**)$$

which can easily be modeled as a rewrite rule. (Note that $\mathsf{r}(x)$ is not strictly necessary as $\mathsf{r}(x) = \mathsf{h}(x,x)$, but it gives rise to a shorter encoding.) And then we also need rewrite rules that enforce the desired interpretations for the function symbols h, r, p and k. To this end, we leverage the techniques presented in [6], in particular the method of polynomial interpolation. The resulting TRS will be referred to as $\mathscr{R}$, and it consists of the following rules:

$$\begin{aligned}
\mathsf{f}(\mathsf{g}(x)) &\to \mathsf{g}^2(\mathsf{f}(x)) && (1)\\
\mathsf{g}(\mathsf{s}(x)) &\to \mathsf{s}^2(\mathsf{g}(x)) && (2)\\
\mathsf{s}(x) &\to \mathsf{h}(\mathsf{0},x) && (3)\\
\mathsf{s}(x) &\to \mathsf{h}(x,\mathsf{0}) && (4)\\
\mathsf{f}(\mathsf{0}) &\to \mathsf{0} && (5)\\
\mathsf{s}^3(\mathsf{0}) &\to \mathsf{f}(\mathsf{s}(\mathsf{0})) && (6)\\
\mathsf{f}(\mathsf{s}(\mathsf{0})) &\to \mathsf{s}(\mathsf{0}) && (7)\\
\mathsf{h}(\mathsf{f}(x),\mathsf{g}(x)) &\to \mathsf{f}(\mathsf{s}(x)) && (8)\\
\mathsf{g}(x) &\to \mathsf{h}(\mathsf{h}(\mathsf{h}(\mathsf{h}(x,x),x),x),x) && (9)\\
\mathsf{f}(\mathsf{s}^2(x)) &\to \mathsf{h}(\mathsf{f}(x),\mathsf{g}(\mathsf{h}(x,x))) && (10)
\end{aligned}$$

$$\begin{aligned}
\mathsf{s}(\mathsf{0}) &\to \mathsf{r}(\mathsf{0}) && (11)\\
\mathsf{s}^3(\mathsf{0}) &\to \mathsf{r}(\mathsf{s}(\mathsf{0})) && (12)\\
\mathsf{r}(\mathsf{s}(\mathsf{0})) &\to \mathsf{s}(\mathsf{0}) && (13)\\
\mathsf{g}(x) &\to \mathsf{r}(x) && (14)
\end{aligned}$$

$$\begin{aligned}
\mathsf{s}(\mathsf{0}) &\to \mathsf{p}(\mathsf{0}) && (15)\\
\mathsf{s}^2(\mathsf{0}) &\to \mathsf{p}(\mathsf{s}(\mathsf{0})) && (16)\\
\mathsf{p}(\mathsf{s}(\mathsf{0})) &\to \mathsf{0} && (17)\\
\mathsf{s}^5(\mathsf{0}) &\to \mathsf{p}(\mathsf{s}^2(\mathsf{0})) && (18)\\
\mathsf{p}(\mathsf{s}^2(\mathsf{0})) &\to \mathsf{s}^3(\mathsf{0}) && (19)\\
\mathsf{h}(\mathsf{p}(x),\mathsf{g}(x)) &\to \mathsf{p}(\mathsf{s}(x)) && (20)
\end{aligned}$$

$$\begin{aligned}
\mathsf{s}(\mathsf{0}) &\to \mathsf{k}(\mathsf{0}) && (21)\\
\mathsf{s}^2(\mathsf{p}^2(\mathsf{a})) &\to \mathsf{s}(\mathsf{k}(\mathsf{p}(\mathsf{a}))) && (22)\\
\mathsf{s}(\mathsf{k}(\mathsf{p}(\mathsf{a}))) &\to \mathsf{p}^2(\mathsf{a}) && (23)\\
\mathsf{g}(x) &\to \mathsf{k}(x) && (24)\\
\mathsf{a} &\to \mathsf{0} && (25)
\end{aligned}$$

$$\mathsf{s}(\mathsf{h}(\mathsf{r}(\mathsf{k}(\mathsf{p}(x))),\mathsf{r}(x))) \to \mathsf{h}(\mathsf{r}^2(\mathsf{p}(x)),\mathsf{k}(x)) \qquad (26)$$

Each of the blocks serves a specific purpose. The largest block consists of the rules (1) – (10) and is basically a slightly modified version of the main TRS of [6, Section 4]. These rules ensure that the symbol s has the semantics of a *successor function* $x \mapsto x + s_0$. Moreover, for any compatible polynomial interpretation over $\mathbb{Q}$ ($\mathbb{R}$), it is guaranteed that s_0 is equal to δ, the minimal step width of the order $>_{\mathbb{Q},\delta}$. In [6], these conditions are identified as the key requirements for the method of polynomial interpolation to work in this setting. Finally, this block also enforces $\mathsf{h}(x,y) := x + y$. The next block, consisting of the rules (11) – (14), makes use of polynomial interpolation to achieve $\mathsf{r}(x) := 2x$. Likewise, the block consisting of the rules (15) – (20) equips the symbol p with the semantics of a squaring function. And the block (21) – (25) enforces the desired semantics for the symbol k, i.e., a linear function $x \mapsto k_1 x$ whose slope k_1 is proportional to the square of the interpretation of the constant a. Finally, the rule (26) encodes the main idea presented at the beginning of this section (cf. $(**)$).

Lemma 3.1. *The TRS $\mathscr{R}$ is polynomially terminating over $\mathbb{N}$ and $\mathbb{R}$.*

Proof. For polynomial termination over $\mathbb{N}$, the following interpretation applies:

$$\begin{array}{ccccc} \mathsf{0}_{\mathbb{N}} := 0 & \mathsf{s}_{\mathbb{N}}(x) := x+1 & \mathsf{f}_{\mathbb{N}}(x) := 3x^2-2x+1 & \mathsf{g}_{\mathbb{N}}(x) := 6x+6 & \\ \mathsf{h}_{\mathbb{N}}(x,y) := x+y & \mathsf{p}_{\mathbb{N}}(x) := x^2 & \mathsf{r}_{\mathbb{N}}(x) := 2x & \mathsf{k}_{\mathbb{N}}(x) := 4x & \mathsf{a}_{\mathbb{N}} := 2 \end{array}$$

Rule (26) gives rise to the constraint

$$8x^2+2x+1 > 4x^2+4x \qquad \Longleftrightarrow \qquad 4x^2-2x+1 > 0$$

which holds for all $x \in \mathbb{N}$. For polynomial termination over $\mathbb{R}$, we let $\delta := 1$ but we have to modify the interpretation as $4x^2-2x+1 >_{\mathbb{R}_0,\delta} 0$ does not hold for all $x \in \mathbb{R}_0$. Taking $\mathsf{a}_{\mathbb{R}} := \sqrt{2}$, $\mathsf{k}_{\mathbb{R}}(x) := 2x$ and the above interpretations for the other function symbols establishes polynomial termination over $\mathbb{R}$. Note that the constraint $4x^2+2x+1 >_{\mathbb{R}_0,\delta} 4x^2+2x$ associated with rule (26) trivially holds. Moreover, the function $\mathsf{f}_{\mathbb{R}}(x) := 3x^2-2x+1$ is strictly monotone with respect to $>_{\mathbb{R}_0,\delta}$ if and only if $\mathsf{f}_{\mathbb{R}}(x+h) - \mathsf{f}_{\mathbb{R}}(x) \geq \delta$ for all non-negative real numbers x and $h \geq \delta$. Thus, we have to show that $h(6x+3h-2) \geq 1$ for all non-negative real numbers x and $h \geq 1$, which is easy. Finally, monotonicity of $\mathsf{p}_{\mathbb{R}}$ can be shown in the same way. □

Lemma 3.2. *The TRS $\mathscr{R}$ is not polynomially terminating over $\mathbb{Q}$.*

Proof. Let us assume that $\mathscr{R}$ is polynomially terminating over $\mathbb{Q}$ and derive a contradiction. Adapting the reasoning in the proof of [6, Lemma 4.4], we infer from compatibility with the rules (1) – (9) that $\mathsf{s}_{\mathbb{Q}}(x) = x + s_0$, $\mathsf{g}_{\mathbb{Q}}(x) = g_1 x + g_0$, $\mathsf{h}_{\mathbb{Q}}(x,y) = x+y+h_0$, and $\mathsf{f}_{\mathbb{Q}}(x) = ax^2+bx+c$, subject to the following constraints:

$$s_0 >_{\mathbb{Q}} 0 \qquad g_1 >_{\mathbb{Q}} 2 \qquad g_0, h_0 \in \mathbb{Q}_0 \qquad a >_{\mathbb{Q}} 0 \qquad c \geq_{\mathbb{Q}} 0 \qquad a\delta + b \geq_{\mathbb{Q}} 1$$

Next we consider the compatibility constraint associated with rules (9) and (10), from which we deduce an important auxiliary result. Compatibility with rule (9) implies the condition $g_1 \geq_{\mathbb{Q}} 5$ on the respective leading coefficients since $\mathsf{h}_{\mathbb{Q}}(x,y) = x+y+h_0$, and compatibility with rule (10) simplifies to

$$4as_0x + 4as_0^2 + 2bs_0 \geq_{\mathbb{Q}} 2g_1x + g_1h_0 + g_0 + h_0 + \delta \quad \text{for all } x \in \mathbb{Q}_0,$$

from which we infer $4as_0 \geq_{\mathbb{Q}} 2g_1$; from this and $g_1 \geq_{\mathbb{Q}} 5$, we conclude $as_0 >_{\mathbb{Q}} 2$.

Now let us consider the combined compatibility constraint imposed by rules (6) and (7), namely $\mathsf{0}_{\mathbb{Q}} + 3s_0 >_{\mathbb{Q}_0,\delta} \mathsf{f}_{\mathbb{Q}}(\mathsf{s}_{\mathbb{Q}}(\mathsf{0}_{\mathbb{Q}})) >_{\mathbb{Q}_0,\delta} \mathsf{0}_{\mathbb{Q}} + s_0$, which implies $\mathsf{0}_{\mathbb{Q}} + 3s_0 \geq_{\mathbb{Q}} \mathsf{0}_{\mathbb{Q}} + s_0 + 2\delta$ by definition of $>_{\mathbb{Q}_0,\delta}$. Thus, we conclude $s_0 \geq_{\mathbb{Q}} \delta$. In fact, we even have $s_0 = \delta$, which can be derived from the compatibility constraint of rule (6) using the conditions $s_0 \geq_{\mathbb{Q}} \delta$, $a\delta + b \geq_{\mathbb{Q}} 1$, $as_0 + b \geq_{\mathbb{Q}} 1$, the combination of the former two conditions, and $\mathsf{f}_{\mathbb{Q}}(\mathsf{0}_{\mathbb{Q}}) \geq_{\mathbb{Q}} \mathsf{0}_{\mathbb{Q}} + \delta$, the compatibility constraint of rule (5):

$$\begin{aligned} \mathsf{0}_{\mathbb{Q}} + 3s_0 - \delta \ &\geq_{\mathbb{Q}} \ \mathsf{f}_{\mathbb{Q}}(\mathsf{s}_{\mathbb{Q}}(\mathsf{0}_{\mathbb{Q}})) \ = \ \mathsf{f}_{\mathbb{Q}}(\mathsf{0}_{\mathbb{Q}}) + 2a\mathsf{0}_{\mathbb{Q}}s_0 + as_0^2 + bs_0 \\ &\geq_{\mathbb{Q}} \ \mathsf{0}_{\mathbb{Q}} + \delta + as_0^2 + bs_0 \\ &\geq_{\mathbb{Q}} \ \mathsf{0}_{\mathbb{Q}} + \delta + as_0^2 + (1-a\delta)s_0 \ = \ \mathsf{0}_{\mathbb{Q}} + s_0 + \delta + as_0(s_0 - \delta) \end{aligned}$$

Hence, $0_{\mathbb{Q}} + 3s_0 - \delta \geq_{\mathbb{Q}} 0_{\mathbb{Q}} + s_0 + \delta + as_0(s_0 - \delta)$, or equivalently, $2(s_0 - \delta) \geq_{\mathbb{Q}} as_0(s_0 - \delta)$. But since $as_0 >_{\mathbb{Q}} 2$ and $s_0 \geq_{\mathbb{Q}} \delta$, this inequality can only hold if

$$s_0 = \delta \qquad (***)$$

This result has immediate consequences concerning the interpretation of the constant 0. To this end, we consider the compatibility constraint of rule (3), which simplifies to $s_0 \geq_{\mathbb{Q}} 0_{\mathbb{Q}} + h_0 + \delta$. Because of $(***)$ and the fact that $0_{\mathbb{Q}}$ and h_0 must be non-negative, we conclude $0_{\mathbb{Q}} = h_0 = 0$. Moreover, condition $(***)$ is the key to the proof of this lemma. To this end, we consider the compatibility constraints associated with rules (15) – (19). By definition of $>_{\mathbb{Q}_0,s_0}$, these constraints give rise to the following system of equations:

$$\mathsf{p}_{\mathbb{Q}}(0) = 0 \qquad \mathsf{p}_{\mathbb{Q}}(s_0) = s_0 \qquad \mathsf{p}_{\mathbb{Q}}(2s_0) = 4s_0$$

Viewing these equations as polynomial interpolation constraints, we conclude that no linear polynomial can satisfy them (because $s_0 \neq 0$). Hence, $\mathsf{p}_{\mathbb{Q}}$ must at least be quadratic. Moreover, by rule (20), $\mathsf{p}_{\mathbb{Q}}$ is at most quadratic (using the same reasoning as for rule (8)). So we let $\mathsf{p}_{\mathbb{Q}}(x) := p_2x^2 + p_1x + p_0$ in the equations above and infer the (unique) solution $p_0 = p_1 = 0$ and $p_2s_0 = 1$; that is, $\mathsf{p}_{\mathbb{Q}}(x) = p_2x^2$, $p_2 \neq 0$.

Next we consider the compatibility constraints associated with rules (11) – (13), from which we deduce the interpolation constraints $\mathsf{r}_{\mathbb{Q}}(0) = 0$ and $\mathsf{r}_{\mathbb{Q}}(s_0) = 2s_0$. Because $\mathsf{g}_{\mathbb{Q}}$ is linear, $\mathsf{r}_{\mathbb{Q}}$ must be linear, too, for compatibility with rule (14). Hence, by polynomial interpolation, $\mathsf{r}_{\mathbb{Q}}(x) = 2x$. Likewise, $\mathsf{k}_{\mathbb{Q}}$ must be linear for compatibility with rule (24); that is, $\mathsf{k}_{\mathbb{Q}}(x) = k_1x + k_0$. In particular, $k_0 = 0$ due to compatibility with rule (21), and then the compatibility constraints associated with rule (22) and rule (23) yield $p_2^3\mathsf{a}_{\mathbb{Q}}^4 + 2s_0 - \delta \geq_{\mathbb{Q}} k_1p_2\mathsf{a}_{\mathbb{Q}}^2 + s_0 \geq_{\mathbb{Q}} p_2^3\mathsf{a}_{\mathbb{Q}}^4 + \delta$. But $s_0 = \delta$, hence $k_1p_2\mathsf{a}_{\mathbb{Q}}^2 = p_2^3\mathsf{a}_{\mathbb{Q}}^4$, and since $\mathsf{a}_{\mathbb{Q}}$ cannot be zero due to compatibility with rule (25), we obtain $k_1 = p_2^2\mathsf{a}_{\mathbb{Q}}^2$. In other words, $\mathsf{k}_{\mathbb{Q}}(x) = p_2^2\mathsf{a}_{\mathbb{Q}}^2x$.

Finally, we consider the compatibility constraint associated with rule (26), which simplifies to

$$(2p_2x^2 - x)((p_2\mathsf{a}_{\mathbb{Q}})^2 - 2) \geq 0 \quad \text{for all } x \in \mathbb{Q}_0$$

However, this inequality is unsatisfiable as the polynomial $2p_2x^2 - x$ is negative for some $x \in \mathbb{Q}_0$ and $(p_2\mathsf{a}_{\mathbb{Q}})^2 - 2$ cannot be zero because both p_2 and $\mathsf{a}_{\mathbb{Q}}$ must be rational numbers. $\square$

References

[1] N. Dershowitz. A note on simplification orderings. *Information Processing Letters*, 9(5):212–215, 1979.

[2] D. Hofbauer. Termination proofs by context-dependent interpretations. In *Proc. 12th International Conference on Rewriting Techniques and Applications (RTA 2001)*, volume 2051 of *Lecture Notes in Computer Science*, pages 108–121, 2001.

[3] D. Lankford. On proving term rewrite systems are noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, 1979.

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

[5] 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.

[6] F. Neurauter and A. Middeldorp. Polynomial interpretations over the reals do not subsume polynomial interpretations over the integers. In *Proc. 21st International Conference on Rewriting Techniques and Applications (RTA 2010)*, Leibniz International Proceedings in Informatics, 2010. To appear.

A Computability Path Ordering for Polymorphic Terms

Jean-Pierre Jouannaud[1] and Jian-Qi Li[2] ⋆

[1] INRIA-LIAMA and the Software Chair at Tsinghua University, Beijing
[2] School of Software, Tsinghua University, Beijing

1 Introduction

It is our program to instrument higher-order rewriting with semi- automated tools. The important properties to be checked are well known: type preservation, confluence and strong normalization. In this paper, we consider the strong normalization property. Several directions have been investigated, whether in the plain or higher-order pattern matching case which we shall not describe for lack of space.
In a series of papers, we have developped first the higher-order recursive path ordering [5], then the computability path ordering [1], which structure is simpler although it is more expressive ; a preliminary version of the higher-order recursive path ordering [3] has been equiped with semantic information in a generic way which applies to later versions [2]. All these orderings allow showing strong normalization properties of rewriting based on plain pattern matching. They all assume either a simple type discipline or weak polymorphism à la ML, and are all implemented [5]. We have also shown how these orderings can be simply modified to show strong normalization of rules based on higher-order pattern matching [4].
To achieve our goal, our strategy is: first, to generalize the computability path ordering in order to accomodate a richer type discipline, including first full polymorphism, then predicate types and dependent types; second, to incorporate semantic information into the order, as provided for example by size interpretations, following the lines of [2]; third to develop a version for higher-order pattern matching, following the lines in [4]. *The present paper is the first step of this program: a new version of the computability path ordering is defined and proved well-founded over polymorphic lambda terms.*
Like its predecessors, the *polymorphic computability path order (PCPO)* on polymorphic terms is generated from an ordering on the set of polymorphic types. We take for the latter the computability path ordering (CPO) on the polymorphic type expressions seen as a simply typed lambda structure. In turn, this order is generated from Dershowitz's recursive path ordering (RPO) on kinds seen as a first-order structure. Comparing polymorphic terms in PCPO generates verification conditions for CPO, which in turns generate verification conditions for RPO. This hierarchical mechanism is the reason for our stepwise approach. It also explains why the strong normalization proof of PCPO differs from its predecessors: ordering comparisons in PCPO happen across types and even kinds, making the construction of interpretations "à la Girard" delicate. *The definition and study of reducibility candidates for type-reducing reductions on terms is a second contribution of general interest of this work.*

⋆ This work was partially supported by the Chinese National 973 Plan grant 2010CB328003, the NSFC grants 60635020, 60903030 and NSFC-ANR grant 90718039.

2 Polymorphic Higher-Order Algebras

Given a set $\mathcal{X}_{\mathcal{S}} = \{\alpha, \beta, \ldots\}$ of type variables and a set $\mathcal{S}$ of *sort symbols* of a fixed arity, the set of *types* is generated by the constructors $\rightarrow$, $\forall$ and type application:

$$\mathcal{T}_{\mathcal{S}} := \mathcal{X}_{\mathcal{S}} \mid (\mathcal{T}_{\mathcal{S}} \rightarrow \mathcal{T}_{\mathcal{S}}) \mid \forall \mathcal{X}_{\mathcal{S}}.\mathcal{T}_{\mathcal{S}} \mid (\mathcal{T}_{\mathcal{S}}\ \mathcal{T}_{\mathcal{S}}) \mid s(\mathcal{T}_{\mathcal{S}}^{n}) \quad \text{for } s \in \mathcal{S} \text{ of arity } n$$

The language of types is seen as a (pure) lambda calculus whith $\forall\alpha$ as binding construct. We now build terms, using Λ as constructor for building polymorphic terms, λ for building functions, and function symbols of a fixed arity for building algebraic terms. Given the signature $\mathcal{F} = \biguplus_{\sigma_1,\ldots,\sigma_n,\sigma} \mathcal{F}_{\forall\bar{\alpha}.\sigma_1\times\ldots\times\sigma_n\rightarrow\sigma}$ and a denumerable set $\mathcal{X}$ of variables disjoint from $\mathcal{X}_{\mathcal{S}}$, the set $\mathcal{T}(\mathcal{F},\mathcal{X})$ of *raw terms* is generated by the rules:

$$\mathcal{T} := \mathcal{X} \mid (\lambda\mathcal{X} : \mathcal{T}_{\mathcal{S}}.\mathcal{T}) \mid @(\mathcal{T}\ \mathcal{T}) \mid \mathcal{F}(\mathcal{T},\ldots,\mathcal{T}) \mid \Lambda\mathcal{X}_{\mathcal{S}}.\mathcal{T} \mid @(\mathcal{T}\ \mathcal{T}_{\mathcal{S}}).$$

An *environment* Γ is a finite set of pairs written as $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$, where x_i is a term variable, σ_i is a type, and $x_i \neq x_j$ for $i \neq j$. $\mathcal{V}ar(\Gamma) = \{x_1, \ldots, x_n\}$ is the set of variables of Γ. Our typing judgements are written as $\Gamma \vdash u : \sigma$. A raw term s has type σ in the environment Γ if the judgement $\Gamma \vdash s : \sigma$ is provable in the inference system given at Figure 1. A typable raw term is called a *term*.

Variables:
$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

Functions:
$$\frac{\begin{array}{c} f : \forall\bar{\alpha}.\tau_1 \times \ldots \times \tau_n \rightarrow \tau \in \mathcal{F} \\ \Gamma \vdash t_1 : \tau_1\Theta\ \ldots\ \Gamma \vdash t_n : \tau_n\Theta \end{array}}{\Gamma \vdash f(t_1,\ldots,t_n) : \tau\Theta} \quad (\Theta \text{ a type substitution})$$

Abstraction:
$$\frac{\Gamma \cdot \{x : \sigma\} \vdash t : \tau}{\Gamma \vdash (\lambda x : \sigma.t) : \sigma \rightarrow \tau}$$

Quantification:
$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash (\Lambda\alpha.t) : \forall\alpha.\tau} \quad (\alpha \notin \mathcal{TV}ar(\Gamma))$$

Application:
$$\frac{\Gamma \vdash s : \sigma \rightarrow \tau \quad \Gamma \vdash t : \sigma}{\Gamma \vdash @(s\ t) : \tau}$$

Type Application:
$$\frac{\Gamma \vdash s : \forall\alpha.\sigma}{\Gamma \vdash @(s\ \tau) : \sigma\{\alpha \mapsto \tau\}} \quad (\tau \in \mathcal{T}_{\mathcal{S}})$$

Fig. 1. The type system for polymorphic higher-order algebras

We denote by $\mathcal{TT}_{\sigma}$ the set of terms of type σ, and categorize the set $\mathcal{TT}$ of all terms into three disjoint classes: (i) *Abstractions* headed by λ; (ii) *Polymorphic terms* headed by Λ; (iii) *Neutral* terms are variables, or headed by @ or by a function symbol.
We use $\mathcal{V}ar(e)$, $\mathcal{BV}ar(e)$, $\mathcal{TV}ar(e)$ and $\mathcal{BTV}ar(e)$ for respectively the set of free term variables, bound term variables, free type variables and bound type variables in e. The notation $\bar{e}$ shall be used for a list, a multiset, or a set of expresions $e_1, \ldots, e_n$.
The reduction relations $\rightarrow_{\beta}$, $\rightarrow_{\eta}$, $\rightarrow_{\beta_{\mathcal{T}_{\mathcal{S}}}}$, $\rightarrow_{\eta_{\mathcal{T}_{\mathcal{S}}}}$ on typed terms are defined as follows:
(β) $(\lambda x : \sigma.t\ s) \rightarrow_{\beta} t\{x \mapsto s\}$ (η) $\lambda x : \sigma.(t\ x) \rightarrow_{\eta} t$ **if** $x \notin \mathcal{V}ar(t)$
$(\beta_{\mathcal{T}_{\mathcal{S}}})$ $(\Lambda\alpha.t\ \sigma) \rightarrow_{\beta_{\mathcal{T}_{\mathcal{S}}}} t\{\alpha \mapsto \sigma\}$ $(\eta_{\mathcal{T}_{\mathcal{S}}})$ $\Lambda\alpha.(t\ \alpha) \rightarrow_{\eta_{\mathcal{T}_{\mathcal{S}}}} t$ **if** $\alpha \notin \mathcal{TV}ar(t)$
We shall use: $\rightarrow_{\beta\eta}$ for $\rightarrow_{\beta} \cup \rightarrow_{\eta}$ and $\rightarrow_{\beta\eta\mathcal{T}_{\mathcal{S}}}$ for $\rightarrow_{\beta\mathcal{T}_{\mathcal{S}}} \cup \rightarrow_{\eta\mathcal{T}_{\mathcal{S}}}$.
For an arbitrary binary (infix) relation R, we write $R(s)$ for the set $\{t \mid sRt\}$ of *successors* of s, and $R^{*}(s)$ for the set $\{t \mid sR^{*}t\}$ of its *descendants*.

3 The Polymorphic Computability Path Ordering PCPO

PCPO on terms is generated from the Computability Path Ordering (CPO) on types. In fact, the actual definition of CPO can be abstracted away, using instead its properties in the strong normalization proof of PCPO: any order on types satisfying these properties would do. A consequence is that we do not need to mention the order on kinds used by CPO. Even in examples, a precise definition of CPO will not be needed. Let us use the notations $\geq_{\mathcal{T}_S}$, $=_{\mathcal{T}_S}$ and $>_{\mathcal{T}_S}$, with $\geq_{\mathcal{T}_S}=>_{\mathcal{T}_S} \cup =_{\mathcal{T}_S}$, for respectively the quasi-order on polymorphic types, its associated equivalence and strict order. The properties required from the order on types (and satisfied by CPO) are:

1. $\longrightarrow_{\beta T} \cup \longrightarrow_{\eta T} \subseteq \; >_{\mathcal{T}_S}$;
2. *Well-foundedness*: $\rhd_{\mathcal{T}_S} := \; >_{\mathcal{T}_S} \cup \rhd$ is well-founded;
3. *Stability*: (i) $\sigma =_{\mathcal{T}_S} \tau$ implies $\sigma\Theta =_{\mathcal{T}_S} \tau\Theta$; (ii) $\sigma >_{\mathcal{T}_S} \tau$ implies $\sigma\Theta >_{\mathcal{T}_S} \tau\Theta$;
4. *Right arrow subterm*: $\sigma \to \tau >_{\mathcal{T}_S} \tau$;
5. *Arrow preservation*: $\sigma \to \tau =_{\mathcal{T}_S} \theta$ iff $\theta = \sigma' \to \tau'$, $\sigma' =_{\mathcal{T}_S} \sigma$, $\tau' =_{\mathcal{T}_S} \tau$;
6. *Arrow decreasingness*: $\sigma \to \tau >_{\mathcal{T}_S} \theta$ implies:
   (i) $\theta = \sigma' \to \tau'$, $\sigma' =_{\mathcal{T}_S} \sigma$ and $\tau >_{\mathcal{T}_S} \tau'$; or (ii) $\tau \geq_{\mathcal{T}_S} \theta$;
7. $\forall$ *preservation*: $\forall\alpha.\sigma =_{\mathcal{T}_S} \theta$ iff $\theta = \forall\alpha.\sigma'$ and $\sigma =_{\mathcal{T}_S} \sigma'$;
8. $\forall$ *decreasingness*: $\forall\alpha.\sigma >_{\mathcal{T}_S} \theta$ implies $\theta = \forall\alpha.\sigma'$ and $\sigma >_{\mathcal{T}_S} \sigma'$.

Well-foundedeness and stability together imply *variable preservation*: (i) $\sigma =_{\mathcal{T}_S} \tau$ implies $\mathcal{V}ar(\sigma) = \mathcal{V}ar(\tau)$; and (ii) $\sigma >_{\mathcal{T}_S} \tau$ implies $\mathcal{V}ar(\tau) \subseteq \mathcal{V}ar(\sigma)$.

Viewed as a rewriting relation, PCPO will order terms which types will themselves be ordered by $\geq_{\mathcal{T}_S}$. As a consequence of property (4), rewriting terms may therefore not preserve arrow types. Existing strong normalization proofs in the literature assume preservation of arrows. Our work shows that the above weaker conditions suffice in the context of System *F*.
We now assume given a quasi-order $\geq_{\mathcal{F}}$ called *precedence* on symbols in $\mathcal{F}$, such that $>_F$ is well-founded.
The following notations are used to define the ordering on terms (equality of terms or types being up to α-conversion):

- $s : \sigma \succ_{\mathcal{T}_S} t : \tau$ for $s \succ t$ and $\sigma \geq_{\mathcal{T}_S} \tau$;
- $s : \sigma \succ_{=_{\mathcal{T}_S}} t : \tau$ for $s \succ t$ and $\sigma =_{\mathcal{T}_S} \tau$;
- $s \succ \bar{t}$ for $s \succ u$ for all $u \in \bar{t}$;
- $\succeq$ for the reflexive closure of $\succ$;
- $(>_{\mathcal{T}_S})_{mul}$ for the multiset extension of $>_{\mathcal{T}_S}$.

In [1], the order on terms is indexed by a set of variables which is initialized by the emptyset and grows when abstractions are pulled out from the righthand side terms. This gives extra expressivity to the order, but complicates proofs and notations. We chose here to ease the presentation of PCPO by removing this complication. We also simplify the definition by restricting the set of rules to those that are most meaningfull.

Definition 1 (PCPO-reduction). *Assume that $\Gamma \vdash s : \sigma$, $\Gamma \vdash t : \tau$, and bound variables are disjoint from free variables. Then $s : \sigma \succ t : \tau$ iff either:*

1. *$s = f(\bar{s})$ with $f \in \mathcal{F}$ and either of*
   - *(a) $t = g(\bar{t})$ with $f =_{\mathcal{F}} g \in \mathcal{F}$, $\bar{s}(\succ_{\mathcal{T_S}})_{mul}\bar{t}$*
   - *(b) $t = g(\bar{t})$ with $f >_{\mathcal{F}} g \in \mathcal{F}$ and $s \succ \bar{t}$*
   - *(c) $u \succeq_{\mathcal{T_S}} t$ for some $u \in \bar{s}$*
2. *$s = @(u\ v)$ and either of*
   - *(a) $t = @(u'\ v')$, $u \succ_{\mathcal{T_S}} u'$ and $v \succeq_{=\mathcal{T_S}} v'$ or $u = u'$ and $v \succ_{=\mathcal{T_S}} v'$*
   - *(β) $u = \lambda x : \alpha.w$ and $w\{x \mapsto v\} \succeq t$*
3. *$s = \lambda x : \sigma.u$ and either of*
   - *(a) $t = \lambda y : \tau.v$, $\sigma =_{\mathcal{T_S}} \tau$ and $u\{x \mapsto z\} \succ v\{y \mapsto z\}$ for some fresh variable z*
   - *(η) $u = @(v\ x), x \notin \mathcal{V}ar(v)$ and $v \succ_{\mathcal{T_S}} t$*
4. *$s = @(u\ \sigma)$ and either of*
   - *(a) $t = @(v\ \tau)$, $u \succ_{\mathcal{T_S}} v$, and $\sigma =_{\mathcal{T_S}} \tau$*
   - *($\beta_{\mathcal{T_S}}$) $u = \Lambda\alpha.w$ and $w\{\alpha \mapsto \tau\} \succ_{\mathcal{T_S}} t$*
5. *$s = \Lambda\alpha.u$ and either of*
   - *(a) $t = \Lambda\beta.v$ and $u\{\alpha \mapsto \zeta\} \succ v\{\beta \mapsto \zeta\}$ for some fresh type variable ζ*
   - *($\eta_{\mathcal{T_S}}$) $u = @(v\ \alpha)$, $\alpha \notin \mathcal{TV}ar(v)$ and $v \succeq t$*

As for CPO, recursive calls perform type verifications only when strictly necessary. It is shown in [1] that this is the case when a subterm is taken on left, but useless in other cases. We apply the same principle here, except for Case 5(a) of polymorphic terms (it is not needed).

Example 1 (Computations over polymorphic lists:).

```
sort: List:
⋆ => ⋆, Nat: ⋆
var: A: ⋆, B: ⋆
fun: 0 : nat
fun: s : Nat -> Nat
fun: nil : \/A . list A
fun: cons : \/A . A x List(A) -> List(A)
fun: fold : \/A . Nat x List(A) x (\/B . B -> B -> B) x A
            -> A
var: x,a : A
var: L : List(A)
var: n : Nat
fun: f : \/ A . A -> A -> A
rew: fold(0, L, x) == x
rew: fold(s(n), cons(a, L), x) == f(a, fold(n, L, x))
```

For the first rule, we have:
(1) $fold(0, L, x) : A \succ_{\mathcal{T_S}} x : A$ succeeds by Case 1(c) and the type comparison.
For the second rule,
(2) $fold(s(n), cons(a, L), x) : A \succ_{\mathcal{T_S}} f(a, fold(n, L, x)) : A$
generates a successful type comparison and generates two new subgoal by Case 1(b):
(2.1) $fold(s(n), cons(a, L), f, x) \succeq a$, which generates by Case 1(c) again the goal
(2.1.1) $cons(a, L) : List(A) \succeq_{\mathcal{T_S}} a : A$, which succeeds by Case 1(c) if $List(A) >_{\mathcal{T_S}}$

A, which is true of CPO; and the second subgoal:
(2.2) $fold(s(n), cons(a, L), x) \succ fold(n, L, x)$, which generates by Case 1(a) the subgoal:
(2.2.1) $\{s(n), cons(a, L), x\}(\succ_{\mathcal{T}_S})_{mul}\{n, L, x\}$, which reduces to two subgoals:
(2.2.1.1) $s(n) : Nat \succ_{\mathcal{T}_S} n : Nat$ which succeeds by type comparison and Case 1(c); and the second subgoal:
(2.2.1.2) $cons(a, L) : List(A) \succ_{\mathcal{T}_S} L : List(A)$ which succeeds by type comparison and Case 1(c) again, ending there the computation with success. □

Routine inductive proofs on the definition of $\succ$ show that PCPO enjoys the properties of monotonicity and stability. A major difficulty is that $\succ_{\mathcal{T}_S}$ does *not* preserve types, requiring a specific definition of Girard's candidates allowing for type decreasing rules. We denote by SN the set of $\succ_{\mathcal{T}_S}$-strongly normalizing terms and by SN_σ the set of strongly normalizing terms of type σ. We omit environments for better readability.

Definition 2 (Girard Sets). *Let $\mathfrak{D} = \{\mathcal{D}_\sigma\}_{\mathcal{T}_S}$ be a $\mathcal{T}_S$-indexed family of sets (in short, a* family*), such that $\mathcal{D}_\sigma$ is a nonempty I_σ-indexed family $\{D_{\sigma,i}\}_{I\sigma}$ of non-empty subsets of $\mathcal{TT}_\sigma$ and $T(\mathfrak{D}) = \bigcup_{\sigma\in\mathcal{T}_S, i\in I_\sigma} D_{\sigma,i}$. $\mathfrak{D}$ is a* family of Girard sets *iff for every $\sigma \in \mathcal{T}_S$:*
(G1) $T(\mathfrak{D}) \subseteq SN$;
(G2) for any $D_{\sigma,i}$ and $s : \sigma \in D_{\sigma,i}$, then $\succ_{=\mathcal{T}_S}(s) \subseteq D_{\sigma,i}$;
(G3) for any neutral term $s : \sigma$ and set $D_{\sigma,i}$ such that $\succ_{=\mathcal{T}_S}(s) \subseteq D_{\sigma,i}$, then $s \in D_{\sigma,i}$;
(G4) (maximality): $\mathcal{D}_\sigma$ contains all subsets of SN_σ satisfying (G2-3).

This axiomatic definition of Girard's sets is the natural generalization of reducibility candidates when reductions on terms reduce types in the type ordering: the trick is to speak *indirectly* via (G4) of reductions which are not type preserving.

Theorem 1 (Extensional characterization of Girard sets).
The family $\mathfrak{C} = \{\mathcal{C}_\sigma\}_{\mathcal{T}_S}$ of Girard sets is unique, and contains all subsets of the set of terms that satisfy (G1-3), more precisely: $\mathcal{C}_\sigma = \{C_{\sigma,i} \subseteq SN_\sigma \mid C_{\sigma,i}$ satisfies (G2-3)$\}$.

Theorem 2 (Strong Normalization of PCPO). *If $\Gamma \vdash s : \sigma$, then $s \in SN_{\succ_{\mathcal{T}_S}}$.*

References

1. Frédéric Blanqui, Jean-Pierre Jouannaud, and Albert Rubio. The computability path ordering: The end of a quest. In Kaminski and Martini [6], pages 1–14.
2. Cristina Borralleras and Albert Rubio. A monotonic higher-order semantic path ordering. In Robert Nieuwenhuis and Andrei Voronkov, editors, *LPAR*, volume 2250 of *Lecture Notes in Computer Science*, pages 531–547. Springer, 2001.
3. Jean-Pierre Jouannaud and Albert Rubio. The higher-order recursive path ordering. In *Fourteenth Annual IEEE Symposium on Logic in Computer Science*, Trento, Italy, July 1999.
4. Jean-Pierre Jouannaud and Albert Rubio. Higher-order orderings for normal rewriting. In *Proc. 17th International Conference on Rewriting Techniques and Applications, Seattle, Washington, USA*, 2006.
5. Jean-Pierre Jouannaud and Albert Rubio. Polymorphic higher-order recursive path orderings. *Journal of the ACM*, 2007.
6. Michael Kaminski and Simone Martini, editors. *Computer Science Logic, 22nd International Workshop, CSL 2008, 17th Annual Conference of the EACSL, Bertinoro, Italy, September 16-19, 2008. Proceedings*, volume 5213 of *Lecture Notes in Computer Science*. Springer, 2008.

The Acyclicity Inference of COSTA

Samir Genaim
DSIC, Complutense University of Madrid (UCM), Spain
Damiano Zanardini
CLIP, DIA, Technical University of Madrid (UPM), Spain

1 Introduction

Programming languages with dynamic memory allocation, such as Java, allow creating and manipulating *cyclic* data structures. The presence of cyclic data structures in the program memory (the *heap*) is a challenging issue in the context of termination analysis [4, 5, 1, 14], resource usage analysis [15, 7, 2], garbage collection [11], etc. Consider the loop "**while** (x!=**null**) **do** x:=x.next;". If x points to an acyclic data structure before the loop, then the *depth* of the data structure to which x points strictly decreases after each iteration; therefore, the number of iterations is bounded by the initial depth of (the structure pointed to by) x.

Automatic inference of such information is typically done by (1) *abstracting* the loop to a numeric loop "$while(x) \leftarrow \{x{>}0, x{>}x'\}, while(x')$"; and (2) bounding the number of iterations of the numeric loop. The numeric loop means that, if the loop entry is reached with x pointing to a data structure with depth $x > 0$, then it will eventually be reached again with x pointing to a structure with depth $x' < x$. The key point is that "x!=**null**" is abstracted to the condition $x > 0$, meaning that the depth of a non-null variable cannot be 0; moreover, abstracting "x:=x.next" to $x > x'$ means that the depth decreases when accessing fields. While the former is meaningful for any structure, the latter holds only if x is acyclic. Therefore, acyclicity information is essential in order to apply such abstractions.

In mainstream programming languages with dynamic memory manipulation, data structures can only be modified by means of *field updates*. If, before x.f:=y, x and y are guaranteed to point to disjoint parts of the heap, then there is no possibility to create a cycle. On the other hand, if they are not disjoint, i.e., *share* a common part of the heap, then a cyclic structure might be created. This simple observation has been used in previous work [12] in order to declare x and y, among others, as cyclic whenever they were sharing before the update. Such approach is simple and efficient. However, there can be an important loss of precision in typical programming patterns. E.g., consider "y:=x.next.next;x.next:=y;" (which typically removes an element from a linked list), and let x be initially acyclic. After the first command, x and y clearly share, so that they should be finally declared as cyclic, even if, clearly, they are not. When considering x.f:=y, the precision of the acyclicity information can be improved if it is possible to know *how* x and y share. There are four possible cases: (1) x and y alias; (2) x reaches y; (3) y reaches x; (4) they both reach a common location. The update x:=y.f might create a cycle only in cases (1) and (3).

This abstract summarizes an acyclicity analysis which is based on the above observation as described in [9]. The analysis has been first developed in [10]; more recent work [9] formalizes it in the theory of *abstract interpretation*, and reports on an implementation for *Java bytecode*. The analysis defines an *abstract domain* $\mathscr{I}_{rc}^{\tau}$ which captures the *reachability* information among program variables (i.e., whether there can be a path in the heap from the location ℓ_v bound to some variable v and the location ℓ_w bound to some w), and the *acyclicity* of data structures (i.e., whether there can be a cyclic path starting from the location bound to some variable). A provably sound *abstract semantics* $\mathscr{C}_{\zeta}^{\tau}[\![\_]\!](\_)$ of a simple object-oriented language is developed, that works on $\mathscr{I}_{rc}^{\tau}$, and can often guarantee the acyclicity of *Directed Acyclic Graphs* (DAGs), which most likely will be considered as cyclic if only sharing, not reachability, is taken into account. The semantics has been implemented in the COSTA [3] COSt and Termination Analyzer as a component whose result is an essential information for proving the termination or inferring the resource usage of programs.

2 The abstract domain

The analysis works on the *reduced product* of two abstract domains. The first domain captures *may-reachability*, while the second deals with the *may-be-cyclic* property of variables. Let μ be a heap, $\ell \in\in \mathrm{dom}(\mu)$ be a location, $\mathscr{L}$ be the set of valid locations, and $\mu(\ell).\mathsf{frm}$ be the set of locations corresponding to the fields of the object located at ℓ The set of *reachable locations* from $\ell \in \mathrm{dom}(\mu)$ is $R(\mu,\ell){=}\cup\{R^i(\mu,\ell) \mid i \geq 0\}$, where $R^0(\mu,\ell){=}\mathrm{rng}(\mu(\ell).\mathsf{frm}) \cap \mathscr{L}$ (i.e., the locations reachable by directly accessing the fields of ℓ), and $R^{i+1}(\mu,\ell){=}\cup\{\mathrm{rng}(\mu(\ell').\mathsf{frm}) \cap \mathscr{L} \mid \ell' \in R^i(\mu,\ell)\}$ (the inductive case). The set of ε-reachable locations from $\ell \in \mathrm{dom}(\mu)$ is $R^\varepsilon(\mu,\ell){=}R(\mu,\ell) \cup \{\ell\}$. Note that ε-reachable locations include the source location ℓ itself, while reachable locations do not (unless ℓ is reachable from itself through a cycle). The rest of this section is developed in the context of a type environment τ which specifies the type of variables at a given program point. The set Σ_τ represents the states which are compatible with τ; every state contains a frame ϕ (a function from variables to locations) and a heap μ.

Reachability. Given a state $\sigma = (\phi,\mu) \in \Sigma_\tau$ containing a heap μ, a reference variable v is said to *reach* w in σ if $\phi(w) \in R(\mu,\phi(v))$. This means that, starting from v and applying *at least one dereference operation*, it is possible to reach the object to which w points. Due to strong typing, τ puts some restrictions on reachability; i.e., it might not be possible to have a heap where a variable of type κ_1 reaches one of type κ_2. Following [13], a class $\kappa_2 \in \mathscr{K}$ is said to be *reachable* from $\kappa_1 \in \mathscr{K}$ if there exists $(\phi,\mu) \in \Sigma_\tau$, and two locations $\ell,\ell' \in \mathrm{dom}(\mu)$ s.t. (a) $\mu(\ell).\mathsf{tag} = \kappa_1$ (where $\mu(\ell).\mathsf{tag}$ is the *class tag* of the object located at $\mu(\ell)$); (b) $\mu(\ell').\mathsf{tag} = \kappa_2$; and (c) $\ell' \in R(\mu,\ell)$. The reachability abstract domain is the complete lattice $\mathscr{I}_r^\tau = \langle \wp(\mathscr{R}^\tau), \subseteq, \emptyset, \mathscr{R}^\tau, \cap, \cup \rangle$ where $\mathscr{R}^\tau = \{v \rightsquigarrow w \mid v,w \in \mathrm{dom}(\tau)$ the class $\tau(w)$ is reachable from the class $\tau(v)\}$. The abstraction and concretization functions α_r^τ and γ_r^τ are defined in the standard way. *May-reach* information is described by *abstract values* $I_r \in \wp(\mathscr{R}^\tau)$. For example, $\{\mathsf{x}\rightsquigarrow\mathsf{z}, \mathsf{y}\rightsquigarrow\mathsf{z}\}$ describes those states where x and y *may* reach z. Note that a statement $\mathsf{x}\rightsquigarrow\mathsf{y}$ does not prevent x and y from aliasing; instead, x can reach y and alias with it at the same time, e.g., when x, y, and $\mathsf{x.f}$ point to the same location. The top element $\mathscr{R}^\tau$ is $\alpha_r^\tau(\Sigma_\tau)$, and represents all states which are compatible with τ. The bottom element $\emptyset$ models the set of all states where, for every two reference variables v and w (possibly the same variable), v does not reach w. Intuitively, reachability is a transitive property; i.e., if x reaches y and y reaches z, then x also reaches z. However, values in $\mathscr{I}_r^\tau$ are *not* closed by transitivity: e.g., it is possible to have $I_r = \{\mathsf{x}\rightsquigarrow\mathsf{y}, \mathsf{y}\rightsquigarrow\mathsf{z}\}$ which contains $\mathsf{x}\rightsquigarrow\mathsf{y}$ and $\mathsf{y}\rightsquigarrow\mathsf{z}$, but not $\mathsf{x}\rightsquigarrow\mathsf{z}$. Such abstract value is a reasonable one, and approximates, for example, the execution of "x=**new** C; y=**new** C; **if** (w>0) **then** x.f=y; **else** y.f=z;".

Cyclicity. Given a state $\sigma = (\phi,\mu) \in \Sigma_\tau$, a variable v is said to be *cyclic* in σ if there exists $\ell \in R^\varepsilon(\mu,\phi(v))$ such that $\ell \in R(\mu,\ell)$. In other words, v is cyclic if it reaches some memory location ℓ (which can possibly be $\phi(v)$ itself) through which a cyclic path goes. The notion of cyclic class is defined similarly to that of reachable classes [12]. The cyclicity domain is the dual of the non-cyclicity domain of [12]. The abstract domain for cyclicity is represented as the complete lattice $\mathscr{I}_c^\tau = \langle \wp(\mathscr{Y}^\tau), \subseteq, \emptyset, \mathscr{Y}^\tau, \cap, \cup \rangle$ where $\mathscr{Y}^\tau = \{\circlearrowleft^v \mid v \in \tau, \tau(v) \text{ is a cyclic class}\}$. *May-be-cyclic* information is described by *abstract values* $I_c \in \wp(\mathscr{Y}^\tau)$. E.g., $\{\circlearrowleft^{\mathsf{x}}\}$ represents states where no variable but x can be cyclic. The top element $\mathscr{Y}^\tau$ corresponds to Σ_τ; the bottom $\emptyset$ does not allow any variable to be cyclic.

The reduced product. As explained below, the abstract semantics uses reachability information in order to detect cycles, and cyclicity information in order to produce, in some cases, reachability information. Therefore, it makes sense to combine both kinds of information: in Abstract Interpretation, this amounts to computing the *reduced product* [6] of the corresponding abstract domains. In the present context, the reduced product can be computed by *reducing* the Cartesian product $\mathscr{I}^{\tau}_{rc} = \mathscr{I}^{\tau}_{r} \times \mathscr{I}^{\tau}_{c}$. Elements of $\mathscr{I}^{\tau}_{rc}$ are pairs $\langle I_r, I_c \rangle$, where I_r and I_c contain, respectively, the may-reach and the may-be-cyclic information. The abstraction and concretization functions are induced by those of $\mathscr{I}^{\tau}_{c}$ and $\mathscr{I}^{\tau}_{r}$:

$$\gamma^{\tau}_{rc}(\langle I_r, I_c \rangle) = \gamma^{\tau}_{r}(I_r) \cap \gamma^{\tau}_{c}(I_c) \qquad \alpha^{\tau}_{rc}(I) = \langle \alpha^{\tau}_{r}(I), \alpha^{\tau}_{c}(I) \rangle$$

However, it can happen that two elements of $\mathscr{I}^{\tau}_{rc}$ are mapped to the same concrete element, which prevents having a Galois insertion between $\mathscr{I}^{\tau}_{rc}$ and the concrete domain $\wp(\Sigma_\tau)$. Computing the reduced product deals exactly with this problem. In order to compute it, an equivalence relation $\equiv$ has to be defined, which satisfies $I^1_{rc} \equiv I^2_{rc}$ iff $\gamma^{\tau}_{rc}(I^1_{rc}) = \gamma^{\tau}_{rc}(I^2_{rc})$. Functions γ^{τ}_{rc} and α^{τ}_{rc} define a Galois insertion between $\mathscr{I}^{\tau}_{rc\equiv}$ and $\mathscr{I}^{\tau}_{\flat}$, where $\mathscr{I}^{\tau}_{rc\equiv}$ is $\mathscr{I}^{\tau}_{rc}$ equipped (reduced) with the equivalence relation. The equivalence relation can be based on the following observation: For every $I^1_r, I^2_r \in \mathscr{I}^{\tau}_{r}$ and $I^1_c, I^2_c \in \mathscr{I}^{\tau}_{c}$, the concretization $\gamma^{\tau}_{rc}(\langle I^1_r, I^1_c \rangle)$ is equal to $\gamma^{\tau}_{rc}(\langle I^2_r, I^2_c \rangle)$ if and only if both conditions hold: (a) $I^1_c = I^2_c$; and (b) $I^1_r \setminus \{v \rightsquigarrow v \mid \circlearrowleft^v \notin I^1_c\} = I^2_r \setminus \{v \rightsquigarrow v \mid \circlearrowleft^v \notin I^2_c\}$. This means that: (a) may-be-cyclic information always makes a difference as regards the set of concrete states; that is, adding a new statement $\circlearrowleft^v$ to $I_{rc} \in \mathscr{I}^{\tau}_{rc}$ results in representing a larger set of states; and (b) adding a pair $v \rightsquigarrow v$ to $I_{rc} \in \mathscr{I}^{\tau}_{rc}$, when v cannot be cyclic, does not make it represent more concrete states, since the acyclicity of v excludes that it can reach itself. From now on, $\mathscr{I}^{\tau}_{rc}$ will be a shorthand for $\mathscr{I}^{\tau}_{rc\equiv}$, where $\equiv$ is left implicit.

Denotational semantics. Abstract denotations for expressions and commands are depicted in Fig. 1. *Possible sharing*, *possible aliasing* and *purity* analysis are used as pre-existent components, i.e., programs are assumed to have been analyzed w.r.t. these properties. Two reference variables v and w *share* in $(\phi/heap)$ iff $R^{\varepsilon}(\mu, \phi(v)) \cap R^{\varepsilon}(\mu, \phi(w)) \neq \emptyset$; also, they *alias* if they point to the same location, namely, if $\phi(v) = \phi(w) \in \mathrm{dom}(\mu)$. The i-th argument of a method m is said to be *pure* if m does not update the data structure to which the argument initially points. For *sharing* and *purity*, the analysis described in [8] (based on [13]) is applied: with it, (1) it is possible to know if v might share with w at any program point (denoted by the pair $\langle v \bullet w \rangle$); and (2) for each method m, a denotation SH_{m} is given: for a set of pairs sh which safely describes the sharing between actual arguments in the input state, $sh' = \mathsf{SH}_{\mathsf{m}}(I)$ is such that (i) if $\langle v \bullet w \rangle \in sh'$, then v and w might share during the execution of m; and (ii) $\dot{v}_i \in sh'$ means that the i-th argument might be non-pure. As for *aliasing*, it is assumed that, at each program point, the pair $\langle v \cdot w \rangle$ tells if v and w can alias. Importantly, any non-null reference variable shares and aliases with itself; also, both are symmetric relations (i.e., $\langle v \bullet w \rangle$ iff $\langle w \bullet v \rangle$, and $\langle v \cdot w \rangle$ iff $\langle w \cdot v \rangle$). An abstract element $\langle I_r, I_c \rangle \in \mathscr{I}^{\tau}_{rc}$ will be represented by the set $I = I_r \cup I_c$; therefore, $v \rightsquigarrow w \in I$ and $\circlearrowleft^v \in I$ are shorthands for, resp., $v \rightsquigarrow w \in I_r$ and $\circlearrowleft^v \in I_c$. The operation $\exists v.I$ (*projection*) removes any statement about v from I, while $I[v/w]$ (*renaming*) v to w in I. For the sake of simplicity, class-reachability and class-cyclicity are taken into account *implicitly*: a new statement $v \rightsquigarrow w$ (resp., $\circlearrowleft^v$) is not added to an abstract state if $v \rightsquigarrow w \notin \mathscr{R}^{\tau}$ (resp., $\circlearrowleft^v \notin \mathscr{Y}^{\tau}$). The abstract semantics has been proven to be sound; i.e., (1) whenever v reaches w in a concrete state σ at a given program point, the statement $v \rightsquigarrow w$ is included in the abstract description I of σ at the same program point; and (2) whenever v is cyclic in σ, $\circlearrowleft^v$ must be present in I. For a detailed explanation of the abstract semantics and the proof of its correctness, the reader can refer to [9].

3 The analysis by examples

This section explains the behavior of the abstract semantics on a couple of interesting examples. The semantics instruments the program code with abstract values I_n, where n is the line number. A statement $v \rightsquigarrow w \in I_n$ means that v could reach w at line n, while $\circlearrowleft^v$ means that v could be cyclic.

Consider the class OrderedList depicted in Fig. 2. It implements an *ordered linked list* where head points to the first element, and lastInserted points to the last element which has been inserted. The class

$$
\begin{array}{lrl}
(1_e) & \mathscr{E}_\zeta^\tau[\![n]\!](I) = & \mathscr{E}_\zeta^\tau[\![\textbf{null}]\!](I) = \mathscr{E}_\zeta^\tau[\![\textbf{new}\ \kappa]\!](I) = I \\
(2_e) & \mathscr{E}_\zeta^\tau[\![v]\!](I) = & \text{if } \tau(v){=}\textbf{int} \text{ then } I \text{ else } I \cup I[v/\rho] \\
(3_e) & \mathscr{E}_\zeta^\tau[\![v.f]\!](I) = & \text{if } f \text{ has type } \textbf{int} \text{ then } I \text{ else } I \cup I' \text{ where} \\
 & & I'{=}I[v/\rho] \cup \{w \rightsquigarrow \rho \mid \langle w \bullet v \rangle\} \cup \{\rho \rightsquigarrow \rho \mid \circlearrowleft^v \in I\} \\
(4_e) & \mathscr{E}_\zeta^\tau[\![exp_1 \oplus exp_2]\!](I) = & \exists \rho . \mathscr{E}_\zeta^\tau[\![exp_2]\!](\exists \rho . \mathscr{E}_\zeta^\tau[\![exp_1]\!](I)) \\
(5_e) & \mathscr{E}_\zeta^\tau[\![v_0.m(v_1,..,v_n)]\!](I) = & \cup\{I, I_m, I_3, I_4\} \text{ such that}
\end{array}
$$

$$
\begin{array}{l}
\bar{v}{=}\{v_0,..,v_n\} \qquad I_0{=}\exists(\tau \setminus \bar{v}).I \\
I_m = \cup \ \{\ (\zeta(\mathsf{m})(I_0[\bar{v}/\mathsf{m}^i]))[\mathsf{m}^i/\bar{v}, out/\rho] \ \mid \mathsf{m} \text{ might be called here } \} \\
sh = \{\langle v_i \bullet v_j \rangle \mid v_i, v_j \in \bar{v} \text{ and } \langle v_i \bullet v_j \rangle\} \\
sh' = \cup\{\mathsf{SH}_\mathsf{m}(sh[\bar{v}/\mathsf{m}^i])[\mathsf{m}^i/\bar{v}, out/\rho] \mid \mathsf{m} \text{ might be called here}\} \\
I_1 = \{w_1 \rightsquigarrow w_2 \mid (v_i \rightsquigarrow v_j {\in} I_m) \wedge (\dot{v}_i {\in} sh') \wedge \langle w_1 \bullet v_i \rangle \wedge ((v_j \rightsquigarrow w_2 {\in} I) \vee \langle w_2 \cdot v_j \rangle)\} \\
I_2 = \{w_1 \rightsquigarrow w_2 \mid (\langle v_i \bullet v_j \rangle \in sh) \wedge (\dot{v}_i {\in} sh') \wedge \langle v_i \bullet w_1 \rangle \wedge (v_j \rightsquigarrow w_2 \in I)\} \\
I_3 = \cup\{(I_1 \cup I_2)[v/\rho] \mid \langle v \cdot \rho \rangle \text{ after the call } \} \\
I_4 = \{\circlearrowleft^w \mid \langle w \bullet v \rangle \wedge (\dot{v} {\in} sh') \wedge (\circlearrowleft^v \in I_m)\}
\end{array}
$$

$$
\begin{array}{lrl}
(1_c) & \mathscr{C}_\zeta^\tau[\![v{:=}exp]\!](I) = & (\exists v . \mathscr{E}_\zeta^\tau[\![exp]\!](I))[\rho/v] \\
(2_c) & \mathscr{C}_\zeta^\tau[\![v.f{:=}exp]\!](I) = & \exists \rho . (I' \cup I_r \cup I_c) \text{ where } I' = \mathscr{E}_\zeta^\tau[\![exp]\!](I) \text{ and}
\end{array}
$$

$$
\begin{array}{l}
I_r{=}\{w_1 \rightsquigarrow w_2 \mid ((\langle w_1 \cdot v \rangle \vee (w_1 \rightsquigarrow v \in I')) \wedge ((\langle \rho \cdot w_2 \rangle \vee (\rho \rightsquigarrow w_2 \in I'))\} \\
I_c{=}\{\circlearrowleft^w \mid ((\rho \rightsquigarrow v \in I') \vee \langle \rho \cdot v \rangle \vee (\circlearrowleft^\rho \in I')) \wedge ((\langle w \cdot v \rangle \vee (w \rightsquigarrow v \in I'))\}
\end{array}
$$

$$
\begin{array}{lrl}
(3_c) & \mathscr{C}_\zeta^\tau\left[\!\!\left[\begin{array}{ll} \textbf{if}\ exp & \textbf{then}\ com_1 \\ & \textbf{else}\ com_2 \end{array}\right]\!\!\right](I) = & \mathscr{C}_\zeta^\tau[\![com_1]\!](I) \cup \mathscr{C}_\zeta^\tau[\![com_2]\!](I) \\
(4_c) & \mathscr{C}_\zeta^\tau[\![\textbf{while}\ exp\ \textbf{do}\ com]\!](I) = & \xi(I) \text{ where } \xi = lfp(\lambda w.\lambda I.w(\mathscr{C}_\zeta^\tau[\![com]\!](I))) \\
(5_c) & \mathscr{C}_\zeta^\tau[\![\textbf{return}\ exp]\!](I) = & \mathscr{E}_\zeta^\tau[\![exp]\!](I)[\rho/out] \\
(6_c) & \mathscr{C}_\zeta^\tau[\![com_1; com_2]\!](I) = & \mathscr{C}_\zeta^\tau[\![com_2]\!](\mathscr{C}_\zeta^\tau[\![com_1]\!](I))
\end{array}
$$

Figure 1: Abstract denotations for expressions and commands

Node (not shown) implements a linked list in the usual way. The method insert adds a new element to the ordered list: it takes an integer i, creates a new node n for i, looks for its position, adds it to the list, and makes lastInserted point to the new node. Suppose insert appears inside a loop (for example, when inserting into the ordered list elements which are stored in an array). The goal is to infer that a call "x.insert(i)" never makes x cyclic. This is important since, if x cannot be proven to be acyclic after insert, then it must be assumed to be cyclic from the second iteration of the loop on. This, in turn, prevents from proving the termination of the loop at lines 10–12, since it might traverse a cycle. The challenge in this example is to prove that the instructions at lines 16 and 17 do not make any data structure cyclic. This is not trivial since **this**, p, and n share between each other at line 15; depending on how they share, the corresponding data structures might become cyclic or remain acyclic. Consider line 17: if there is a path (of length 0 or more) from n to p, then the data structures bound to them become cyclic, while they remain acyclic in any other case. The present analysis is able to infer that n and p share before line 17, but n does not reach p, which, in turn, guarantees that no data structure ever becomes cyclic (as evident from the absence of any cyclicity statement $\circlearrowleft^v$). Note that reachability information is essential for proving acyclicity, since the mere information that p and n share, without knowing how they do, requires to consider then as possibly cyclic, as done in [12].

As another example, consider the method mirror in Fig. 2, and suppose class Tree implements a binary tree in the standard way, with fields left and right. The call mirror(t) exchanges the values of left and right of each node in t. An initial state $\emptyset$ is transformed by mirror as follows. The first branch of the *if* (when t is **null**) does not change the initial denotation; on the other hand, when t is different from **null**, line 7 adds $t \rightsquigarrow l$; line 8 adds $t \rightsquigarrow r$; line 9 adds again $t \rightsquigarrow r$; and line 10 adds again $t \rightsquigarrow l$. Recursive calls mirror(l) and mirror(r) do not add any statement (since initially mirror ha a denotation

```
1  class OrderedList {
2   Node head, lastInserted;
3
4   void insert(int i) {
5     Node c,p,n;
6                           //I_6=∅
7     n:=new Node;          //I_7=∅
8     n.value:=i;           //I_8=∅
9     c:=this.head;         //I_9={this⇝c}
10    while (c!=null && c.value<i) do
11         p:=c;            //I_11={this⇝c,this⇝p}
12         c:=c.next;       //I_12={this⇝c,this⇝p,p⇝c}
13                          //I_13={this⇝c,this⇝p,p⇝c}
14    n.next:=c;            //I_14={this⇝c,this⇝p,p⇝c,n⇝c}
15    if (p=null) then
16      this.head:=n;  //I_16=I_15∪{this⇝n}
17    else p.next:=n;  //I_17=I_15∪{this⇝n,p⇝n}
18                               //I_18=I_16∪I_17=I_17
19    this.lastInserted:=n;  //I_19=I_18
20  }
21 }
```

```
1  void mirror(Tree t) {
2    Tree l,r;
3
4    if (t=null) then
5      return 0;
6    else
7      l:=t.left;
8      r:=t.right;
9      t.left:=r;
10     t.right:=l;
11     mirror(l);
12     mirror(r);
13 }
```

```
1  Node connect() {
2    Node c=this;
3
4    while (c.next!=null) do
5       c:=c.next;
6    c.next:=this;
7    return c;
8  }
```

Figure 2: The running example and the analysis results (in comments).

$\emptyset$) Projecting $\{t \rightsquigarrow l, t \rightsquigarrow r\}$ on t and *out* results in $\emptyset$, so that $\xi(\emptyset)$ does not change, and there is no need for another iteration. It can be concluded that, as expected, mirroring the tree does not make it cyclic.

Finally, consider the method connect, defined in the class Node. A call l.connect() with l acyclic makes the last element of l point to l, so that it becomes cyclic. It also returns a reference to the last element in the list. An initial state $\emptyset$ is transformed by connect as follows. Line 2 does not add any statements, while line 5 in the loop adds $this \rightsquigarrow curr$. Another iteration of the loop does not change anything, so that the loop is exited with $\{this \rightsquigarrow curr\}$. Since **this** is now reaching curr, line 6 adds $\{curr \rightsquigarrow this, curr \rightsquigarrow curr, this \rightsquigarrow this\}$, and $\{\circlearrowleft^{curr}, \circlearrowleft^{this}\}$. Finally, line 7 clones *curr* to *out*. In conclusion, the analysis correctly infers that l.connect() makes both l and the return value cyclic (i.e., statements $\circlearrowleft^{l}$ and $\circlearrowleft^{out}$ are produced).

The result of the abstract semantics for the above examples has been taken from the outcome of the implementation in the COSTA [3] COSt and Termination Analyzer on the Java bytecode version of these programs. The acyclicity analysis is a component of the system which is used in order to help to infer statically the termination of a program or information about its resource consumption. For example, suppose that, as pointed out above, insert is called inside a Java-style loop like

```
for (i=0; i<n; i++) { l.insert(a[i]); }
```

where the content of an array is copied to the ordered list. If l cannot be proven to be acyclic on exit from insert, then the list must be considered as possibly cyclic when entering the **for** loop the second time. This implies that the loop at line 10–12 in the code of insert could be non-terminating from the second iteration of the **for**. On the other hand, COSTA can prove that this code is terminating since the acyclicity analysis guarantees that the list is always acyclic, so that every iteration of the **for** loop terminates.

References

[1] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In *FMOODS*, LNCS 5051, pages 2–18, 2008.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *ESOP*, LNCS 4421, pages 157–172. Springer, 2007.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *FMCO'07*, number 5382 in LNCS, pages 113–133. Springer, 2008.

[4] J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Proc. CAV*, 2006.

[5] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.

[6] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *POPL'79*, pages 269–282. ACM, 1979.

[7] S. K. Debray and N. W. Lin. Cost analysis of logic programs. *ACM TOPLAS*, 15(5):826–875, November 1993.

[8] S. Genaim and F. Spoto. Constancy analysis. In *10th Workshop on Formal Techniques for Java-like Programs*, July 2008.

[9] S. Genaim and D. Zanardini. Automatic inference of acyclicity. Technical report, 2010. `http://costa.ls.fi.upm.es/papers/`.

[10] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *POPL*, pages 1–15, 1996.

[11] R. Jones and R. Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[12] S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In *VMCAI*, LNCS 3855. Springer, 2006.

[13] S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In *SAS*, number 3672 in LNCS, pages 320–335, 2005.

[14] F. Spoto, F. Mesnard, and É. Payet. A Termination Analyser for Java Bytecode based on Path-Length. *ACM TOPLAS*, 2010. To appear.

[15] B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9), 1975.

Higher Order Dependency Pairs With Argument Filterings

Cynthia Kop and Femke van Raamsdonk

Vrije Universiteit, Department of Theoretical Computer Science

Abstract. We present a termination method for left-linear Higher-order Rewrite Systems (HRSs) that are algebraic using a higher-order generalization of dependency pairs with argument filterings.

1 Introduction

An important method to (automatically) prove termination of first-order term rewriting is the dependency pair framework by Arts and Giesl [3]. This approach transforms a term rewriting system into a set of ordering constraints, to be satisfied by a well-founded ordering. This ordering, which is not required to be monotonic, can be further simplified using for example argument filterings.

Extending the dependency pair method to higher-order rewriting turns out to be difficult. A very natural extension is defined in [6], but the simplicity comes at a price: the dependency ordering must satisfy the *subterm property.* This property, which requires a term to be greater than its subterms, makes definition and use of argument filterings problematic. Moreover, well-foundedness of the relation is no longer equivalent to termination of the system.

Other extensions crucially rely on some restriction of the higher-order aspect, either by disallowing abstractions altogether (as is done in Simply Typed Term Rewriting Systems [1]) or by placing limitations on the rules; such restrictions are right-linear or non-nested [5], or plain-function-passing [4]. The present work continues on this line of research by restricting attention to *algebraic and left-linear HRSs.* Left-hand sides of algebraic HRSs only contain abstractions in a very simple form; right-hand sides are not restricted. This yields a natural class, which contains for instance functional programs in most common languages.

2 Algebraic Rules

We assume the reader is familiar with Nipkow's pattern HRSs. We will consider left-linear and *algebraic* HRSs, where left-hand sides have a basic form.

Definition 1. *A term is* simple *if it is either (the eta-long form of) a variable, or has the form $f(s_1, \ldots, s_n)$ with $f \in \mathcal{F}$ and all s_i simple. A rewrite rule $l \rightarrow r$ is* algebraic *if l is simple. An HRS is algebraic if all its rewrite rules are algebraic.*

Every simple term is a pattern, but not every pattern is a simple term. As examples, consider the non-simple terms $\lambda x.\, \mathtt{o}$, $\lambda x.\, \mathtt{s}(x)$, $\lambda x.\, Z$ and $X(\mathtt{o})$. The rules $\mathtt{map}(\lambda x.\, F(x), \mathtt{cons}(H,T)) \to \mathtt{cons}(F(H), \mathtt{map}(\lambda x.\, F(x), T))$ and $\mathtt{up}(X) \to \mathtt{map}(\lambda x.\, s(x), X)$ are algebraic; note that this notion only concerns the left-hand side of a rule.

Algebraic HRSs clearly form a restriction of the usual HRSs. One might compare them to simply typed term rewriting systems (STTRSs) [1,2], but unlike STTRSs, abstractions are permitted in terms. To note the difference, consider the algebraic system `LAMBDA` with function symbols $@ : o \Rightarrow o \Rightarrow o$ and $\Lambda : (o \Rightarrow o) \Rightarrow o$, and single rule $@(\Lambda(\lambda x.\, Z(x)), Y) \to Z(Y)$. In an STTRS this system would be terminating, as every rule makes the term shorter. As an HRS, this system implements untyped lambda-calculus, and is evidently not terminating!

All rewrite systems where the left-hand sides are η-equivalent to abstraction-free terms can be transformed into an algebraic system, without affecting termination. As this includes the HRS-representation of functional programs in any of the common languages, the restriction does not seem excessive.

The reason to consider algebraic HRSs is that in combination with left-linearity steps which take place below an abstraction can be postponed.

Definition 2. *A step $s \to t$ is* algebraic *if it does not occur inside an abstraction: either $s = l\gamma$, $t = r\gamma$, $l \to r \in \mathcal{R}$ or $s = f(s_1, \dots, s_i, \dots, s_n)$, $t = f(s_1, \dots, s_i', \dots, s_n)$ and $s_i \to s_i'$ by an algebraic step.*

Lemma 1. *In an algebraic and left-linear HRS non-algebraic steps can be postponed: if $s \to^* t$ there exists q such that $s \to^* q$ algebraicly and $q \to^* t$ non-algebraicly.*

From now on, we consider a fixed algebraic and left-linear HRS.

3 Dependency Pairs, Chains and Orderings

The basic idea of dependency pairs is to identify *minimal non-terminating* terms. To this end we combine the rewrite relation with a subterm relation for HRSs.

Definition 3. *The* subterm relation*, notation $\trianglerighteq$, is generated by the clauses:*

1. *$s \trianglerighteq s$,*
2. *$a(s_1, \dots, s_n) \trianglerighteq t$ if $s_i \trianglerighteq t$ for some i, for $a \in \mathcal{V} \cup \mathcal{F}$,*
3. *$\lambda \boldsymbol{x}.\, s \trianglerighteq t$ if $s[\boldsymbol{x} := \boldsymbol{c}] \trianglerighteq t$.*

Here, $c_\sigma : \sigma$ is a new symbol for each type σ. The substitution $[\boldsymbol{x} := \boldsymbol{c}]$ maps every variable $x_i : \sigma$ in $\{\boldsymbol{x}\}$ to c_σ.

The subterms of $f(\lambda x.\, X(x))$ are $f(\lambda x.\, X(x))$, $\lambda x.\, X(x)$, $X(c)$, and c. We have just a single constant c_σ for every type σ; this doesn't cause problems due to left-linearity. The relation $\trianglerighteq$ has all the nice properties of the normal subterm relation; in particular $\to_{\mathcal{R}} \cdot \trianglerighteq$ is terminating if and only if $\to_{\mathcal{R}}$ is terminating.

Let $\mathcal{F}^\#$ denote $\mathcal{F}$ extended with for every defined symbol f its marked version $f^\#$ with the same arity. Let $\mathcal{F}_c = \mathcal{F} \cup C$, where C contains the fresh constants c_σ. Let $\mathcal{F}_c^\# = \mathcal{F}^\# \cup C$. We define $s^\#$ as $f^\#(s_1, \dots, s_n)$ if $s = f(s_1, \dots, s_n)$ with f defined and $s^\# = s$ otherwise. Now we are ready to define dependency pairs.

Definition 4. *Given a rewrite rule $l \rightarrow r$, the pair of terms $l^\# \rightsquigarrow p^\#$ is a* dependency pair *for $l \rightarrow r$ if $p \trianglelefteq r$ and either p is headed by a defined symbol, or $p = X(s_1, \dots, s_n)$ with X a free variable in r and $n > 0$. The set of dependency pairs of $\mathcal{R}$ is denoted by* $\mathsf{DP}(\mathcal{R})$*, or by* DP *if $\mathcal{R}$ is clear from the context.*

For example, in the HRS $\{\mathtt{f}(\lambda x.\, Z(x)) \rightarrow Z(\mathtt{g}(\mathtt{a})), \mathtt{g}(x) \rightarrow \mathtt{h}(x)\}$, the dependency pairs are: $\mathtt{f}^\#(\lambda x.\, Z(x)) \rightsquigarrow Z(\mathtt{g}(\mathtt{a}))$ and $\mathtt{f}^\#(\lambda x.\, Z(x)) \rightsquigarrow \mathtt{g}^\#(\mathtt{a})$.

Dependency pairs find their use in the notion of a dependency chain: a sequence of dependency pairs with certain properties.

Definition 5. *A* dependency chain *for $\mathcal{R}$ is a sequence $[(l_i, p_i, t_i, \gamma_i) \mid i \in A]$ with either $A = \mathbb{N}$ or $A = \{1, \dots, N\}$ for some N such that, for all i:*

- *$l_i \rightsquigarrow p_i \in \mathsf{DP}(\mathcal{R})$,*
- *If p_i is headed by $f^\#$ for some f, then $t_i = p_i\gamma_i$,*
- *If p_i is headed by a variable, then there is q such that $t_i = q^\#$ and $p_i\gamma_i \trianglerighteq q$, but not $q \trianglelefteq \gamma_i(x)$ for any x, nor $q \trianglelefteq s\gamma_i$ for any $s \triangleleft p_i$,*
- *$t_i \rightarrow^*_{\mathcal{R},in} l_{i+1}\gamma_{i+1}$.*

The dependency chain is safe *if always $t_i \rightarrow^*_{\mathcal{R},in} l_{i+1}\gamma_{i+1}$ by only algebraic steps.*

Theorem 1. *An algebraic and left-linear HRS is terminating if it has no infinite safe dependency chain.*

The proof is straightforward; the *safe* condition holds because non-algebraic reductions can be postponed. Note that the condition in the theorem is sufficient, not required. There are terminating HRSs which admit an infinite (safe) dependency chain. This is because a non-terminating subterm may be beta-reduced away immediately.

To prove non-existence of a safe dependency chain we generate a set of inequalities from $\mathsf{DP}(\mathcal{R})$. If these inequalities can be satisfied by a so-called *quasi-monotonic ordering pair*, termination follows.

Definition 6. *A* quasi-monotonic ordering pair *is a pair $(>, \geq)$ of an ordering and a quasi-ordering, comparing base-type terms of equal types) such that:*

1. *$>$ and $\geq$* combine*: $> \cdot \geq$ is contained in $>$,*
2. *if s_i has base type and $s_i \geq s_i'$, then $f(s_1, \dots, s_i, \dots, s_n) \geq f(s_1, \dots, s_i', \dots, s_n)$*
3. *if s is simple then $s > t$ implies $s\gamma > t\gamma$ and $s \geq t$ implies $s\gamma \geq t\gamma$*

We say a quasi-monotonic ordering pair $(>, \geq)$ is well-founded if $>$ is well-founded. Comparing the definition for an ordering pair to similar definitions in the literature, requirements 2 and 3 are weakened versions from having the pair quasi-monotonic and closed under substitution.

Definition 7. *A* dependency ordering *for* DP *and* $\mathcal{R}$ *is a quasi-monotonic ordering pair* $(>, \geq)$ *satisfying the following inequalities:*

1. $l > p$ *for all dependency pairs* $l \rightsquigarrow p \in \mathsf{DP}$*,*
2. $l \geq r$ *for all rules* $l \rightarrow r \in \mathcal{R}$*,*
3. $l\gamma > t^{\#}$ *for all pairs* $l \rightsquigarrow p \in \mathsf{DP}$ *with* p *headed by a variable* X*, substitution* γ *and* $t \trianglelefteq p\gamma$ *such that* t *is headed by a defined symbol and neither* $t \trianglelefteq \gamma(X)$ *or* $t \trianglelefteq s\gamma$ *for any direct subterm* s *of* p*.*

The first two of these requirements are usual for dependency orderings; requirement 3 replaces the *subterm property* in [6]. Requirement 3 always holds if $\trianglerighteq$ is contained in $\geq$ (since then $l\gamma > p\gamma \geq t$), but there may be other ways.

Theorem 2. $\mathsf{DP}(\mathcal{R})$ *and* $\mathcal{R}$ *admit a well-founded dependency ordering if and only if there is no infinite safe dependency chain over* $\mathcal{R}$*.*

4 Argument Filterings

The idea behind argument filterings is to simplify the constraints on the rewrite rules and the dependency pairs by omitting some arguments of function symbols.

An argument filtering is defined as a partial function that maps a symbol $f : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_n \Rightarrow b$ in $\mathcal{F}^{\#}$ to either some $i \in \{1, \ldots, n\}$ (the collapsing case), or to a sequence $[i_1, \ldots, i_k]$ with $1 \leq i_1 < i_2 < \ldots < i_k \leq n$. In the collapsing case σ_i should have output-type b. Note that both unmarked and marked symbols are filtered, but the symbols c_σ, used to replace bound variables, are not.

Given an argument filtering A, let $\mathcal{F}_A$ contain all symbols in $\mathcal{F}^{\#}_c$, and additionally for every function symbol $f : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_n \Rightarrow b$ in $\mathcal{F}^{\#}$ with $A(f) = [i_1, \ldots, i_k]$ a fresh filtered function symbol $f_A : \sigma_{i_1} \Rightarrow \ldots \Rightarrow \sigma_{i_k} \Rightarrow b$. In the argument filtering of a term a symbol is not filtered if one of its arguments contains a bound variable (clause 2), and in the collapsing case the fresh constants c are used.

Definition 8. *Given an argument filtering* A*, the term filtering* $\mathtt{fil}$ *is defined as* $\mathtt{fil}(s) = \mathtt{fil}_{\emptyset}(s)$*, where the auxiliary mapping* $\mathtt{fil}_X$ *is defined by the clauses:*

1. $\mathtt{fil}_X(\lambda y.\, s) = \lambda y.\, \mathtt{fil}_{X \cup \{y\}}(s)$*,*
2. $\mathtt{fil}_X(a(s_1, \ldots, s_n)) = a(\mathtt{fil}_X(s_1), \ldots, \mathtt{fil}_X(s_n))$ *if* $a \notin \mathrm{Dom}(A)$ *or* $X \cap FV(a(s_1, \ldots, s_n)) \neq \emptyset$*,*
3. $\mathtt{fil}_X(a(s_1, \ldots, s_n)) = a^{\#}(\mathtt{fil}(s_{i_1}), \ldots, \mathtt{fil}(s_{i_k}))$ *if* $a \in \mathrm{Dom}(A)$*,* $X \cap FV(a(s_1, \ldots, s_n)) = \emptyset$ *and* $A(a) = [i_1, \ldots, i_k]$*,*
4. $\mathtt{fil}_X(a(s_1, \ldots, \lambda \boldsymbol{x}.\, s_i, \ldots, s_n)) = \mathtt{fil}_{\{\boldsymbol{x}\}}(s_i)[\boldsymbol{x} := \boldsymbol{c}]$ *if* $a \in \mathrm{Dom}(A)$*,* $X \cap FV(a(s_1, \ldots, s_n)) = \emptyset$ *and* $A(a) = i$ *(*$\boldsymbol{x}$ *may be empty).*

For example, let $\mathcal{F} = \{f : o \Rightarrow o \Rightarrow o, a : o\}$ and $A(f) = []$. Then $\mathtt{fil}_{\emptyset}(\lambda x.\, f(x, f(a, a))) = \lambda x.\, \mathtt{fil}_{\{x\}}(f(x, f(a, a))) = \lambda x.\, f(\mathtt{fil}_{\{x\}}(x), \mathtt{fil}_{\{x\}}(f(a, a))) = \lambda x.\, f(x, f_A)$. In a filtered term a symbol might both occur in normal and filtered form. The filtered and unfiltered symbols should be considered as different symbols.

To each ordering pair $(\succ, \succeq)$ over $\mathcal{T}(\mathcal{F}_A)$, we associate a pair $(>, \geq)$ over $\mathcal{T}(\mathcal{F}^{\#})$ in the natural way: $s > t$ iff $\mathtt{fil}(s) \succ \mathtt{fil}(t)$, $s \geq t$ iff $\mathtt{fil}(s) \succeq \mathtt{fil}(t)$.

Theorem 3. *Let $(\succ, \succeq)$ be a quasi-monotonic ordering pair on $\mathcal{T}(\mathcal{F}_A)$ which satisfies the following requirements:*

1. *$\succ$ contains the superterm relation $\rhd$ (but doesn't have to be monotonic),*
2. *always $f(x_1, \ldots, x_n) \succeq \mathtt{fil}(f(x_1, \ldots, x_n))$ if all x_i are (free) variables, and also $f(x_1, \ldots, x_n) \succeq \mathtt{fil}(f^{\#}(x_1, \ldots, x_n))$ for defined symbols f,*
3. *$\mathtt{fil}(l) \succ \mathtt{fil}(p)$ for all $l \rightsquigarrow p \in \mathsf{DP}(\mathcal{R})$,*
4. *$\mathtt{fil}(l) \succeq \mathtt{fil}(r)$ for all $l \rightarrow r \in \mathcal{R}$.*

Then the associated pair is a dependency filtering for $\mathcal{R}$; well-founded iff $\succ$ is.

That is, to prove termination of $\mathcal{R}$ it suffices to find a well-founded ordering pair over the filtered rules, satisfying the requirements in the theorem.

5 Concluding Remarks

In this paper we have defined a new generalization of dependency pairs with argument filterings for left-linear HRSs that are algebraic. algebraic. For left-linear first-order TRSs, our definition of dependency pairs coincides with the original one. Also, a higher-order generalization of argument filterings is given. The work has been done for HRSs, but the results are equally viable for different styles of higher-order rewriting, as long as the rules are presented in eta-long form. The results have been implemented in a tool, WANDA, which will participate in the termination competition of 2010.

For future work, we aim to further investigate dependency graphs and less standard filterings (for instance replacing a term $f(s_1, s_m)$ by $s_1 \cdot s_2$). We also intend to look into non-eta-normal algebraic systems, without transforming them.

References

1. T. Aoto and Y. Yamada. Dependency pairs for simply typed term rewriting. In J. Giesl, editor, *Proceedings of the 6th International Conference on Rewriting Techniques and Applications (RTA 2005)*, volume 3467 of *LNCS*, pages 120–134, Nara, Japan, April 2005. Springer Verlag.
2. T. Aoto and Y. Yamada. Argument filterings and usable rules for simply typed dependency pairs (extended abstract). In *Proceedings of the 4th International Workshop on Higher-Order Rewriting (HOR 2007)*, pages 21–27, Paris, France, June 2007. Workshop proceedings.
3. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
4. K. Kusakari, Y. Isogai, M. Sakai, and Blanqui. F. Static dependency pair method based on strong computability for higher-order rewrite systems. *IEICE Transactions on Information and Systems*, 92(10):2007–2015, 2009.
5. M. Sakai and K. Kusakari. On dependency pair method for proving termination of higher-order rewrite systems. *IEICE Transactions on Information and Systems*, E88-D(3):583–593, 2005.
6. Y. Sakai, M. Watanabe and T. Sakabe. An extension of the dependency pair method for proving termination of higher-order rewrite systems. *IEICE Transactions on Information and Systems*, E84-D(8):1025–1032, 2001.

Improvements on the "Size Change Termination Principle" in a functional language

Pierre Hyvernat, Christophe Raffalli
Université de Savoie, 73376 Le Bourget-du-Lac Cedex, France

Abstract

We present small improvements of Lee, Jones and Ben-Amram *size change termination principle* specialised to functional programming languages, where destructors come from pattern-matching. This allows for a finer analysis of call-graphs and yields a more precise test for termination. [1]

Introduction

Lee, Jones and Ben-Amram size change termination principle [1] is a simple yet surprisingly strong termination checker for generic programming languages. The ingredients are simply a well-founded order on values, a static analysis of the program to get a "call graph" of the recursive functions and a simple "transitive closure" computation on this graph.

The static analysis should yield a safe description of the potentials calls among the recursive function. The improvements described below were implemented on top of the PML programming language [2, 3], which due to its constraint checking algorithm is able to output a very detailed call-graph even when we use all the power of functional languages to try to hide the calls.

1 The size change principle

The central ingredients for the size-change termination principle are quite simple. Let G be a directed graph on vertices $F = \{f_1, f_2, ..., f_n\}$, with arcs labelled by elements of a *finite* monoid $(P, \circ)$. Let G^+ be the "transitive closure" of G in the following sense:

- G^+ has the same vertices as G,
- for all non-empty directed path $e_1, \dots, e_n$ from f to g in G, there is an edge from f to g in G^+ with label $l_1 \circ \cdots \circ l_n$, l_i being the label of e_i.

Note that because we impose that P is finite, G^+ is also finite.

Suppose now that G is the "call graph" of a set of mutually recursive definitions: vertices are function names and an edge from f to g describes a call to g in the definition of f. The elements of P can be used to describe the relationship between the *parameters* of the calling function f (variables) and the *arguments* of the called function g (terms). An element of P could, for example, tell us that "the second argument of g contains at least two constructors less than the third parameter of f".

With the appropriate monoid operation, the graph G^+ then describes the relation between arguments and parameters in iterated calls. If we have a well-founded order on terms, we have:

Theorem 1. *The following are equivalent [1]:*

- *all infinite directed path $e_1, \dots, e_n, \dots$ in G contain an infinitely decreasing thread,*[2]
- *G^+ contains an idempotent loop with label l (i.e. $l = l \circ l$) meaning "parameter i of f (as the called function) is smaller than argument i of f (as the calling function)", for some i.*

[1]Both authors want to thank Andreas Abel for the fruitful discussions that took place during his visit to Chambéry.

[2]where a thread is simply the trace of a parameter in the sequence of calls

It is interesting to note that this really is an equivalence. The proof is rather simple and uses a simple form of Ramsey’s theorem. What is crucial is the finiteness of the monoid describing size relations between parameters and arguments.

The magic of this theorem is that while the first condition may seem uncomputable, the second is rather easy to check.[3] If our programming language only allows non termination from infinite calls to recursive functions, we can thus deduce from the second condition that our recursive definitions are all terminating. Fortunately, PML is such a programming language (see [3]).

2 Detecting impossible cases

The original monoid used by Lee, Jones and Ben-Amram was very simple. It consisted in bipartite graphs that could also be seen as matrices where the coefficient i, j described the relationship between argument j of g and parameter i of f:

- “<” meaning “is strictly smaller”,
- “=” meaning “is smaller (possibly of the same size)”,
- “?” meaning “I don’t know, is possibly much bigger”.

Composition was then some kind of matrix multiplication with the obvious composition of coefficients.

While this already achieved quite a lot, this is not really sufficient in a functional programming language where analysis of pattern-matching can give us a lot of information. Consider the following, rather ad-hoc function:

```
val rec f x = match x with
    A[x] -> f B[x]
  | B[x] -> C[]
```

There is a single recursive call, but because of the variant constructors, this call cannot be composed with itself. This function would not pass the original test of [1].

We add information to our monoid in the following way: for a specific call to g from f we consider a list of trees describing the relations between arguments of g and parameters of f. Each tree will describe how an argument of g is built (using variants or tuples) from the different *parts* of the parameters of f. Each such *part* is obtained from a parameter of f by destructing it (projection or pattern-matching).

Definition 1 (full call-tree and full call-information)**.** *We define two sets $\mathscr{L}$ and $\mathscr{T}$ inductively. The set $\mathscr{L}$ corresponds to a sequence of destructors applied to a parameter of the calling function, or a term of unknown size (call to an external function for example):*

- $\texttt{?} \in \mathscr{L}$ *(“no information”),*
- $(i) \in \mathscr{L}$ *if* $i \in \mathbf{N}$ *(corresponding to the use of the i-th parameter),*
- $\texttt{-C}\,w \in \mathscr{L}$ *if* $w \in \mathscr{L}$ *(destruction of a variant: branch of a pattern-matching),*
- $\pi_i w \in \mathscr{L}$ *if* $w \in \mathscr{L}$ *(projection).*

The set $\mathscr{T}$ contains trees of constructors, with leafs containing elements of $\mathscr{L}$:

- $\mathscr{L} \subset \mathscr{T}$ *(leafs of the tree),*
- $\texttt{+C}\,t \in \mathscr{T}$ *if* $t \in \mathscr{T}$ *(construction of a variant),*
- $(t_1, \dots, t_n) \in \mathscr{T}$ *if* $\forall 1 \le i \le n, t_i \in \mathscr{T}$ *(construction of a tuple).*

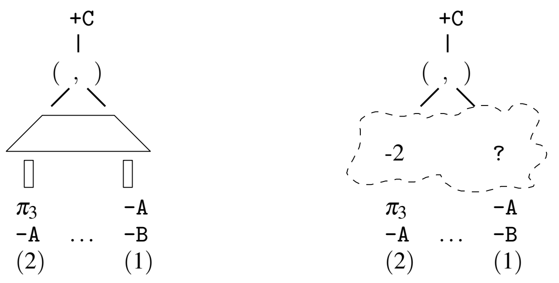

Figure 1: full tree and collapsed tree

A full call-information is simply given by a full call-tree for each argument of the called function.

Composition of full call-information is achieved by pasting appropriate trees on the leafs (*i.e.* pasting the i-th tree on parameter “(i)”) and “normalising” by matching consecutive destructors / constructors pairs. Whenever a variant destructor “`-A`” meets with a variant constructor “`+B`” we can abandon the composition: this is an impossible case. Typing should moreover ensure that a projection (record destructor) can never meet a variant constructor and vice and versa.

Composition could make the tree become larger and larger. To ensure finiteness, we need to collapse the trees to have a given *depth*. This should bound both the depth of the constructor part and the length of all the destructor parts. The missing information is summarised by a *size*: “<”, “=” or “?” as in the original principle. We actually use a segment of the integers, centered around 0. The original test is thus recovered using a depth of 0 and a size of 1. Of course, the two bounds (depth and size) can be increased to keep more information, at the expense of time and space consumption during the algorithm.

Definition 2 (bounded call-tree and call-information)**.** *The set $\mathscr{T}_{d,s}$ contains bounded trees of $\mathscr{T}$ with a middle “fuzzy” section. See the figure above for a graphical representation of such a tree. More formally, we replace the clause “(i)” by the following:*

- *if $-s \leq n \leq s$ then $(n,i) \in \mathscr{L}_{d,s}$.*

Moreover, if t is an element of this set:

- *the constructor tree of t is of depth at most d,*
- *the destructor sequences of t are of lenght at most d.*

We usually write ? for the size s, which represents an unbounded increase in size greater that s.

An element of $\mathscr{T}_{d,s}$ is obtained by collapsing the middle section of an element of $\mathscr{T}$ to a “fuzzy section” by counting the deficit destructors / constructors (this is actually subtle if one wants to keep as much information as possible...).

Composition of bounded call-information cannot be obtained by first composing in an “unbounded” manner and then collapsing, because the middle section contains unknown destructors / constructors. We thus need to match destructors of the first call-information with constructors of the second call-information. If constructors or destructors remain after this matching, they need to be incorporated in the middle section of the composition, except if we know the appropriate fuzzy section to be empty, in which case they can be incorporated to the constructor tree of the destructor sequences. To do this, we need an *empty* element distinct from 0, the former meaning no change and the later meaning a constant size.

[3]This problem is however P-space complete. It should be noted that in practise, this isn’t a problem...

3 Details on the notion of size

This monoid also allows for a more precise principle. When trying to compare the size of argument *i* and parameter *j* for a given call, we have to look at the branches in the *i*-th tree that reach parameter *j*. We only need to look at the best of these branches to guarantee termination. This allows us to treat pairs as "morally" distinct arguments, even when they are hidden beneath a variant constructor.

For example, this is what allows the test to tag the following (rather strange) function as terminating. This function computes the sum of a list of integers by keeping an accumulator in the head of the list:

```
val rec sum = fun
    | []  -> Z[]                      | [n] -> n
    | n::Z[]::l -> sum n::l           | n::S[k]::l -> sum S[n]::k::l
```

Note that typing isn't used in the algorithm, and the test doesn't distinguish between "`Z[]`" / "`S[]`" and the list constructors "`Nil[]`" / "`Cons[]`". The single full tree associated with the second recursive call is:

$$\texttt{+Cons}\,(\texttt{+S -Cons}\,\pi_1\,(1)\,,\ \texttt{+Cons}\,(\texttt{-Cons}\,\pi_2\,\texttt{-Cons}\,\pi_1\,\texttt{-S}\,(1)\,,\ \texttt{-Cons}\,\pi_2\,\texttt{-Cons}\,\pi_2\,(1)))$$

The middle branch also has a deficit of one variant constructor and will thus accounts for termination...

Note that when computing the deficit of size, we only count variant constructors. Record constructors / destructors do not serve any purpose there...

4 Building the initial call graph

PML computes the call graph from a set of typing constraints extracted from the program (see [3] for a formal description of the process). Given an uncountable set of formal *type names* $\mathscr{N}$, we annotate each subterm of a program with distinct type names. The only exception is for variables: all occurrences of the same variable have the same annotation. We write type names as Greek letter and annotation as upper-script. The constraints $\mathscr{C}$ are then collected as follows:

- functions: if $(\mathtt{f}^\alpha\,\mathtt{u}^\beta)^\gamma$ occurs in the program then $\alpha \subset (\beta \to \gamma) \in \mathscr{C}$ and if $(\mathtt{fun}\,x^\beta \to \mathtt{u}^\gamma)^\alpha$ occurs then $(\beta \to \gamma) \subset \alpha \in \mathscr{C}$.
- tuples: if $\pi_i(\mathtt{u}^\alpha)^\beta$ occurs then $\alpha \subset \pi_i\beta \in \mathscr{C}$ and if $(t_1^{\alpha_1}, \dots, t_n^{\alpha_n})^\beta$ occurs then $\alpha_1 \times \cdots \times \alpha_n \subset \beta \in \mathscr{C}$.
- variants: if $C[t^\alpha]^\beta$ occurs then $\mathtt{C}[\alpha] \subset \beta \in \mathscr{C}$ and if $(\mathtt{case}\,t^\beta\,\mathtt{of}\,\mathtt{C}_1[x^{\alpha_1}] \to u_1^{\gamma_1} \mid \cdots \mid \mathtt{C}_n[x^{\alpha_n}] \to u_n^{\gamma_n})^\gamma$ then $\{\beta \subset C_1[\alpha_1] + \cdots + C_n[\alpha_n], \gamma_1 \subset \gamma, \dots, \gamma_n \subset \gamma\} \subset \mathscr{C}$.

Those formal constraints are a syntactical version of inclusions between semantical "types".

These extracted constraints are saturated by "deduction", which correspond to computing an approximation of the constraints obtained after reduction of the term. A well-foundedness test is then performed to ensure that loop can only occur in recursive definitions. This is described in [3].

Finally, by traversing the resulting constraints, we can construct the call graph associated to our program. There are basically four steps: collecting the parameters (the function abstraction), collecting the arguments to each calls (the function applications), building the destructor parts of the calls and finally the constructors parts.

This approach allows us to deal with tuples containing functions, tests before function abstractions and other non-standard ways of writing functions and calling them. Those are not frequent but still possible in functional languages, as in the following example which is treated as two separate mutually recursive functions by PML:

```
  val rec f x =
    (fun S[y] -> (f y).2 x x | Z[] -> Z[]),
    (fun y -> (fun S[z] -> (f z).1 y | Z[] -> Z[]))
```

5 Examples and further work

The first example shows what we can gain from the original size-change termination principle:

```
val rec f a b =
  match a,b with
      Node[ Node[a1 , _] , _]  ,  b -> f Node[a1,b] Node[a,b]
    | _,_ -> A[]
```

What is new in this example is that even thought both arguments may increase in global size, the left branch of the first argument decreases. If the depth bound is 2, the corresponding tree will be: +Node (-Node π_1 $(-1,1),(0,2)$), where the "-1" will account for the decreasing branch...

The second example illustrates how the extraction from the typing constraints allow to detect termination when using external function calls in recursive calls. This example is an implementation of subtraction and modulo on unary natural numbers:

```
val pred x = match x with
    S[x] -> x

val rec sub x y = match x, y with
  _, Z[] -> x
| Z[], _ -> raise Undef[]
| _ -> sub (pred x) (pred y)
```

```
val rec mod_aux acc x y' =
  try   let x' = (sub (pred x) y') in
        mod_aux S[acc] x' y'
  with  Undef[] -> acc

val mod x y = match y with
    Z[] -> raise Undef[]
  | S[y'] -> mod_aux Z[] x y'
```

In this example, the call-information for subtraction is -S(1) for argument 1 and -S(2) for argument 2. This means that we detected that the function pred removes a S on each arguments. The call information for the mod$_a$ux function is +S(1), -S(2) and (3), which means that we detected that the composition of pred and sub remove one successor. However, replacing (sub (pred x) y') by sub x S[y'] would fails, because we do not detect that the subtraction will remove at least one successor. Work is being done in that direction...

References

[1] Chin Soon Lee, Neil D. Jones and Amir M. Ben-Amram, *The Size-Change Principle for Program Termination*, ACM SIGPLAN Notices, Volume 36, Issue 3, 2001.

[2] Christophe Raffalli, *The* PML *programming language*, homepage of the project: http://www.lama.univ-savoie.fr/tracpml.

[3] Christophe Raffalli, *Realizability for programming languages*, course notes for the "École jeunes chercheurs du GDR IM", 2010. (hal-00474043), http://www.lama.univ-savoie.fr/~raffalli/pdfs/ejc.pdf.

Match-Bounds for Relative Termination

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

Johannes Waldmann
HTWK Leipzig
Leipzig, Germany
waldmann@imn.htwk-leipzig.de

Abstract

Match-bounds can be used to prove relative termination. We compare a recent such method (Zankl and Korp, RTA 2010) with earlier work (Waldmann, JALC 2007). We recall that matchbounds are related to interpretations in the fuzzy semi-ring (with operations min, max) that can be found by constraint solving. The order in the semi-ring (with zero at the top) is such that it supports the closure construction required in the RFC method.

1 Introduction

The underlying idea of the match-bound method for proving termination of rewriting systems is to annotate symbols by natural numbers, so-called *heights*. During a derivation, new heights are assigned to symbols in the reduct of a rewrite step depending on the heights occurring in the corresponding redex. For string rewriting, for instance, reduct symbols get height $h+1$, where h is the minimal height of a redex symbol. A derivation is said to be *match-bounded* by b if, starting with height 0 throughout, no height greater than b is reached. Accordingly, a rewriting system is match-bounded by b in case all possible derivations are, and it is said to be just match-bounded if such a bounding number exists. It is known that match-bounded rewriting systems are terminating, and that they have linearly bounded derivational complexity.

In order to prove match-boundedness, weighted automata are employed as certificates provided that certain (local) compatibility properties are fulfilled. Those certificates can be construced by an appropriate completion procedure. This method is complete in the sense that for any match-bounded rewriting system a certifying automaton exists, which in turn can be found by completion. Variants of this method are known that enable producing huge certificates, i. e., automata with many states efficiently. A different approach is to use constraint-solvers in order to find certificates.

The match-bound method was first published for string rewriting in 2003 ([3, 4]). As a prominent example, termination of the one-rule system $\{aabb \to bbbaaa\}$ (which is problem `SRS/Zantema/z001` in [10]) could be proven terminating automatically for the first time. Later, the approach was extended to term rewriting ([5, 7]). While for proofs of termination the match-bound method became largely obsolet after introducing matrix interpretations ([6, 2]), revived interest in the topic is due to recent efforts to find proof methods entailing (low) derivational complexity.

Therefore, this paper first recalls a seemingly little-known result on match-bounds for relative termination. With the help of constraint solvers, we can use it to prove match-boundedness of some rewriting systems from the termination problem data base [10] where completion methods fail (due to resource exhaustion). We then connect match-bounds for relative termination to the RFC-method (see [3, Section 7]). Although in this paper considerations are limited to string rewriting, most of the results can be extended to term rewriting along the lines of [5, 7].

2 Notation and Preliminaries

We write $u \to_{p,\ell\to r} v$ if string u rewrites to string v by applying the rule $\ell \to r$ at position p in u. A derivation is a (finite or infinite) sequence $u_0, u_1, \ldots$ with $u_i \to_R u_{i+1}$. Let R be a string rewriting system over an alphabet Σ, and let $\mathrm{lhs}(R)$ and $\mathrm{rhs}(R)$ denote the set of left and right hand sides resp. of its rules. For $\varepsilon \notin \mathrm{lhs}(R)$ (where ε is the empty string) we denote by $\mathrm{match}(R)$ the (infinite) system over $\Sigma \times \mathbb{N}$

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

where the morphisms $\mathrm{lift}_c : \Sigma^* \to (\Sigma \times \mathbb{N})^*$, $\mathrm{base} : (\Sigma \times \mathbb{N})^* \to \Sigma^*$ and $\mathrm{height} : (\Sigma \times \mathbb{N})^* \to \mathbb{N}^*$ are defined for $x \in \Sigma$ and $h \in \mathbb{N}$ by $\mathrm{lift}_h : x \mapsto (x,h)$, $\mathrm{base} : (x,h) \mapsto x$ and $\mathrm{height} : (x,h) \mapsto x$.

For an R-derivation D, let $\mathrm{match}(D)$ be the corresponding $\mathrm{match}(R)$-derivation, defined as follows: If D starts with u, then $\mathrm{match}(D)$ starts with $\mathrm{lift}_0(u)$.

If D contains a step $u_i \to_{p_i,\ell_i \to r_i} u_{i+1}$, then $\mathrm{match}(D)$ contains the step $u'_i \to_{p_i,\ell'_i \to r'_i} u'_{i+1}$ where $\ell'_i \to r'_i$ is the unique rule in $\mathrm{match}(R)$ with $\mathrm{base}(\ell'_i) = \ell_i$ and $\mathrm{base}(r'_i) = r_i$ that matches at position p_i.

3 Matchboundedness and Interpretations

A finite automaton A over Σ is said to be compatible with a rewriting system R over Σ if for each rule $\ell \to r$ in R and each path $p \xrightarrow{\ell}_A q$, there is a path $p \xrightarrow{r}_A q$. A rewriting system R over Σ is match-bounded by c for $L \subseteq \Sigma^*$ if there is an automaton over $\Sigma \times \{0,1,\ldots,c\}$ that accepts $\mathrm{lift}_0(\Sigma^*)$ and that is compatible with $\mathrm{match}_{c+1}(R)$.

A more liberal definition can be given (that turns out to be equivalent). Let $\mathbb{F}$ be the *fuzzy semiring* $\mathbb{Z} \cup \{-\infty, +\infty\}$ with the minimum operation as semiring addition and the maximum operation as semiring multiplication. Then, $+\infty$ is the zero element and $-\infty$ is the unit element of the semiring. We use the natural order $>$ on $\mathbb{F}$ and define $x >_1 y$ as $x = +\infty = y \vee x > y$.

A *fuzzy matrix interpretation* is a mapping from Σ to square matrices over $\mathbb{F}$ of identical dimension. An interpretation $[\cdot]$ can be extended from Σ to Σ^* homomorphically (by matrix multiplication). An interpretation $[\cdot]$ is called *compatible* with a rewriting system R if $[l] >_1 [r]$ for all rules $(l \to r) \in R$, where $>_1$ on matrices is the point-wise extension of $>_1$ on elements. The *support* of an interpretation $[\cdot]$ is the set of strings w such that $[w]$ contains at least one non-zero entry.

Proposition 1. *If R admits a compatible fuzzy interpretation $[\cdot]$, then R is match-bounded for its support. The support is closed w.r.t. $\to_R$.*

Example 2. The fuzzy matrix interpretation for $\Sigma = \{a,b,c,d\}$

$$a \mapsto \begin{pmatrix} 3 & 0 & \infty \\ \infty & \infty & \infty \\ 2 & 0 & \infty \end{pmatrix},\ b \mapsto \begin{pmatrix} 2 & 3 & 1 \\ \infty & \infty & 1 \\ \infty & 3 & 1 \end{pmatrix},\ c \mapsto \begin{pmatrix} 0 & 0 & 3 \\ \infty & 0 & 6 \\ \infty & 0 & \infty \end{pmatrix},\ d \mapsto \begin{pmatrix} 4 & 4 & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \end{pmatrix}$$

is compatible with $\{bca \to ababc, b \to cc, cd \to abca, aa \to acba\}$ (SRS/Zantema/z003). Since the top left entry of $[x]$ is $< \infty$ for each $x \in \Sigma$, and this property is kept during multiplication, the support of this interpretation is Σ^*. So the interpretation proves that R is match-bounded (by 4). This interpretation was found (quickly) via SAT encoding. Note that the match-bound certificate automaton obtained by completion has 217 states (but is found quickly as well).

4 Relative Termination

A rewriting system R is *terminating relative* to a rewriting system S if any $R \cup S$-derivation contains only finitely many R-steps (i. e., $\to_S^* \circ \to_R \circ \to_S^*$ is terminating), denoted by $\mathrm{SN}(R/S)$. For instance, $\{aa \to aba\}$ is terminating relative to $\{b \to bb\}$. Note that $\mathrm{SN}(R/S)$ and $\mathrm{SN}(S)$ imply $\mathrm{SN}(R \cup S)$.

Definition 3. A pair (R,S) of string rewriting systems over Σ is called match-bounded by $b \in \mathbb{N}$ for $L \subseteq \Sigma^*$, if for each $(R \cup S)$-derivation D that starts with some string $u \in L$ and each rule $(\ell' \to r') \in \text{match}(R)$ that is applied in $\text{match}(D)$, we have $\min(\text{height}(\ell')) < b$.

Theorem 4. *If (R,S) is match-bounded, then R is terminating relative to S.*

Proof. (Sketch) Assign to each string in an $(R \cup S)$-derivation the multiset of its heights that are less than b. Then each R-step strictly decreases this interpretation under the induced multiset ordering. □

A morphism $h : \Sigma \to \Delta$ is extended to a string rewriting system R over alphabet Σ by applying h to each of its rules, i. e., we obtain $h(R) = \{h(\ell) \to h(r) \mid (\ell \to r) \in R\}$ over alphabet Δ.

Definition 5. For $c \in \mathbb{N}$ let cut_c denote the morphism from $\Sigma \times \mathbb{N}$ to $\Sigma \times \mathbb{N}$ given by

$$(x,h) \mapsto (x, \min\{c,h\}).$$

Proposition 6. *For rewriting systems R and S over Σ and a language $L \subseteq \Sigma^*$, if there exists an automaton over $\Sigma \times \{0,1,\ldots,c\}$ that accepts $\text{lift}_0(L)$, is compatible with $\text{match}(R)$, and is compatible with $\text{cut}_c(\text{match}(S))$, then (R,S) is match-bounded by c for L.*

Note that this assumption implies that no rule from $\text{match}(R)$ can be applied such that the labels in its left hand side are all $\geq c$. A certificate automaton according to this proposition cannot be constructed by exact completion since $\text{cut}_c(S)$ is not deleting.

We can formulate the idea in the language of fuzzy interpretations as follows, where $-\infty$ now plays a special role (corresponding to "cutting at the bound"). We use the order $>_2$ on $\mathbb{F}$ where $x >_2 y$ iff $x >_1 y \vee x = -\infty = y$. A fuzzy matrix interpretation $[\cdot]$ is called *weakly compatible* with a rewriting system S if $[l] \geq_2 [r]$ for all rules $(l \to r) \in R$ where $\geq_2$ on matrices is the point-wise extension of $>_2$.

{prop:rela-

Proposition 7. *If a fuzzy interpretation $[\cdot]$ is compatible with R and weakly compatible with S, then R/S is terminating on the support of $[\cdot]$.*

This allows us to use the match-bound method in modular termination proofs.

Proposition 8. *If both (R,S) and S are match-bounded, and $\text{rhs}(R)$ and $\text{lhs}(S)$ are overlap-free, then $R \cup S$ is match-bounded.*

Note that the non-overlapping condition cannot be dropped, see Example 15.

Example 9. As an application, we can prove match-boundedness of `SRS/Zantema06/17` from [10] using the partition $R = \{aua \to ubu \mid u \in X\}$ with $X = \{\varepsilon, b, bb, bbb\}$ and $S = \{aua \to ubu \mid u \in Y\}$ with $Y = \{a, aa, ab, ba, aaa, aab, aba, abb, baa, bab, bba\}$.

5 Forward Closures and Relative Termination

For a rewriting system R over Σ, let $\text{RFC}(R)$ denote the set of right-hand sides of R-forward-closures.

We can compute $\text{RFC}(R)$ from $(R \cup R_\#)^*(\text{rhs}(R)\#^*)$, where $R_\# = \{u\# \to r \mid (uv \to r) \in R,\ u,v \neq \varepsilon\}$ over $\Sigma \cup \{\#\}$, where # is a fresh letter.

It is known that R is terminating on Σ^* iff R is terminating on $\text{RFC}(R)$, and this statement can be extended:

Proposition 10. *If R/S is terminating on $\text{RFC}(R \cup S)$, then R/S is terminating.*

This proposition allows to "remove rules" (namely, R). The remaining termination proof then can start with $\text{RFC}(S)$, which is a smaller language.

We need termination on $\text{RFC}(R \cup S)$ only. This is a *local termination* problem and would require the construction of a (partial) model [1]. Here we use matchbounds, and Proposition 7 gives a closure property.

We define (R,S) to be *RFC-matchbounded by b* if $(R, S \cup R_\# \cup S_\#)$ is match-bounded by b for $\text{rhs}(R \cup S)\#^*$.

Proposition 11. *If (R,S) is RFC-matchbounded by b, then R/S is terminating.*

Example 12. For $R = \{cb \to bbc\}$ and $S = \{ab \to baa\}$ we have $\text{RFC}(R \cup S)\#^* \subseteq \{a,b\}^* \cdot \{c, \varepsilon\} \cdot \#^* = L$, since L is closed w.r.t. $R \cup R_\# \cup S \cup S_\#$. In fact, L contains no R-redex, so (R,S) is RFC-matchbounded by 0, therefore R/S is terminating and we can infer termination of $R \cup S$ from termination of S. For that, we consider $\text{reverse}(S) = \{ba \to aab\}$ with $\text{RFC}(S)\#^* \subseteq a^*b\#^*$, again without a S-redex, and this match-bounded by 0. Note that neither (R,S) nor (S,R) is match-bounded since both R and S alone have exponential derivational complexity.

This method had been applied by Matchbox in the 2005 competition, using approximate automata completion. We propose to revive that idea, using fuzzy interpretations, found by constraint solving.

6 An Alternate Definition of Relative Matchboundedness

Zankl and Korp [11] suggest a notion of relative match-boundedness for term rewriting: for relative rules that are non-size-increasing, and have all lhs labels equal, the rhs gets the same label. For string rewriting, this is:

Definition 13.

$$\text{match}'(R) = \left\{ l' \to \text{lift}_{h+d}(r) \mid \begin{array}{l} (\text{base}(l') \to r) \in R, h = \min \text{height}(l'), \\ d = \text{if } |l'| \le |r'| \wedge h = \max \text{height}\, l' \text{ then } 0 \text{ else } 1 \end{array} \right\}$$

Definition 14. A pair (R,S) is called match'-bounded by b, if each $(\text{match}(R) \cup \text{match}'(S))$-derivation is bounded by b.

The definition by Zankl and Korp is incomparable to ours:

They study term rewriting whereas we restrict ourselves to string rewriting. We can handle size-increasing rules, and use our definition of relative match-bounds to prove termination of systems where match'-bounds are not applicable, see Example 15. On the other hand, for non-size-increasing rules, the approach of Zankl and Korp is strictly more powerful, see Example 16.

{ex:comp:2}

Example 15. For $R = \{cL \to R\}$ and $S = \{Ra \to bbR, R \to Ld, bL \to Laa\}$, (R,S) is match-bounded by 1 since each R redex contains a letter c, and this gets height 0 since c does not occur in right-hand sides. Also, S is match-bounded by 2 (certified by an automaton with 15 states). Hence, both R/S and S are terminating, so $R \cup S$ is terminating as well. Since all rules in S are length-increasing, (R,S) is match'-bounded iff $R \cup S$ is match-bounded. But its derivational complexity is too large for that: $R \cup S$ allows for derivations of exponential length. They can be combined from $Ra^k \to_S^k b^{2k}R \to_S b^{2k}Ld \to_S^{2k} La^{4k}d$ and $cRa^k \to_S^* cLa^{4k}d \to_R Ra^{4k}d$, so $c^iRa \to^* Ra^{4^i}d^i$.

{ex:comp:3}

Example 16. For $R = \{a \to b\}$ and $S = \{a \to a\}$, (R,S) is match'-bounded by 1 since all heights of a are 0 (they are not changed by the S rule), and heigths of b are ≤ 1. On the other hand, (R,S) is not match-bounded since for any k, we have a mixed derivation $a \to_S^k a \to_R b$, resulting in an application of the R rule at height k.

References

[1] Jörg Endrullis, Roel C. de Vrijer, and Johannes Waldmann. Local termination. In Ralf Treinen, editor, *RTA*, volume 5595 of *Lecture Notes in Computer Science*, pages 270–284. Springer, 2009.

[2] Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of term rewriting. *J. Autom. Reasoning*, 40(2-3):195–220, 2008.

[3] Alfons Geser, Dieter Hofbauer, and Johannes Waldmann. Match-bounded string rewriting systems. In Rovan and Vojtás [9], pages 449–459.

[4] Alfons Geser, Dieter Hofbauer, and Johannes Waldmann. Match-bounded string rewriting systems. *Appl. Algebra Eng. Commun. Comput.*, 15(3-4):149–171, 2004.

[5] Alfons Geser, Dieter Hofbauer, Johannes Waldmann, and Hans Zantema. On tree automata that certify termination of left-linear term rewriting systems. *Inf. Comput.*, 205(4):512–534, 2007.

[6] Dieter Hofbauer and Johannes Waldmann. Termination of string rewriting with matrix interpretations. In Pfenning [8], pages 328–342.

[7] Martin Korp and Aart Middeldorp. Match-bounds revisited. *Inf. Comput.*, 207(11):1259–1283, 2009.

[8] Frank Pfenning, editor. *Term Rewriting and Applications, 17th International Conference, RTA 2006, Seattle, WA, USA, August 12-14, 2006, Proceedings*, volume 4098 of *Lecture Notes in Computer Science*. Springer, 2006.

[9] Branislav Rovan and Peter Vojtás, editors. *Mathematical Foundations of Computer Science 2003, 28th International Symposium, MFCS 2003, Bratislava, Slovakia, August 25-29, 2003, Proceedings*, volume 2747 of *Lecture Notes in Computer Science*. Springer, 2003.

[10] The termination problems data base. `www.termination-portal.org/wiki/TPDB`.

[11] Harald Zankl and Martin Korp. Modular complexity analysis via relative complexity. to appear in Proc. RTA, `http://cl-informatik.uibk.ac.at/users/hzankl/new/publications/ZA10_03.pdf`, 2010.