Fourth International Workshop on Termination

WST '99

Dagstuhl, Germany

May 10-12, 1999

Abstracts

Fourth International Workshop on Termination
WST '99

Dagstuhl, Germany

May 10-12, 1999

Abstracts

Program Committee

Adam Cichon (Nancy, France)
Nachum Dershowitz (Urbana, USA)
Jürgen Giesl (Darmstadt, Germany)
Pierre Lescanne (Lyon, France)
Ursula Martin (St. Andrews, Scotland)
Hans Zantema (Utrecht, The Netherlands)

Contents

The termination hierarchy for term rewriting

Hans Zantema
Utrecht University, The Netherlands
e-mail: hansz@cs.uu.nl

(Tutorial talk)

A natural way to prove termination of a term rewriting system is to find an interpretation in an algebra equipped with a well-founded order, in such a way that for every rule the interpretation of the left hand side is greater than that of the right hand side. If the operations are strictly monotone in all arguments then indeed termination of the TRS can be concluded; the algebra is then called *well-founded monotone*. The requirement that every left hand side exceeds the corresponding right hand side is called *compatibility*. In [1] a hierarchy of termination was proposed based on the carrier set and the operations in this algebra:

- a TRS is terminating if it is compatible with a well-founded monotone algebra;
- a TRS is simply terminating if it is compatible with a well-founded monotone algebra satisfying $f_A(\ldots, a, \ldots) \geq a$ for all operations f_A and all positions in these operations;
- a TRS is totally terminating if it is compatible with a well-founded monotone algebra in which the order is total;
- a TRS is ω-terminating if the interpretation is in the natural numbers;
- a TRS is polynomially terminating if the interpretation is in the natural numbers and the operations are polynomials.

We restrict to finite signatures. Where alternative definitions of these notions appeared before, they are equivalent to ours except for total termination of string rewriting. The following implications hold:

polynomial termination
$\Longrightarrow$ ω-termination
$\Longrightarrow$ total termination
$\Longrightarrow$ simple termination
$\Longrightarrow$ termination.

It is possible to extend this hierarchy by some extra levels and some ramifications. For instance, termination by recursive path order always implies

ω-termination, and weak normalization and non-loopingness can be seen as levels beyond termination.

None of the implications in the hierarchy holds in the reverse direction: for every implication $A \Rightarrow B$ even a single string rewrite rule can be given satisfying B but not A. Moreover, deciding these properties is hard: in [2] it has been shown that for most implications $A \Rightarrow B$ the question whether A holds for a TRS satisfying B is undecidable. This even holds for single rules ([3]).

More recent we considered some modifications on the notion of well-founded monotone algebra. One original requirement is that all operations f_A are strictly monotone in all arguments. This requirement may be weakened to the combination of weak monotonicity and the condition that $f_A(\ldots, a, \ldots) \geq a$ for all operations f_A and all positions in these operations. It can be shown that compatibility with this kind of an algebra still implies termination, while much more operations are allowed to be chosen. For instance, the operation choosing the maximum of two natural numbers is not strictly monotone in both arguments, but satisfies the new condition. It is a natural question whether this modification on the requirements on the operations affects the various notions in the termination hierarchy. Indeed it does. For instance, the single rewrite rule $f(g((x)) \rightarrow g(f(f(x)))$ is not ω-terminating in the old sense, but it allows an interpretation in the natural numbers with the weakened requirements.

Another subtility is the difference between the requirement $f_A(\ldots, a, \ldots) \geq a$ and $f_A(\ldots, a, \ldots) > a$. Without referring to the algebras it can be stated as the question whether adding the rules $f(\ldots, x, \ldots) \rightarrow x$ affects the termination behaviour or not. For termination, simple termination and total termination it is easily shown that adding these rules have no effect. Surprisingly however, ω-termination and polynomial termination may be destroyed by adding these rules. For instance, the TRS consisting of the two rules $f(f(x)) \rightarrow g(f(x))$ and $g(g(x)) \rightarrow f(g(h(x)))$ is ω-terminating, but after adding the single rule $h(x) \rightarrow x$ it is not ω-terminating.

References

[1] Zantema, H. Termination of term rewriting: interpretation and type elimination. *Journal of Symbolic Computation 17* (1994), 23–50.

[2] Geser, A., Middeldorp, A., Ohlebusch, E., and Zantema, H. Relative undecidability in term rewriting. In *Proceedings of the Conference of the European Association of Computer Science Logic (CSL96)* (1997), D. van Dalen, Ed., Lecture Notes in Computer Science, Springer Verlag.

[3] Geser, A., Middeldorp, A., Ohlebusch, E., and Zantema, H. Relative undecidability in the termination hierarchy of single rewrite rules. In *Proceedings Theory and Practice of Software Development (TAPSOFT97, CAAP/FASE)* (1997), M. Bidoit and M. Dauchet, Eds., vol. 1214 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 237–248.

Termination Semantics of Logic Programs with Cut and Related Features

Jamie Andrews
Dept. of Computer Science
Univ. of Western Ontario
London, Ontario, Canada

There are various varieties of termination for logic programs. I am particularly interested in the termination of logic programs which use practical features such as the Prolog "cut". In order to prove termination of such programs, it is necessary first to define a semantics for them. In my talk, I will discuss the various varieties of termination and the semantics proposed for characterizing them, leading to semantics of Prolog with cut which facilitates proofs of termination properties.

Varieties of Termination. The first, and most obvious, dimension in which we can classify logic programming termination is the result which is given upon termination. Goals can either succeed (returning one or more substitutions), fail (returning nothing), or diverge. There is also clearly only one way for goals to fail: the search tree must be finite, and all the leaf nodes of that tree must be failure nodes. Beyond that, we can identify several different senses in which a goal can succeed.

A goal can *existentially terminate*, which means that either it fails or at least one leaf node in the search tree is a success node. Alternatively, it can *universally* or *strongly terminate*, which means that the search tree is finite (every leaf node necessarily being a success or failure node). Clearly, the notion of existential termination is useful only when discussing breadth-first search strategies, whereas if a goal strongly terminates then we know that Prolog (for instance) can find a solution.

However, in fact neither notion characterizes what we might see as the natural concept of termination in Prolog. Prolog uses a depth-first, left-to-right search strategy, and can return a solution iff it finds one before the first infinite branch in the tree during its traversal. The user can then request further solutions on backtracking, but the computation of the query has succeeded at least once. Hence an insistence on strong termination disallows goals which, on backtracking, return a finite number of solutions and then diverge, or return an infinite number of solutions. Strong termination is "strong" in the sense of insisting on strong conditions for termination.

Most work on termination of logic programs has concentrated on strong termination. However, since the depth-first, left-to-right search strategy is used by most logic programming systems, the problem of characterizing this strategy is also important.

The Prolog Cut. Characterizing depth-first, left-to-right termination is especially important in the context of programs which use the Prolog "cut" or related features. These features cut away part of the search tree. Which part is cut depends on what is the current node in the depth-first traversal.

The cut (!) has been a feature of Prolog implementations for over fifteen years. When used as a subgoal in a clause body, it has the effect of cutting away all alternative clauses to the current clause, and (in the case of "hard" cut) all backtrack points encountered since the processing of the current clause started. Most of the uses of cut in typical Prolog programs have the effect of if-then-else expressions; we can therefore replace clauses which use cut for this purpose with clauses which use negation as failure. Termination of goals with respect to these programs can then be characterized by some semantics of programs with negation as failure under depth-first termination, such as the author's [And97], or strong termination, such as Stärk's [Stä98].

However, cut is also used for other purposes in practical logic programs. It is used when the programmer knows that all solutions being returned by a previous subgoal will have the same effect as the first solution. It is also used for efficiency, to cut away any backtrack points left over from a previous subgoal, which might otherwise occupy memory. Finally, it is often used to select the first solution returned by the previous subgoals in the clause. It is for this latter use that it is important to characterize depth-first termination, since cut selects the first solution returned by this strategy.

Characterizing Cut. Given the apparently non-logical nature of the Prolog cut, it is interesting that we can characterize a large subset of the programs using it with a semantics similar to that used to logically characterize depth-first Prolog with negation as failure [And97]. We can adapt the preliminary technique reported on in [And95] as follows.

We begin with a generalization of simple Prolog programs with cut, and an operational semantics of this generalization. We then show that we can transform any such program into a "completed" form which has the same behaviour but in which the predicates have only one clause each; a new "if-then" construct is introduced for this purpose. We then place a mode restriction on the completed program similar to Stärk's for Prolog with negation as failure [Stä98]. The mode restriction disallows the most egregiously non-logical Prolog programs which use cut, but still allows the first-solution and efficiency behaviours which hard cut gives us.

We are then able to characterize the mode-restricted, completed programs with a style of semantics referred to as "unfolding-normal-form-valuation" (UNV) semantics in [And97]. Given a goal G and a program P, we consider all unfoldings G' of G with respect to P. We put these unfoldings in a normal form, and apply a valuation (compositional function from goals to truth values) to the normal forms. The program valuation v_P is then defined as the function which returns the "most true" of these truth values, where "true" is more true than "undefined", which is more true than "false". The UNV semantics characterizes the operational semantics in the sense that a goal G succeeds (fails, diverges) with respect to a program P in the operational semantics iff $v_P(G)$ is "true" ("false", "undefined"). Since the operational semantics uses the depth-first, left-to-right strategy, we can claim to have abstractly characterized depth-first termination with respect to this version of cut.

References

[And95] James H. Andrews. A paralogical semantics for the Prolog cut. In *Proceedings of the International Logic Programming Symposium*, Portland, December 1995. MIT Press.

[And97] James H. Andrews. A logical semantics for depth-first Prolog with ground negation. *Theoretical Computer Science*, 184(1-2):105–143, September 1997.

[Stä98] Robert Stärk. The theoretical foundations of LPTP (a logic program theorem prover). *Journal of Logic Programming*, 36(3):241–269, 1998.

TERMINATION OF SIMPLY MODED WELL-TYPED LOGIC PROGRAMS UNDER TABLED EXECUTION MECHANISM

Sofie Verbaeten and Danny De Schreye
Department of Computer Science, K.U.Leuven,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium.
{sofie.verbaeten, danny.deschreye}@cs.kuleuven.ac.be

Tabled logic programming [2, 4, 12, 14] is receiving increasing attention in the Logic Programming community. It avoids many of the shortcomings of SLD(NF) execution [10] and provides a more flexible and often extremely efficient execution mechanism for logic programs. In particular, tabled execution of logic programs terminates more often than execution based on SLD-resolution. So, if a program can be proven to terminate under SLD-resolution (by one of the existing automated techniques surveyed in [5]), then the program will trivially also terminate under SLG-resolution, the resolution principle of tabulation [4]. But, since there are SLG-terminating programs which are not SLD-terminating, better proof techniques can be found.

The ideas underlying *tabling* are very simple. Essentially, under tabled execution mechanism, answers for selected atoms are stored in a table. When a variant of such an atom is recursively called, the selected atom is not resolved against program clauses, instead, all corresponding answers computed so far are looked up in the table and the corresponding answer substitutions are applied to the atom. This process is repeated for all subsequent computed answer substitutions that correspond to the atom.

We study *universal* termination of definite tabled logic programs executed under SLG using a fixed left-to-right selection rule (we drop the "S" in SLD and SLG whenever we refer to the left-to-right selection rule). We refer to the full paper [13] for the formal definitions, theorems and motivating examples. Note that, whenever SLD-computation from an initial goal leads to infinitely many different, non-variant calls, the SLG-computation from this initial goal will also be infinite. This leads to the first basic notion of termination under tabled execution mechanism: *quasi-termination* (this term is borrowed from [9], defining a similar notion in the context of termination of off-line partial evaluation of functional programs). For a program P and a set of queries Q, let $Call(P,Q)$ denote the set of atoms (modulo variant relation) which occur as selected atom in an LD-derivation of some query in Q. This set is also called the call set of the program P w.r.t. Q. With this notation, a program P is said to quasi-terminate w.r.t. a set of queries S iff for all $A \in S$, $\sharp Call(P, \{A\}) < \infty$. Even when tabling, quasi-termination only partially corresponds to our intuitive notion of a "terminating computation". This is because an atom can have infinitely many computed answers (note that this does not have to lead to infinitely many new calls). Therefore, the stronger notion of *LG-termination* is introduced: P LG-terminates w.r.t. S iff P quasi-terminates w.r.t. S and for all $A \in S$ the set of all computed answers for atoms in $Call(P, \{A\})$ is finite.

One of the few approaches studying termination of tabled logic programs was developed by Decorte et al [8]. They present necessary and sufficient conditions for quasi-termination and LG-termination. The main differences with characterizations (of e.g.[6]) of LD-terminating programs and queries are:

- level mappings are assumed to be *finitely partitioning* on the call set, i.e. only finitely many atoms of the call set can be mapped to the same natural number,
- decreases of the level mapping are *not* assumed to be *strict* (but are imposed on all body atoms).

Starting from the necessary and sufficient conditions of [8], we introduce sufficient, easy to *automatize* conditions for quasi-termination and LG-termination. We show how our termination conditions can be automatized by extending the recently developed constraint-based automatic termination analysis for SLD-resolution by Decorte and De Schreye [7]. In order to automatically prove termination under SLG-resolution, our termination conditions:

- reason fully at the *clause level* (and not on "calls" as in [8]),
- give a *syntactical* condition on a level mapping (we will consider variants of semi-linear level mappings, see [3]) to be *finitely partitioning*.

In order to be able to reason fully at the clause level, most automatic termination analyses (like [7]) require that the level mapping is *rigid* on the call set. If a level mapping is rigid on the call set, the atoms in the call set can be considered as ground w.r.t. the level mapping (their value is invariant under substitutions). In this way, the problem of backpropagation of bindings in the calls is dealt with, and the conditions can be stated at the clause level. It turned out that a straightforward combination of the notions of rigid level mapping (as it appears in automatic termination analyses) and finitely partitioning level mapping (as it appears in termination analyses of tabled logic programs) offers some difficulties (being rigid and being finitely partitioning are in a way contradictory requirements). But, having some mode and type information at our disposal, we are able to solve these problems.
For *simply moded, well-moded* programs and queries (see for instance [1]), we show how to combine these two requirements on level mappings. Namely, for a simply moded, well-moded program P and set of queries S, a level mapping is rigid and finitely partitioning on $Call(P,S)$ if it measures all and only input positions in $Call(P,S)$. The condition to measure all and only input positions in a set of atoms is a syntactical condition (on the kind of level mappings we consider, namely variants of semi-linear level mappings [3]). In the case of quasi-termination, this leads to one of the main results in the paper, namely:

Proposition 1 *Let P be a simply moded, well-moded program and S be a set of simply moded, well-moded queries. Let $|.|$ be a level mapping which measures all and only the input positions in $Call(P,S)$, such that*

- *for any clause $H \leftarrow B_1, \ldots, B_n$ in P,*
- *for any atom B_i, $i \in \{1,..,n\}$,*
- *for any correct answer substitution ψ for $\leftarrow B_1, \ldots, B_{i-1}$,*
  such that $(H\psi)^{In}$, $(B_1\psi)^{In}, \ldots, (B_i\psi)^{In} \in Call(P,S)^{In}$,

$$|H\psi| \geq |B_i\psi|.$$

Then P quasi-terminates w.r.t. S.

Where the notation $A^{In} \in Call(P,S)^{In}$ stands for A is input-correct w.r.t. $Call(P,S)$, i.e. there is an atom $A^{'} \in Call(P,S)$ such that the input positions of A and $A^{'}$ are equal (modulo variable renaming). In the case of LG-termination, we obtain a similar result. However, in this case, the conditions are not on the program P, but on the program P^a, which is obtained by applying the answer-transformation on P. We refer to [13] for more details.
In order to deal with non-ground input, we present also a solution for *simply moded, well-typed* [1] programs and queries, which generalizes our result for simply moded, well-moded programs and queries. Instead of trying to combine the rigidity requirement on the level mapping with the requirement to be finitely partitioning, we abandon the rigidity requirement and provide syntactical conditions on the program and set of queries which allow us to reason fully at the clause level in the termination condition. Namely, these syntactical conditions will ensure that the input arguments of the queries get never instantiated during a derivation. The conditions are based on concepts introduced in [1]. The results for quasi-termination and LG-termination are of a similar nature as Proposition 1.
Besides the work of [8], there are only relatively few works studying termination under tabled execution mechanism: in [11], in the context of well-moded programs, a sufficient condition is given for the bounded term-size property, which implies LG-termination; [9] provides another sufficient condition for quasi-termination in the context of functional programming.
Finally, the study of termination of *normal* logic programs under tabled execution mechanism is an interesting topic for future research.

References

[1] K. R. Apt and S. Etalle. On the unification free prolog programs. In A. Borzyszkowski and S. Sokolowski, editors, *Proceedings of the Conference on Mathematical Foundations of Computer Science (MFCS 93)*, pages 1–19. Lecture Notes in Computer Science, Springer-Verlag, 1993.

[2] R. Bol and L. Degerstedt. The underlying search for magic templates and tabulation. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 793–811, Budapest, Hungary, june 1993. The MIT Press.

[3] A. Bossi, N. Cocco, and M. Fabris. Norms on terms and their use in proving universal termination of a logic program. *Theoretical Computer Science*, 124(2):297–328, februari 1994.

[4] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, january 1996.

[5] D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, 19 & 20:199–260, may/july 1994.

[6] D. De Schreye, K. Verschaetse, and M. Bruynooghe. A framework for analysing the termination of definite logic programs with respect to call patterns. In *Proc. FGCS'92*, pages 481–488, ICOT Tokyo, 1992. ICOT.

[7] S. Decorte and D. De Schreye. Demand-driven and constraint-based automatic termination analysis for logic programs. In L. Naish, editor, *Proc. 14th International Conference on Logic Programming*, pages 78–92, Leuven, Belgium, july 1997.

[8] S. Decorte, D. De Schreye, M. Leuschel, B. Martens, and K. Sagonas. Termination analysis for tabled logic programs. In *Proceedings of LOPSTR'97: Logic Program Synthesis and Transformation*, Leuven, Belgium, july 1997.

[9] C. K. Holst. Finiteness Analysis. In John Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA)*, number 523 in LNCS, pages 473–495. Springer-Verlag, august 1991.

[10] J.W. Lloyd. *Foundations of logic programming.* Springer-Verlag, 1987.

[11] L. Plümer. *Termination proofs for logic programs.* Number 446 in LNAI. Springer-Verlag, 1990.

[12] H. Tamaki and T. Sato. OLD Resolution with Tabulation. In *Proceedings ICLP'86*, Lecture Notes in Computer Science 225, pages 84–98. Springer Verlag, 1986.

[13] S. Verbaeten and D. De Schreye. Termination analysis of tabled logic programs using mode and type information. Technical Report 277, Department of Computer Science, K.U.Leuven, 1999.

[14] L. Vieille. Recursive query processing: the power of logic. *Theoretical computer science*, 69(1):1–53, 1989.

Well-Terminating, Input-Driven Logic Programs

Jan–Georg Smaus

University of Kent, Canterbury, CT2 7NF, United Kingdom,
phone +44/1227/827553, fax +44/1227/762811,
j.g.smaus@ukc.ac.uk

I will present what might be called the "second-most abstract" approach to termination of logic programs [6].

Termination is a problem in logic programming as much as in any other paradigm. For any paradigm, termination proofs rely on measuring the size of a state in the computation and showing that this size decreases during the computation. Measuring the size of a computation state usually involves measuring the size of the *data*, and this data originates from the *input* of the program.

To be more specific, let us look at functional programming and the well-known function $append :: (list, list) \rightarrow list$.[1] Consider the input $([1, 2], [])$. Then the computation is as follows:

$$append([1, 2], []) \rightsquigarrow [1|append([2], [])] \rightsquigarrow [1, 2|append([], [])] \rightsquigarrow [1, 2].$$

At each step, the length of the list in the first argument of *append* decreases.

This introduction points us to a major if not the main problem for termination in logic programming: a priori, there is no notion of *input*. Corresponding to the above function would be a predicate *append* which has three arguments. Here are some computations for the predicate *append*:

$$append([1, 2], [], Zs) \rightsquigarrow append([1], [], Zs') \rightsquigarrow append([], [], Zs'') \rightsquigarrow \epsilon.$$
$$append(Xs, Ys, [1, 2]) \rightsquigarrow append(Xs', Ys, [2]) \rightsquigarrow append(Xs'', Ys, []) \rightsquigarrow \epsilon.$$
$$append(Xs, Ys, Zs) \rightsquigarrow append(Xs', Ys, Zs') \rightsquigarrow append(Xs'', Ys, Zs'') \ldots$$

To prove termination of logic programs, it is useful to introduce a notion of input by declaring certain argument positions to be *input positions*.

To explain the effect of such a declaration, we must first explain what "evaluation strategy" means for logic programs. A computation state in a logic program is a *query*, which is a sequence of atoms. For example, $append([1, 2], [], Zs)$ or $append([1, 2], [], As), append(As, [], Bs)$ are queries. At each computation step, one atom in the query is selected and resolved with one clause in the program. The "evaluation strategy" (in logic programming terminology: the *selection rule*) states which atom is selected in each step.

[1] I avoid Currying to keep the notation similar to logic programming.

The most abstract approach is making no assumptions about the selection rule at all [2]. However, very few programs terminate for all selection rules. The most common selection rule is the rule that always selects the leftmost atom in a query.

My assumption about the selection rule is that an atom is selected for resolution only when it is instantiated to such a degree that its input arguments do not become instantiated any further in this resolution step. This is a minimal assumption if we want to give a natural meaning to "input".

We will see that many predicates terminate under such a weak assumption. Although implicitly present in previous work, my assumption has never been formulated in such an abstract way. Other authors have usually made stronger assumptions [1, 3, 4, 5].

The technical core of my work is establishing a dependency between the atoms of a query of the kind: the first atom cannot produce an infinite computation; the second atom cannot produce an infinite computation unless the first does; and so forth.

I believe that my approach is well suited for presentation at a general workshop because it will point out the similarities and differences between termination in logic programming and other paradigms.

References

[1] K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In *Proceedings of AMAST'95*, LNCS, Berlin, 1995. Springer-Verlag. Invited Lecture.

[2] M. Bezem. Strong termination of logic programs. *Journal of Logic Programming*, 15(1 & 2):79–97, 1993.

[3] S. Lüttringhaus-Kappel. Control generation for logic programs. In D. S. Warren, editor, *Proceedings of the 10th International Conference on Logic Programming*, pages 478–495. MIT Press, 1993.

[4] E. Marchiori and F. Teusink. Proving termination of logic programs with delay declarations. In J. W. Lloyd, editor, *Proceedings of the 12th International Logic Programming Symposium*, pages 447–461. MIT Press, 1995.

[5] L. Naish. Coroutining and the construction of terminating logic programs. Technical Report 92/5, University of Melbourne, 1992.

[6] J.-G. Smaus. Well-terminating, input-driven logic programs. Technical Report 16-98, University of Kent at Canterbury, 1998.

Transforming Context-Sensitive Rewrite Systems

Aart Middeldorp
University of Tsukuba

joint work with

Jürgen Giesl
Darmstadt University of Technology

Context-sensitive rewriting (Lucas [1, 2]) is a variant of term rewriting in which for every function symbol one indicates which arguments may be evaluated and a redex contraction is allowed only if it is does not take place in a forbidden argument of a function symbol above it. In this talk we are concerned with the problem of showing termination of context-sensitive rewriting. More precisely, we consider transformations from context-sensitive rewrite systems to ordinary term rewrite systems that are *sound* with respect to termination: termination of the transformed term rewrite system implies termination of the original context-sensitive rewrite system. The advantage of such an approach is that all techniques for proving termination of term rewriting can be used to infer termination of context-sensitive rewriting. Two such transformations are reported in the literature, by Lucas [1] and by Zantema [3]. We add two more. Our first transformation is simple, its soundness is easily established, and it improves upon the transformations of [1, 3]. To be precise, we prove that the class of terminating context-sensitive rewrite systems for which our transformation succeeds is larger than that of Lucas' transformation and we claim that the same holds for Zantema's transformation. None of these three transformations succeeds in transforming every terminating context-sensitive rewrite system into a terminating term rewrite system. In other words, they all lack *completeness*. We analyze the failure of completeness for our first transformation, resulting in a second transformation with is both sound and complete. This latter result can be interpreted as stating that from a termination perspective there is no reason to study context-sensitive rewriting.

References

[1] S. Lucas. Termination of context-sensitive rewriting by rewriting. In *Proceedings of the 23rd International Colloquium on Automata, Languages and Programming*, volume 1099 of *LNCS*, pages 122–133, 1996.

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

[3] H. Zantema. Termination of context-sensitive rewriting. In *Proceedings of the 8th International Conference on Rewriting Techniques and Applications*, volume 1232 of *LNCS*, pages 172–186, 1997.

Context-Sensitive AC-rewriting

M. C. F. Ferreira A. L. Ribeiro
Dep. de Informática, Fac. de Ciências e Tecnologia, Univ. Nova de Lisboa,
2825-114 Caparica, Portugal, {cf,ar}@di.fct.unl.pt,
tel: +351 - 1 - 294 85 36, fax: +351 - 1 - 294 85 41

Context-sensitive rewriting was introduced in [1] and consists of syntactical restrictions imposed on a Term Rewriting System indicating how reductions *can* be performed; this is achieved by associating to each function symbol a set of positions, namely the positions where reductions can be performed. So *context-sensitive rewriting* is a restriction of the usual rewrite relation which reduces the reduction space and allows for a finer control of the reductions of a term and is related to lazy evaluation strategies in functional languages.

We extended the concept of context-sensitive rewriting to rewriting modulo an AC-theory by restricting both rewrite steps and AC-steps, and we studied the termination properties of context-sensitive AC-rewriting following the approach of Zantema [2] for context-sensitive rewriting. Context-sensitive (AC) termination, or μ-termination is a more general property than (AC) termination since this last property implies the former but not vice-versa, thus context-sensitive relations can also be used to study termination properties of reduction strategies for systems that are not terminating. We present two techniques for proving μ-AC-termination: the first method provides a complete technique and consists in the definition of a suitable interpretation algebra for the (restricted) AC-rewrite system and the context-sensitive AC-rewrite relation, then μ-AC-termination of a system R/AC can be concluded from the existence of such a suitable interpretation. The second method consists in transforming the (restricted) AC-rewrite system and the context-sensitive relation in an non-restricted AC-rewrite system where the reduction relation considered is the usual AC-rewrite relation; μ-AC-termination of the original system can be inferred from AC-termination of the transformed system. The advantage of this method over the previous one is that usual well-known techniques for proving AC-termination can be used to infer μ-AC-termination.

References

[1] S. Lucas. Fundamentals of context-sensitive rewriting. In J. S. M. Bartŏsek and J. Wiedermann, editors, *Proceedings of the XXII Seminar on Current Trends in Theory and Practice of Informatics, SOFSEM'95*, volume 1012 of *Lecture Notes in Computer Science*, pages 405–412. Springer, 1995.

[2] H. Zantema. Termination of context-sensitive rewriting. In *Proceddings of the 8th International Conference on Rewriting Techniques and Applications, RTA'97*, Lecture Notes in Computer Science. Springer, 1997.

A fully syntactic AC-RPO*

Albert Rubio

Universitat Politècnica de Catalunya, Barcelona, Spain.
E-mail: `rubio@lsi.upc.es`

Rewrite-based methods with built-in associativity and commutativity (AC) properties for some of the operators are well-known to be crucial in theorem proving and programming. Therefore a lot of work has been done on the development of suitable *AC-compatible* reduction or simplification orderings. An essential additional property of the ordering that is needed in order to preserve the completeness of most rewrite-based theorem proving techniques (modulo AC) is *AC-totality*, i.e. the totality on (AC-different) ground terms.

Since the initial attempts, it has always been an aim to obtain AC-compatible versions of Dershowitz' *recursive path ordering* [Der82], as it is simple, easy to automatize and use, and normally orients the rules in an adequate direction. In [RN95] we gave the first RPO-based AC-total and AC-compatible reduction ordering without any restriction on the number of AC-symbols or on the precedence over the signature. Unfortunately, although being defined in terms of RPO, it does not behave like RPO; e.g. it does not orient the distributivity rule in the "right" (i.e. distributing) way, since a transformation on the terms is applied before using RPO (this approach, with different transformations, is also used in [BP85] among others). Therefore, a better approach seems to be to directly apply an RPO-like scheme, treating as the only special case the *AC-equal-top case*, that is, when both terms to be compared are headed by the same AC-symbol. In this direction the first AC-compatible simplification ordering with an RPO scheme was defined in [KSZ90] and the first one AC-total on ground terms in [KS97]. Other simpler proposals for AC-orderings with RPO scheme were given in [Rub97] and in [KS98].

However, all these AC-orderings need to interpret terms (apart from *flattening*) in some way, which makes their behaviour less intuitive, unlike it happens with the standard RPO, whose simple fully syntactic definition has been an important reason for its success.

We propose the first fully syntactic AC-RPO, i.e., no interpretation is needed apart from flattening. It is very simple, and hence easy to implement, and its behaviour is intuitive as for the standard RPO. The ordering is AC-total, and defined uniformly for both ground and non-ground terms, as well as for partial precedences.

Moreover, precisely due to the fact that it is not interpretation-based, it is the first AC-RPO that can deal *incrementally* with partial precedences, i.e. if $s \succ t$, then $s \succ t$ under any extension of the precedence. This aspect is essential, together with its intuitive behaviour, for interactive applications like Knuth-Bendix completion. Of course, previously existing orderings could work with

* Partially supported by the ESPRIT working group CCL-II, ref. WG # 22457.

partial precedences, but in a useless way, simply by considering an arbitrarily chosen total extension of the partial precedence, and hence loosing incrementality.

In order to introduce the concepts smoothly we present the ordering in three steps, first for ground terms and total precedences, then for terms with variables and total precedences and finally for terms with variables and partial precedences, each definition strictly extending the previous one. For this reason we prove all properties only for the last one, showing that it is indeed an AC-compatible simplification ordering.

All this results are reported in [Rub98].

References

[BP85] L. Bachmair and D. A. Plaisted. Termination orderings for associative-commutative rewriting systems. *Journal of Symbolic Computation*, 1:329–349, 1985.

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

[KS97] D. Kapur and G. Sivakumar. A total, ground path ordering for proving termination of ac-rewrite systems. In H. Comon, editor, *8th International Conference on Rewriting Techniques and Applications*, LNCS, 1997.

[KS98] D. Kapur and G. Sivakumar. A recursive path ordering for proving associative-commutative termination, 1998.

[KSZ90] D. Kapur, G. Sivakumar, and H. Zhang. A new method for proving termination of ac-rewrite systems. In *Conf. Found. of Software Technology and Theoretical Computer Science*, LNCS 472, pages 134–148, New Delhi, India, December 1990. Springer-Verlag.

[RN95] Albert Rubio and Robert Nieuwenhuis. A total AC-compatible ordering based on RPO. *Theoretical Computer Science*, 142(2):209–227, May 15, 1995.

[Rub97] Albert Rubio. A total AC-compatible ordering with RPO scheme. Technical Report UPC-LSI-97, Univ. Polit. Catalunya, September 1997.

[Rub98] Albert Rubio. A fully syntactic AC-RPO. Technical Report UPC-LSI-97, Univ. Polit. Catalunya, December 1998.

This article was processed using the LaTeX macro package with LLNCS style

Monotonic Semantic Path Orderings

Cristina Borralleras†, Maria C. F. Ferreira‡ and Albert Rubio††

† Universitat de Vic, Spain
Email: Cristina.Borralleras@uvic.es
‡ Universidade Nova de Lisboa, Portugal
Email: cf@di.fct.unl.pt
†† Universitat Politècnica de Catalunya, Barcelona, Spain
Email: rubio@lsi.upc.es

Due to it simplicity the Recursive Path Ordering (RPO) [Der82] has been extensively used in (semi-)automatic termination proof. Unfortunately, RPO can only be used to prove termination of a particular class of term rewriting systems (TRS), namely the *simply terminating* TRS's.

A way to avoid this restriction is given in the Semantic Path Ordering (SPO) [KL80] in which the *precedence* on function symbols is replaced by any (well-founded) underlying (quasi-)ordering involving the whole term. The SPO is too general to be automated, but, even if we restrict the kind of underlying orderings allowed, it still has an important weakness: it is not monotonic in general.

With the aim of using SPO in (semi-)automatic proofs of termination we have defined a *monotonic* version of SPO, which is obtained by restricting the use of arguments in the first comparison. Then by considering only some classes of underlying orderings we can automate the SPO for proving termination. In particular, we can capture many termination proof methods based on transformations.

Additionally we define some original variations of the RPO, which combined with some simple transformations on terms, turn out to be suitable candidates as underlying ordering in the SPO. Using these orderings we can fully automate the termination proof of several non-simply terminating TRS's, some of them including mutual recursion.

Finally we have adapted the monotonic semantic path ordering to prove termination of rewriting modulo asociativity and commutativity (AC), by combining our results with the recent results for RPO modulo AC in [Rub98].

References

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

[KL80] S. Kamin and J.-J. Levy. Two generalizations of the recursive path ordering. Unpublished note, Dept. of Computer Science, Univ. of Illinois, Urbana, IL, 1980.

[Rub98] Albert Rubio. A fully syntactic AC-RPO. Technical Report UPC-LSI-97, Univ. Polit. Catalunya, December 1998.

This article was processed using the LaTeX macro package with LLNCS style

Higher-Order Recursive Path Orderings

Jean-Pierre Jouannaud† and Albert Rubio‡

† LRI, Bat. 490, CNRS/Université de Paris Sud, 91405 Orsay, FRANCE
Email: Jean-Pierre.Jouannaud@lri.fr
‡ Universitat Politècnica de Catalunya, Pau Gargallo 5, 08028 Barcelona, SPAIN
Email: rubio@lsi.upc.es

Rewrite rules are increasingly used in programming languages and logical systems, with two main goals: defining functions by pattern matching; describing rule-based decision procedures. ML, Elf [9] and Isabelle [8] examplify the first use. A future version of Coq [4] will examplify the second use [3]. In Elf and Isabelle, rules operate on terms in β-normal, η-expanded form. In ML and Coq, they operate on arbitrary terms. In the future version of Coq, both kinds will probably coexist.

Our ambition is to develop for the higher-order case the same kind of semi-automated termination proof techniques that are available for the first-order case, of which the most popular one is the recursive path ordering [2].

Our first important contribution is a reduction ordering for typed higher-order terms which conservatively extends Dershowitz's recursive path ordering for first-order terms. In the latter, the precedence rule allows to decrease from the term $s = f(s_1, \ldots, s_n)$ to the term $g(t_1, \ldots, t_n)$, provided that (i) f is bigger than g in the given precedence on function symbols, and (ii) s is bigger than every t_i. For typing reasons, in our ordering the latter condition becomes: (ii) for every t_i, either s or some s_j is bigger than or equal to t_i. Indeed, we can instead allow t_i to be obtained from the subterms of s by computability preserving operations. Here, computability refers to Tait and Girard's strong normalization proof technique which we have used to show that our ordering is well-founded and compatible with β-reductions. To hint at the strength of the ordering, let us mention that the polymorphic version of Gödel's recursor for the natural numbers is easily oriented. And indeed, our ordering applies to arbitrary polymorphic signatures. Several other examples are given which examplify the expressive power of the ordering.

As a second contribution, we then restrict the previous ordering so as to obtain a reduction ordering operating on terms in η-long β-normal form. Only very weak orderings had been obtained so far for that purpose [6, 7, 1, 5].

References

1. J. Van de Pol and H. Schwichtenberg. Strict functional for termination proofs. In *Typed Lambda Calculi and Applications, Edinburgh*. Springer-Verlag, 1995.
2. Nachum Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, March 1982.

3. G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo, 1998. Rapport de recherche INRIA 3400.
4. Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Christine Paulin-Mohring, and Benjamin Werner. The coq proof assistant user's guide version 5.6. INRIA Rocquencourt and ENS Lyon.
5. Jean-Pierre Jouannaud and Albert Rubio. Rewrite orderings for higher-order terms in η-long β-normal form and the recursive path ordering. *Theoretical Computer Science*, 208(1–2):3–31, November 1998.
6. C. Loría-Sáenz and J. Steinbach. Termination of combined (rewrite and λ-calculus) systems. In *Proc. 3rd Int. Workshop on Conditional Term Rewriting Systems, Pont-à-Mousson, LNCS 656*, volume 656 of *Lecture Notes in Computer Science*, pages 143–147. Springer-Verlag, 1992.
7. Olav Lysne and Javier Piris. A termination ordering for higher order rewrite systems. In *Proc. 6th Rewriting Techniques and Applications, Kaiserslautern, LNCS 914*, Kaiserslautern, Germany, 1995.
8. Lawrence C. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*. Academic Press, 1990.
9. Frank Pfenning. Elf: A language for logic definition and verified meta-programming. In *Proc. 4th IEEE Symp. Logic in Computer Science*, pages 313–322, 1989.

A terminating schema for higher-order rewrite systems

Frédéric Blanqui[†], Jean-Pierre Jouannaud[†] and Mitsuhiro Okada[‡]
[†] LRI, CNRS and Université de Paris-Sud
Bât 405, 91405 Orsay Cedex, France
[‡] Department of Philosophy, Keio University,
108 Minatoku, Tokyo, JAPAN
Tel: +33-1-69156905 FAX: +33-1-69156586 Tel-FAX: +33-1-43212975
`http://www.lri.fr/~blanqui/`

February 1st, 1999

We propose to present a new technique to prove the termination of computations for languages combining terms of some typed λ-calculus [1], inductively defined base types [7], and function symbols defined by higher-order rewrite rules [11, 12].

In contrast with Combinatory Rewrite Systems [13] or, equivalently, Higher-order Rewrite Systems [14] which, in general, are not terminating systems since they are based on the untyped λ-calculus (their authors are essentially interested in the confluence/Church-Rosser property), in our higher-order rewrite rules, we assume that the lefthand sides are made only of type constructors and function symbols: they are algebraic. However, this is powerful enough to capture any algebraic rewrite systems [8], or higher-order rewrite rules defining recursors for higher-order inductive types (called "strictly positive" in [7]) like the ones for ordinals.

Since the works by Breazu-Tannen and Gallier [4, 5], and independently by Okada [15], on the termination of computations for the simply-typed λ-calculus combined with first-order (algebraic) rewriting [1], it is well known how to deal with first-order rewriting even with richer type systems: their combination is modular, that is the combined system is terminating if the first-order rewrite system and the type system are both terminating. Now, for further combining such a system with higher-order rewriting, it is necessary for the first-order rewrite system to be non-duplicating. Termination of higher-order rewriting with algebraic lefthand sides has been studied in [11, 12] where a rule schema was introduced that generalizes that of higher-order primitive recursion. It was capturing the rules for recursors of "basic" inductive types like naturals and lists, or the "map" function on lists, but it was unable to treat the case of the recursor for non-basic inductive types like the one of ordinals.

Our new proof technique allows us to deal with such rewrite rules. It relies on two important novelties. Firstly, we define the set of higher-order rewrite rules as an inductively defined set, that we call the "General Schema". This will allow us to inductively reason about these rules. Secondly, we use an interpretation for comparing the arguments of a function call defined by the lefthand side of a rule with the recursive calls occurring in the righthand side. This interpretation is based on the notion of "critical subterm" for a strictly positive inductive type. Let us consider the example of the recursor `rec` on the strictly positive inductive type `ord` of ordinals whose constructors are `0` of type `ord`, `S` of type `ord` $\rightarrow$ `ord` and `lim` of type (`nat` $\rightarrow$ `ord`) $\rightarrow$ `ord`, where `nat` is the basic inductive type of naturals. `rec` is defined by the following rewrite rules:

$$\begin{array}{rcl}
\mathtt{rec}(\mathtt{0}, u, v, w) & \longrightarrow & u \\
\mathtt{rec}(\mathtt{S}(x), u, v, w) & \longrightarrow & v\ x\ \mathtt{rec}(x, u, v, w) \\
\mathtt{rec}(\mathtt{lim}(f), u, v, w) & \longrightarrow & w\ f\ \lambda n{:}\mathtt{nat}.\mathtt{rec}(f\ n, u, v, w)
\end{array}$$

[1] Both works follow from the pioneer work by Breazu-Tannen on the confluence of the combination of the simply-typed λ-calculus with first-order (algebraic) rewriting [3].

In the third rule, we have to compare $\texttt{lim}(f)$ to $(f\ n)$. The fact that, for a strictly positive inductive type σ, the type of an argument of a constructor is of the form $\sigma_1 \rightarrow \sigma_2 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \sigma$ where σ does not occur in any σ_i, allows us to neglect the terms to which an argument of a constructor is applied, giving the "critical subterm". Within the example, the critical subterm of $(f\ n)$ is simply f which is a strict subterm of $\texttt{lim}(f)$.

To illustrate our discourse, we will take the example of the simply-typed λ-calculus. We have submitted an article to RTA'99 [2] where we present this method in the case of the Calculus of Constructions [6], and that you can find on the web. Assuming that we are given some base types, upon which function types may be built using the arrow operator $\rightarrow$, the language we consider is defined as follows:

$$t := x \mid \lambda x{:}\sigma.t \mid (t_1\ t_2) \mid C(t_1, \ldots, t_n) \mid f(t_1, \ldots, t_n)$$

where x designates a variable, $\lambda x{:}\sigma.t$ the function that associates t to each variable x of type σ, $(t_1\ t_2)$ the application of the function t_1 to the term t_2, $C(t_1, \ldots, t_n)$ the constructor term of some inductive type, and $f(t_1, \ldots, t_n)$ the application of some defined function symbol f to some terms $t_1, \ldots, t_n$. Naturally, assuming that each function symbol is defined by a set of rewrite rules $f(t_1, \ldots, t_n) \longrightarrow t$, the computations associated to this language will be the combination of these rewrite rules with the β-reduction, that is, the operation that substitutes the bound variable of an abstraction by the argument to which it is applied: $(\lambda x{:}\sigma.t_1\ t_2) \longrightarrow t_1\{x \mapsto t_2\}$. Since such a calculus is not terminating in general, we restrict our attention to the so-called "well-typed" terms, that is the terms to which we can give some type. For example, an abstraction $\lambda x{:}\sigma.t$ has type $\sigma \rightarrow \tau$ if x is a variable of type σ and t is a term of type τ. We refer the reader to [1] for an introduction to generalized type systems. We denote by $\vdash t : \sigma$ the fact that t has type σ.

The termination proof technique that we propose relies on the well known proof technique of "reducibility candidates" due to Tait and Girard [10, 9]. Since a direct proof by induction on the structure of the typed judgements does not go through, the idea is to use a strengthened induction hypothesis and to instead prove that, every term t of type σ is "reducible", that is, belongs to the interpretation of its type that we denote by $[\![\sigma]\!]$. Then, the termination follows from the fact that $[\![\sigma]\!]$ is chosen so as to be included into the set of terminating terms. In fact, to treat the abstraction case, we always reason modulo a "reducible" substitution, that is, a substitution that associates reducible terms to each variable of its domain. Such a proof can be further extended in a modular way to deal with the case of function symbols. Indeed, it suffices to prove that, for every well-typed terms $f(t_1, \ldots, t_n)$ and reducible substitution θ, $f(t_1\theta, \ldots, t_n\theta)$ is also reducible.

So, one can see how interesting it is to inductively define the set of higher-order rewrite rules for which we would be able to prove the previous result. Given a lefthand side $f(t_1, \ldots, t_n)$, it suffices to start with the "accessible" subterms, that is, the subterms that are reducible whenever $t_1, \ldots, t_n$ are so. Then, we can extend this base set with any operation that preserves the reducibility. This includes the abstraction, the application, the application of a constructor or of a defined function symbol and the reduction. We can also add recursive calls whose arguments are strict subterms of $t_1, \ldots, t_n$ modulo the critical subterm interpretation. But, in this case, since the critical subterm interpretation is not compatible with the reduction relation, and not stable by substitution either, in order to carry out the proof of reducibility of the higher-order function symbols, we need to define yet another interpretation which is compatible with the critical subterm interpretation and satisfies the previous properties.

Finally, we expect to extend our technique to generalized inductive types as defined in [7], that is, inductively defined polymorphic types, and inductively defined predicates (dependant types). Together with function symbol definitions at the type level, our technique would lead to a simpler proof of termination of the Calculus of Inductive Constructions with strong elimination [16] (but without taking into account the η-reduction into the type conversion rule).

We also think that our technique could be useful for resolving two problems left open. First, the extension of the General Schema so as to capture the recursor rules for non-strictly positive inductive types. One example is given by a coding of Montague' semantical objects as an inductive type $\texttt{mon}$ with a constructor of type $((\texttt{mon} \rightarrow \texttt{bool}) \rightarrow \texttt{mon}) \rightarrow \texttt{mon})$, where $\texttt{bool}$ is the type of booleans. Secondly, it would be nice to allow function symbols to be defined by pattern-matching on some higher-order function symbol like the addition operator on ordinals. Indeed, it is possible to do so with first-order operators like the addition operator on naturals or the append operator on lists, allowing one to refine the definition of the map function for example.

References

[1] H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1992.

[2] F. Blanqui, J.-P. Jouannaud, and M. Okada. The Calculus of Algebraic Constructions, 1998. RTA'99 submission.

[3] V. Breazu-Tannen. Combining algebra and higher-order types. In *Proceedings, Third Annual Symposium on Logic in Computer Science*, pages 82–90. IEEE Computer Society, 5–8 July 1988.

[4] V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic strong normalization. In *Proceedings of the Sixteenth International Colloquium on Automata, Languages and Programming*, volume 372 of *LNCS*, pages 137–150, Stresa, Italy, July 1989. European Association of Theoretical Computer Science, Springer-Verlag.

[5] V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic strong normalization. *Theoretical Computer Science*, 83(1):3–28, June 1991.

[6] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:96–120, 1988.

[7] T. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, LNCS 417. Springer-Verlag, 1990.

[8] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–309. North-Holland, 1990.

[9] J. Gallier. On Girard's "Candidats de Réductibilité". In P.-G. Odifreddi, editor, *Logic and Computer Science*. North Holland, 1990.

[10] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1988.

[11] J.-P. Jouannaud and M. Okada. Executable higher-order algebraic specification languages. In *Proc. 6th IEEE Symp. Logic in Computer Science, Amsterdam*, pages 350–361, 1991.

[12] J.-P. Jouannaud and M. Okada. Abstract Data Type Systems. *Theoretical Computer Science*, 173(2):349–391, February 1997.

[13] J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121(1-2):279–308, December 1993.

[14] T. Nipkow. Higher-order critical pairs. In *Proc. 6th IEEE Symp. Logic in Computer Science, Amsterdam*, pages 342–349, 1991.

[15] M. Okada. Strong normalizability for the combined system of the typed lambda calculus and an arbitrary convergent term rewrite system. In G. H. Gonnet, editor, *Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation*, pages 357–363. ACM Press, July 1989.

[16] B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris VII, 1994.

A semantical analysis of structural recursion

Andreas Abel, Thorsten Altenkirch

February 1st, 1999

We are developing a system (MuTTI - Munich Type Theory Implementation) with dependent types which can be used for the development of provably correct programs in Type Theory. Inspired by Coquand's pattern matching for dependent types [Coq92] as implemented in the ALF system [Alf94] and its successors, we define a total language as a subset of a partial one. Hence, we are faced with the problem of verifying termination.

We restrict ourselves to structural recursion, where by structural recursion we mean that the only termination orderings we consider are lexical products of the natural structural ordering on a strictly positive datatype. We also allow mutual recursion. A further restriction is that smaller terms are only generated by primitive operators like case-analysis, projection and application.

In the type-theoretic context this is sufficient, since general terminating recursion can be represented by adding additional (computationally irrelevant) parameters. We are not sure whether structural recursive without the afore mentioned restriction is actually decidable.

Abel implemented a termination checker for a simply typed sublanguage of MuTTI (called foetus), this system allows mutual recursive definitions on general strict positive datatypes and returns a lexical ordering on the arguments of the function if one exists.

In the work we want to present here, we show that from foetus output we can actually conclude that the function is terminating. We define a semantic interpretation of each type and formally define the structural ordering on semantic values of possibly different types. A central result is that the structural ordering is wellfounded at each type. A general soundness theorem allows us to conclude that each term accepted by the foetus system actually terminates.

Our approach is related to the work by Telford & Turner, who are interested in a total functional programming language (ESFP). In a recent (unpublished) article [TTu98b] they present also a termination analysis based on abstract interpretation. It seems that they accept a larger set of functions but that our analysis of the output of the termination checker is more detailed.

Our next goal is to extend our termination checker to coinductively defined types [Coq93, TTu97b] and dependent types, including the definition of universes. We also hope to be able to answer the question whether the restriction of structural recursive function is actually necessary.

References

[Alf94] Thorsten Altenkirch, Veronica Gaspes, Bengt Nordström, and Björn von Sydow. *A user's guide to ALF*. Department of Computing Science, University of Göteborg/Chalmers, http://www.cs.chalmers.se/Cs/Research/Logic/alf/guide.html, May 1994.

[Coq92] Thierry Coquand. Pattern matching with dependent types. *(to be updated)*, 1992.

[Coq93] Thierry Coquand. Infinite objects in type theory. In Henry Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs (TYPES '93)*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, 1993.

[TTu97b] Alastair Telford and David Turner. Ensuring Streams Flow. In Michael Johnson, editor, *Algebraic Methodology and Software Technology, 6th International Conference, AMAST '97, Sydney Australia, December 1997*, volume 1349 of *Lecture Notes in Computer Science*, pages 509–523. AMAST, Springer-Verlag, December 1997.

[TTu98b] Alastair Telford and David Turner. Guarded Recursion in ESFP. *Submitted to TYPES '98 proceedings*, 1998.

On Termination and Derivation Lengths for Ground Rewrite Systems

Dieter Hofbauer[1]

Universität GH Kassel
Fachbereich 17 Mathematik/Informatik
D–34109 Kassel, Germany
`dieter@theory.informatik.uni-kassel.de`

Abstract. It is shown that for terminating ground term rewrite systems the length of derivations is linearly bounded in the size of the starting term. (The constant of the linear bound, however, depends exponentially on the cardinality of the signature.) This is proven by constructing a termination proof via a suitable interpretation into the natural numbers.

Terminating ground systems are not necessarily ω-terminating, i.e., a monotone interpretation over ω might not exist. As an example, consider the system with the two rules $g(a) \to g(b)$ and $f(b) \to f(a)$. We show, however, that those systems are "almost ω-terminating" in the sense that a monotone interpretation over a well-founded ordering of the form $\alpha + \omega$ can always be constructed where α is a finite partially ordered set.

In the second part of my talk, it is shown how to apply the theory of tree languages to decision problems related to ground systems, e.g., termination, normalization, fair termination. Extensions of known results to regular canonical (string and tree) systems are given.

A slow growing analysis of the canonical rewrite system for the Ackermann function

Andreas Weiermann
Institut für Mathematische Logik und Grundlagenforschung
der Westfälischen Wilhelms-Universität Münster
Einsteinstr. 62, D-48149 Münster, Germany

Abstract

Let $R_A := \{A(0,y) \to S(y); A(S(x),y) \to A(x,S(y)), A(S(x),S(y))\}$ be the canonical rewrite system for the Ackermannfunction. It is well known the the ordinal notation system T for the ordinals less than Γ_0 yields a canonical termination proof for R_A. In addition the slow growing hierarchy up to Γ_0 classifies the R_A derivation lengths (more precisely, the derivation lengths of A in the system R_A). Let T^- be the subsystem of T in which the ordinal addition function is omitted. Then – due to de Jongh – the order type of T^- is ε_0. Nevertheless T^- yields the termination of R_A. We investigate the behaviour of the associated slow growing hierarchy for T^- and obtain the following two results (to appear in JSL):
1. If the elements of T^- are considered as tree ordinals then the associated slow growing hierarchy classifies the R_A-derivation lengths.
2. If the elements of T^- are considered as set-theoretic ordinals then the associated slow growing hierarchy classifies the Kalmar-elementary recursive functions.

These results indicate that sometimes tree ordinals reflect the computational behaviour of rewrite systems better than set-theoretic ordinals.

A Totally Terminating Rewrite System whose Complexity is not Γ_0-recursive

INGO LEPPER*

Weiermann [2] has shown that the complexity of a rewrite system which is known to terminate by Kruskal's Theorem is bound by $(s^\omega)^{\bar{\theta}_{\Omega^\omega}(0)}$. This result appeared to be of rather theoretical nature, because all known examples had multiply recursive complexities, until recently Touzet [1] gave examples of (even totally) terminating rewrite systems whose complexities are not multiply recursive. This was established by simulating the famous Hydra battle for ordinals below ε_0.

Based on the work of Touzet we define a totally terminating rewrite system which is able to simulate Hydra battles for all ordinals below Γ_0, the first infinite ordinal which is closed under $+$ and the binary Veblen φ-function. In a second step, we show how the ternary φ-function can be approached by our method.

Since the bound of Weiermann is the first infinite ordinal closed under $+$ and all n-ary φ-functions, we are led to the conjecture that this bound is sharp.

References

[1] H. Touzet. Encoding the Hydra Battle as a Rewrite System. `http://www.loria.fr/~touzet/mfcs98.ps.gz`, 1998.

[2] A. Weiermann. Complexity Bounds for some Finite Forms of Kruskal's Theorem. *Journal of Symbolic Computation*, 18:463–488, 1994.

*Supported by DFG grant WE 2178/2-1

Translating TRSs into OS-TRSs: a result on modularity of termination*
(Abstract)

P. Inverardi and M. Nesi

Dipartimento di Matematica Pura ed Applicata
Università degli Studi di L'Aquila
via Vetoio, 67010 Coppito, L'Aquila, Italy

email: {inverard,monica}@univaq.it

An important aspect of the termination property of term rewriting systems (TRSs) is its *modularity*, that is termination is preserved under combinations of TRSs. These combinations vary from TRSs with a disjoint union of signatures to more complex, hierarchical combinations of TRSs, through TRSs which share constructors [Toy87,GG87,TKB89,Pol92,MT93,FJ94,Ohl94,Der95,Gra96]. Very often, non-termination of the combined system arises because of subtle and (semantically) meaningless interactions between the TRSs. The criteria used to guarantee modularity are usually given at the syntactic level by relying on the form of terms which occur in the rules of the TRSs, often using a detailed inspection of the structure of the terms. Frequently, it is difficult to understand intuitively how these syntactic criteria interact.

We present a translation of unsorted TRSs into suitable order-sorted TRSs (OS-TRSs) [Gna92,Mat96,ÖL96]. The aim of such a translation is to add more structure (i.e. the sorts) to the rewriting rules and the terms to be rewritten, in order to enforce termination. The translation builds on the usual distinction between constructors and functions to yield a rewriting relation that induces a particular rewriting strategy, i.e. a bottom-up (innermost) rewriting constrained by the sorts of terms.

The translation is correct and complete, in the sense that the OS-TRS T resulting from the translation of an unsorted TRS R is equivalent to R with respect to terminating derivations. This means that every terminating derivation in the unsorted TRS R can be simulated by a derivation in T that reaches the same normal form. Thus, for the class of systems for which the completeness holds, the translation prunes all the possible infinite derivations of R. Moreover, the OS-TRS T can, on the one hand, have properties that the TRS R does not have (e.g. termination) and, on the other hand, allows a different combination of systems, for which more general modularity results for termination can be proved.

* This work was partially supported by MURST 40% "Tecniche formali per la specifica, l'analisi, la verifica, la sintesi e la trasformazione di sistemi software".

Our result is the modularity of termination for a subclass of OS-TRSs. This result holds for the OS-TRSs resulting from the translation which, with respect to the original unsorted TRSs, possess a more constrained structure both in the terms and in the rewriting relations. However, the modularity of termination for the combined OS-TRSs provides a non-trivial criterion for modular combinations of a particular class of OS-TRSs. Our current combination only allows common constructors, and results on modularity of termination for such a combination already exist for suitable subclasses of TRSs. We believe that, by adding the sort information and establishing suitable orderings on sorts, it will be possible to achieve modularity of termination for more complex combinations of TRSs, such as hierarchical unions.

References

[Der95] N. Dershowitz. Hierarchical Termination. In *Proc. CTRS '94*, *Lecture Notes in Computer Science* 968, pp. 89-105, Springer-Verlag, 1995.

[FJ94] M. Fernández and J.-P. Jouannaud. Modular Termination of Term Rewriting Systems Revisited. In *Proc. ADT '94*, *Lecture Notes in Computer Science* 906, Springer-Verlag, 1994.

[GG87] H. Ganzinger and R. Giegerich. A Note on Termination in Combinations of Heterogeneous Term Rewriting Systems. In *Bulletin of EATCS* 31, pp. 22-28, 1987.

[Gna92] I. Gnaedig. Termination of order-sorted rewriting. In *Proc. ALP '92*, *Lecture Notes in Computer Science* 632, pp. 37-52, Springer-Verlag, 1992.

[Gra96] B. Gramlich. Termination and Confluence Properties of Structured Rewrite Systems. PhD Thesis, University of Kaiserslautern, 1996.

[Mat96] B. Matthews. Dynamic Order-Sorted Term-Rewriting Systems. PhD Thesis, University of Glasgow, 1996.

[MT93] A. Middeldorp and Y. Toyama. Completeness of Combinations of Constructor Systems. *Journal of Symbolic Computation*, 15:331-348, 1993.

[Ohl94] E. Ohlebush. On the modularity of termination of term rewriting systems. *Theoretical Computer Science*, 136:333-360, 1994.

[ÖL96] P. C. Ölveczky and O. Lysne. Order-Sorted Termination: The Unsorted Way. In *Proc. ALP '96*, *Lecture Notes in Computer Science* 1139, pp. 92-106, Springer-Verlag, 1996.

[Pol92] J. van de Pol. Modularity in Many-sorted Term Rewriting Systems. Master's Thesis, University of Utrecht, 1992.

[Toy87] Y. Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25:141-143, 1987.

[TKB89] Y. Toyama, J. W. Klop, and H. P. Barendregt. Termination for the direct sum of left-linear term rewriting systems. In *Proc. RTA '89*, *Lecture Notes in Computer Science* 355, pp. 477-491, Springer-Verlag, 1989.

Modularity Results for Termination Proofs using Dependency Pairs

Jürgen Giesl
Department of Computer Science, TU Darmstadt
Alexanderstr. 10, 64283 Darmstadt, Germany
E-mail: `giesl@informatik.tu-darmstadt.de`

Thomas Arts
Computer Science Laboratory, Ericsson Telecom AB
126 25 Stockholm, Sweden
E-mail: `thomas@cslab.ericsson.se`

Enno Ohlebusch
Faculty of Technology, University of Bielefeld
P.O. Box 10 01 31, 33501 Bielefeld, Germany
E-mail: `enno@TechFak.Uni-Bielefeld.DE`

Most classical techniques for automated termination proofs of term rewriting systems (TRSs) are based on simplification orderings. In practice, however, many TRSs are non-simply terminating, i.e., their termination cannot be verified by any simplification ordering. For that reason, we developed the dependency pair approach [AG97a, AG97b, AG99] which allows automated termination proofs for many TRSs where such proofs were not possible before.

In the talk we present a refinement of the dependency pair framework which further extends the class of term rewriting systems for which termination can be proved automatically. By means of this refinement, one can now prove termination in a modular way. To be more precise, in contrast to other methods, one may use several different orderings in one termination proof, cf. [AG98].

Subsequently, we present new modularity results based on dependency pairs. It it is well known that simple termination is modular for certain kinds of combinations of TRSs. This result is of practical relevance, because it ensures modularity whenever the previous standard techniques for automated termination proofs succeed. But by the development of dependency pairs, the class of TRSs where termination is provable automatically has been increased considerably. The reason is that by using (quasi-)simplification orderings in combination with dependency pairs, it is now possible to prove termination of non-simply terminating systems automatically. Thus, it is natural to ask whether modularity of simple termination can be extended to those systems which can be handled by this new technique. In our talk we show that for disjoint unions this is indeed the case, cf. [GO98]. Moreover, under certain additional conditions, this result also holds for constructor-sharing and composable systems.

References

[AG97a] T. Arts and J. Giesl. Automatically proving termination where simplification orderings fail. In M. Dauchet, editor, *Proceedings of the 7th International Joint Conference on the Theory and Practice of Software Development, TAPSOFT '97*, volume 1214 of *Lecture Notes in Computer Science*, pages 261–272, Lille, France, April 1997. Springer Verlag, Berlin.

[AG97b] T. Arts and J. Giesl. Proving innermost normalisation automatically. In H. Comon, editor, *Proceedings of the 8th International Conference on Rewriting Techniques and Applications, RTA-97*, volume 1232 of *Lecture Notes in Computer Science*, pages 157–171, Sitges, Spain, June 1997. Springer Verlag, Berlin.

[AG98] T. Arts and J. Giesl. Modularity of termination using dependency pairs. In T. Nipkow, editor, *Proceedings of the 9th International Conference on Rewriting Techniques and Applications, RTA-98*, volume 1379 of *Lecture Notes in Computer Science*, pages 226–240, Tsukuba, Japan, March/April 1998. Springer Verlag, Berlin.

[AG99] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 1999. To appear.

[GO98] J. Giesl and E. Ohlebusch. Pushing the frontiers of combining rewrite systems farther outwards. In *Proceedings of the Second International Workshop on Frontiers of Combining Systems, FroCoS '98*, Applied Logic Series, Amsterdam, The Netherlands, October 1998. Kluwer Academic Publishers, Amsterdam.

Verification of Erlang Processes*

Thomas Arts[1] and Jürgen Giesl[2]

[1] Computer Science Laboratory, Ericsson Telecom AB, 126 25 Stockholm, Sweden, E-mail: thomas@cslab.ericsson.se
[2] FB Informatik, Darmstadt University of Technology, Alexanderstr. 10, 64283 Darmstadt, Germany, E-mail: giesl@informatik.tu-darmstadt.de

Erlang is a functional language, developed and extensively used within Ericsson Telecom, for implementing concurrent systems [AVWW96]. In the application area of these programs there are high demands on the correctness of the software. For that reason, there is great interest for a formal verification of critical and complicated components in order to isolate potential bugs that are not or cannot be found by testing.

Techniques for verification are in general based on model-checking and theorem proving. There even exists a special purpose theorem prover for Erlang programs [Dam95,ADFG98]. The advantage of model-checking is that it can be performed completely automatically, but its use is restricted to an abstraction of the communicating behaviour of the program. The use of a theorem prover demands a lot of human interaction and understanding of the problem, but this allows to prove properties of the outcome and behaviour of the functions as well.

Termination is a property that often has to be proved during verification, e.g. for function calls of which the result is irrelevant for the final outcome, termination might still be important for progress. Moreover, even properties of non-terminating programs (which often occur in the application area) might sometimes be expressed in terms of a termination property. Since proving this kind of termination properties goes far beyond the possibilities of model-checking, we have been looking for other techniques to automatically perform this task.

The semantics of Erlang differs from those of term rewriting systems, i.e. Erlang is not a pure functional language and the notion of concurrency is lacking in the standard term rewriting formalisms. Nevertheless, sequential parts of Erlang (i.e. programs without communication) that are written like pure functions are so similar to TRSs that treating the program like a TRS is almost natural. Doing so, techniques for proving termination of TRSs automatically, in particular the dependency pair approach [AG97,AG98a,AG99], become available. Additionally, techniques for innermost termination come into scope, since the TRSs corresponding to Erlang functions are non-overlapping.

In a small but realistic case-study we demonstrated the usefulness of this approach [AG98b]. We proved from a certain Erlang process that after constantly

* This work was partially supported by the Deutsche Forschungsgemeinschaft under grants no. Wa 652/2-2 as part of the focus program 'Deduktion'.

receiving a certain message, it eventually would send this message to another process. The proof is divided in several steps. First, we translate the process and the property to a termination problem of a TRS. This step is performed by hand, but care has been taken to stay close to the original problem and to make the transformation commonly applicable and understandable. Second, we extend the dependency pair approach by introducing *rewriting of dependency pairs*. Lastly, we can automatically prove termination of the obtained TRS by using our refined techniques.

References

[AVWW96] J. Armstrong, R. Verding, C. Wikström and M. Wiliams, *Concurrent Programming in Erlang.* 2:nd edition, Pretence Hall, 1996.

[ADFG98] T. Arts, M. Dam, L-å. Fredlund, and D. Gurov, System Description: Verification of Distributed Erlang Programs. In *Proc. CADE'98*, LNAI 1421, pp. 38–42, Lindau, Germany, July 1999.

[AG97] T. Arts & J. Giesl, Proving innermost normalisation automatically. In *Proc. RTA-97*, LNCS 1232, pp. 157–172, Sitges, Spain, 1997.

[AG98a] T. Arts & J. Giesl, Modularity of termination using dependency pairs. In *Proc. RTA-98*, LNCS 1232, pp. 226–240, Tsukuba, Japan, 1998.

[AG98b] T. Arts & J. Giesl, Applying Rewrite Techniques for Verification of Erlang Processes. *Tech. Rep. IBN 98/50*, Technical University Darmstadt, Germany, December 1998

[AG99] T. Arts & J. Giesl, Termination of term rewriting using dependency pairs. *Theoretical Computer Science*. To appear.

[Dam95] M. Dam, Compositional proof systems for model checking infinite state processes. In *Proc. CONCUR'95*, LNCS 962, pp. 12–26, 1995.

Invariants, patterns and weights for ordering terms

Ursula Martin

University of St Andrews, Computer Science Department
St Andrews, Fife KY16 6SS, Scotland

Any simplification ordering on a monadic term algebra is an extension of an ordering by weight. We extend this result to simplification orderings over arbitrary terms, by considering a related monadic term algebra called the spine. We show that any simplification ordering on the spine lifts to an ordering on the full term algebra. Conversely a simplification ordering on the term algebra defines a weight function on the spine which in turn can be lifted to a real valued function on the original ground terms, with the property that the original ordering is an extension of the weight ordering thus defined on ground terms. Thus we may associate numeric weights to any simplification ordering on the term algebra, and hence determine a classifying space. We also investigate the Knuth Bendix and polynomial orderings in this light, provide a general framework for orderings terms by counting embedded patterns, and investigate linear orderings related to the recursive path ordering.

See http://www-theory.dcs.st-and.ac.uk/~um/publications/papersonline.html

On Some Abstract Termination Criteria

Bernhard Gramlich

Technische Universität Wien
Institut für Computersprachen
Resselgasse 3, A-1040 Wien, Austria

e-mail: `gramlich@logic.at`
www: `http://www.logic.at/~gramlich`

Abstract. We investigate some abstract criteria for proving termination of (abstract and term) rewriting systems. In particular, we consider the problem under which conditions termination follows from weak termination, both locally and globally. We provide some refined versions of a corresponding known result from [Klo80]. Furthermore we review some other simple, but quite useful techniques for proving termination of a rewrite relation by reducing the problem to termination of some subrelation. Applying these abstract criteria to term rewriting systems reveals the essence of some known termination results and partially enables generalized versions.

References

[Klo80] Jan Willem Klop. *Combinatory Reduction Systems.* Mathematical Centre Tracts 127, Centre for Mathematics and Computer Science, Amsterdam, 1980.

Top Termination of CL(S)

Johannes Waldmann

Institut für Informatik, Universität Leipzig
Augustusplatz 10-11, D-04109 Leipzig, Germany
joe@informatik.uni-leipzig.de
http://www.informatik.uni-leipzig.de/~joe/

Introduction. In a term rewriting system (TRS) that is both confluent and terminating, normal forms are unique (each computation has a unique result) and thus the word problem is decidable. Even if the TRS is not terminating, we would like to obtain partial results of computations, and to compare terms. We lift the restriction that redexes must eventually vanisch (that is, termination) and require only that each initial segment of the term eventually becomes free from redexes.

A TRS is called *top terminating* iff in each infinite derivation, the redex is at the root of the term only finitely often. If a TRS is top terminating and confluent, we declare the (unique) limit of a fair reduction starting from a term X to be the normal form of X. [DKP91] [KKSdV95]

Main result. The fragment of Combinatory Logic with the only combinator $\mathbf{S}xyz \to xz(yz)$ is top terminating.

This seems to require a rather technical proof, that can, however, be supported by machine calculations.

Aim of the talk. While the result perhaps is nice, but certainly not earth shattering, we mainly present it to invite a discussion on general methods for proving top termination that could be applied here, resulting in a) a nice pen-and-paper proof or b) a fully automated proof.

Open Problems. Describe infinite normal forms in $\mathbf{CL}(\mathbf{S})$, and thus solve the word problem.

Related Work. The word problem for $\mathbf{CL}(\mathbf{L})$ with $\mathbf{L}xy \to x(yy)$ is shown decidable in [Sta89], [SWB93]. Normalization is decidable for $\mathbf{CL}(\mathbf{S})$ [Wal98].

References

[DKP91] Nachum Dershowitz, Stéfane Kaplan, and David A. Plaisted. Rewrite, rewrite, rewrite, rewrite, rewrite, ... *Thoeretical Computer Science*, 83:71–96, 1991.

[KKSdV95] Richard Kennaway, Jan Willem Klop, Ronan Sleep, and Fer-Jan de Vries. Transfinite reductions in orthogonal term rewriting systems. *Information and Computation*, 119(1):18–38, May 1995.

[Nip98] Tobias Nipkow, editor. *Rewriting Techniques and Applications*, number 1379 in Lecture Notes in Computer Science. Springer Verlag, 1998.

[Sta89] Richard Statmann. The word problem for Smullyan's lark combinator is decidable. *J. Symbolic Comput.*, 7:103–112, 1989.

[SWB93] M. Sprenger and M. Wymann-Böni. How to decide the lark. *Theoretical Computer Science*, 110:419–432, 1993.

[Wal98] Johannes Waldmann. Normalization of S terms is decidable. In Nipkow [Nip98], pages 138–150.

Strong convergence of term rewriting using strong dependency pairs

(Extended abstract)

Jürgen Giesl[1], Vincent van Oostrom[2], and Fer-Jan de Vries[3]

[1] Department of Computer Science, Darmstadt University of Technology, Alexanderstraße 10, 64283 Darmstadt, Germany.
[2] Department of Philosophy, Utrecht University, P.O.Box 80.126 3508 TC, the Netherlands.
[3] Rewriting Group, ETL, Tsukuba, 305-8568 Japan. Email: `ferjan@etl.go.jp`

Introduction. Let $\mathcal{R}$ be a finite term rewrite system over a finite signature [4]. Recall the theorem of Arts and Giesl [1] that $\mathcal{R}$ is terminating if and only if $\mathcal{R}$ satisfies the dependency pair criterion. Inspection of the proof learns that if we decrease the set of dependency pairs, then the set of infinite reduction sequences that can be rejected by the dependency pair criterion may decrease. An interesting extension of the set of finite reduction sequences in $\mathcal{R}$ is the set of strongly converging reductions in $\mathcal{R}$ of Kennaway e.a. [2]. In general not all infinite reductions in $\mathcal{R}$ will be strongly converging.

In this note we will reduce the set of dependency pairs for $\mathcal{R}$ to a subset of *strong dependency pairs*. We will prove that $\mathcal{R}$ satisfies a similar strong dependency pair criterion if and only if all reductions in $\mathcal{R}$ are strongly convergent.

Strongly converging reductions. Recall that a reduction is *strongly converging* [2], if either it is a finite reduction or it is an infinite reduction $t_0 \to t_1 \to \ldots$ satisfying $\lim_{n\to\infty} d_n = \infty$, where d_n is the depth of the redex contracted in the reduction step $t_n \to t_{n+1}$. Among the possibilities [3] the standard one is to measure depth by the number of nodes on the path from the root to the redex. Like termination, strong convergence of term rewriting systems is undecidable in general. Observe that the terms of an infinite strongly convergent reduction converge to a (possibly infinite) limit. Strong convergence captures the idea of progressing approximation: the limit of an infinite strongly convergent reduction can alternatively be described as the limit $\lim_{n\to\infty} s_n$ of approximating prefixes $s_n \sqsubseteq t_n$, where the s_n remain reduction free in the rest of the reduction. Here we will not consider transfinite reductions and infinite terms.

Strong Dependency Pairs. We will now define strong dependency pairs as dependency pairs with an extra depth dependent condition. As in [1] it is notationally convenient to extend $\mathcal{R}$'s signature with a fresh, capitalised symbol F for each defined function symbol f of $\mathcal{R}$. If $f(s_1,\ldots,s_n) \to C[g(t_1,\ldots,t_m)]$ is a rule of $\mathcal{R}$, g is a defined symbol of $\mathcal{R}$ and the hole $[\ ]$ occurs at depth 0 in $C[\ ]$, then the pair $\langle F(s_1,\ldots,s_n), G(t_1,\ldots,t_n)\rangle$ is called a *strong dependency pair*.

As in [1] we define that a (possibly infinite) sequence $\langle s_1, t_1\rangle$, $\langle s_2, t_2\rangle$, ... of pairs of terms in $\mathcal{R}$ is a *chain* of if there is a substitution σ such that $t_j\sigma \to^* s_{j+1}\sigma$ in $\mathcal{R}$ for each $j \geq 1$. The hard work now goes into proving:

Theorem 1. *Let $\mathcal{R}$ be a TRS without collapsing rules. There is a reduction sequence in $\mathcal{R}$ in which infinitely many reduction steps take place at depth* 0 *if and only if there is an infinite chain of strong dependency pairs.*

Corollary 1. *A term rewrite system $\mathcal{R}$ without collapsing rules is strongly converging if and only if there is no infinite chain of strong dependency pairs.*

The strong dependency pairs form a subset of the dependency pairs. A similar proof as in [1] involving contexts with holes at depth 0 shows:

Theorem 2. *A TRS $\mathcal{R}$ without collapsing rules is strongly converging if and only if there exists a well-founded weakly monotonic quasi-order $\geq$, such that*

- *both $\geq$ and $>$ are closed under substitution,*
- *$l \geq r$ for all rules $l \to r$ in $\mathcal{R}$ and*
- *$s > t$ for all strong dependency pairs $\langle s, t\rangle$ of $\mathcal{R}$.*

Conclusion and example. As in the case of termination the benefit of the last theorem is that the proof of strong convergence of a term rewrite system is now reduced to the search of a suitable quasi-order. For example one may prove that the following non-terminating term rewriting system is strongly converging. Finally, one may observe that the concept of depth is actually a parameter of this note as in [3]. Similar theorems hold for suitable variations: the original theorem of Arts and Giesl can be recognized as an instance.

$$
\begin{array}{lcr}
\hline
filter(x : y, 0, m) & \to & 0 : filter(y, m, m) \\
filter(x : y, s(n), m) & \to & x : filter(y, n, m) \\
sieve(0 : y) & \to & sieve(y) \\
sieve(s(n) : y) & \to & s(n) : sieve(filter(y, n, n)) \\
odds(n) & \to & n : odds(s(s(n))) \\
primes & \to & s(s(0)) : sieve(odds(s(s(s(0))))) \\
take(0, x : y) & \to & x \\
take(s(n), x : y) & \to & take(n, y) \\
prime(n) & \to & take(n, primes) \\
\hline
\end{array}
$$

References

1. T. Arts and J. Giesl. Termination of rewriting using dependency pairs. *Theoret. Comput. Sci.*, to appear, 1999.
2. J.R. Kennaway, J.W. Klop, M.R Sleep, and F.J. de Vries. Transfinite reductions in orthogonal term rewriting systems. *Inform. and Comp.*, 119(1):18–38, 1995.
3. J.R. Kennaway, J.W. Klop, M.R Sleep, and F.J. de Vries. Infinitary lambda calculus. *Theoret. Comput. Sci.*, 175(1):93–125, 1997.
4. J.W. Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, eds., *Handbook of Logic in Computer Science*, volume 2, chapter 1, pages 1–117. Oxford University Press, Oxford, 1992.

Equational Termination by Semantic Labelling

Hitoshi Ohsaki* Aart Middeldorp† Jürgen Giesl‡

Semantic labelling (Zantema [5]) is a powerful tool for proving termination of term rewrite systems. The usefulness of the extension to equational term rewriting described in [5] is however rather limited. In this talk, we introduce a more powerful version of *equational semantic labelling*, parameterized by three choices: (1) the well-founded order on the underlying algebra (partial order vs. quasi-order), (2) the relation between the algebra and the rewrite system (model vs. quasi-model), and (3) the labelling of the function symbols appearing in the equations (allowed vs. forbidden). We present soundness and completeness results for the various instantiations and analyze the relationships between them.

We present several applications of our equational semantic labelling technique. We give an alternative proof of the correctness of the dependency pair approach (Arts and Giesl [1]) for proving termination of term rewrite systems. We give a short proof of the main result of Ferreira *et al.* [2]: The correctness of a version of dummy elimination for AC-rewriting which completely removes the AC-axioms. We extend some of the results of Zantema [4] concerning distribution elimination to the equational setting.

Every terminating term rewrite system can be transformed by self-labelling (a special case of semantic labelling) into a rewrite system whose termination can be shown by the recursive path order (Middeldorp *et al.* [3]). In the final part of the talk, we discuss to what extent our equational semantic labelling technique is complete in this sense.

References

[1] T. Arts and J. Giesl, *Termination of Term Rewriting Using Dependency Pairs*, Technical report IBN-97/46, Department of Computer Science, Darmstadt University of Technology, Germany, 1997. To appear in Theoretical Computer Science.

[2] M.C.F. Ferreira, D. Kesner, and L. Puel, *Reducing AC-Termination to Termination*, Proceedings of the 23rd International Symposium on Mathe-

*ohsaki@etl.go.jp, Rewriting Group, Electrotechnical Laboratory, Tsukuba 305-8568, Japan.

†ami@is.tsukuba.ac.jp, Institute of Information Sciences and Electronics, University of Tsukuba, Tsukuba 305-8573, Japan.

‡giesl@informatik.tu-darmstadt.de, Department of Computer Science, Darmstadt University of Technology, Alexanderstr. 10, 64283 Darmstadt, Germany.

matical Foundations of Computer Science, Brno (Czech Republic), LNCS 1450, pp. 239-247, 1998.

[3] A. Middeldorp, H. Ohsaki, and H. Zantema, *Transforming Termination by Self-Labelling*, Proceedings of the 13th International Conference on Automated Deduction, New Brunswick (New Jersey), LNAI 1104, pp. 373–387, 1996.

[4] H. Zantema, *Termination of Term Rewriting: Interpretation and Type Elimination*, Journal of Symbolic Computation 17, pp. 23–50, 1994.

[5] H. Zantema, *Termination of Term Rewriting by Semantic Labelling*, Fundamenta Informaticae 24, pp. 89-105, 1995.

Automatic Termination Proofs of Logic Programs via Rewrite Systems

Enno Ohlebusch

Faculty of Technology, University of Bielefeld
P.O. Box 10 01 31, 33501 Bielefeld, Germany
email: enno@TechFak.Uni-Bielefeld.DE

In the last decade, several methods have been proposed to check termination of logic programs (semi-)automatically. In [GW93], Ganzinger and Waldmann described a transformation of well-moded logic programs into deterministic conditional term rewriting systems–systems with a restricted use of extra-variables. If the transformed system $\mathcal{R}(P)$ is quasi-reductive (a property which ensures effective termination), then the logic program P is terminating for all well-moded queries but not vice versa (so the method is sound but not complete). Quasi-reductivity is in general undecidable but sufficient conditions for quasi-reductivity are known; see [Gan91, ALS94].

Inspired by the transformational approach described above, Arts and Zantema [AZ95, AZ96] proposed to transform the logic program into an unconditional term rewriting system. A similar approach appeared in [AM93]. In contrast to the former techniques, the latter does not require any prior information about modes of predicates because these are computed during the transformation according to a given query.

In Arts and Zantema's method, innermost termination of the term rewriting system ensures termination of the logic program. The advantage is that well-known powerful methods like the dependency pair approach [AG97] can now be used to prove termination of logic programs automatically. Consequently, it is remarked in [AZ95] that the suggested method "is applicable to a wider class of logic programs" and hence it is "stronger than the other results". It will be shown, however, that the technique of Ganzinger and Waldmann is equally powerful. This is done by first translating the deterministic conditional term rewriting system $\mathcal{R}(P)$ into an unconditional term rewriting system $U(\mathcal{R}(P))$. Then one can show that quasi-reductivity of $\mathcal{R}(P)$ coincides with innermost termination of $U(\mathcal{R}(P))$.

References

[AG97] T. Arts and J. Giesl. Automatically Proving Termination where Simplification Orderings Fail. In *Proceedings of the 22nd International Colloquium on Trees in Algebra and Programming*, pages 261–273. LNCS **1214**, 1997.

[ALS94] J. Avenhaus and C. Loría-Sáenz. On Conditional Rewrite Systems with Extra Variables and Deterministic Logic Programs. In *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, pages 215–229. LNAI **822**, 1994.

[AM93] G. Aguzzi and U. Modigliani. Proving Termination of Logic Programs by Transforming them into Equivalent Term Rewriting Systems. In *Proceedings of the 13th Conference on the Foundations of Software Technology and Theoretical Computer Science*, pages 114–124. LNCS **761**, 1993.

[AZ95] T. Arts and H. Zantema. Termination of Logic Programs via Labelled Term Rewrite Systems. In *Proceedings of Computing Science in the Netherlands*, pages 22–34, 1995.

[AZ96] T. Arts and H. Zantema. Termination of Logic Programs using Semantic Unification. In *Proceedings of the Fifth International Workshop on Logic Program Synthesis and Transformation*, pages 219–233. LNCS **1048**, 1996.

[Gan91] H. Ganzinger. Order-Sorted Completion: The Many-Sorted Way. *Theoretical Computer Science* **89**, pages 3–32, 1991.

[GW93] H. Ganzinger and U. Waldmann. Termination Proofs of Well-Moded Logic Programs via Conditional Rewrite Systems. In *Proceedings of the 3rd International Workshop on Conditional Term Rewriting Systems 1992*, pages 113–127. LNCS **656**, 1993.

When Linear Norms Are Not Enough

Nachum Dershowitz [*] Naomi Lindenstrauss [†] Yehoshua Sagiv[†]
Alexander Serebrenik[†‡]

We consider left termination of queries to logic programs ([Apt97, DSD94]). The general approach to proving termination is by showing that during the evaluation of a given program, there is a sequence of terms, such that the size of these terms is decreasing. Usually, orders induced by defining linear norms on terms ([UvG88, Plü90, LS97, CT97]) are sufficient for showing termination. However, in some cases, termination cannot be shown using a linear norm. Two examples are given below.

1. Consider the program for repeated derivations with a query that is symbolically represented in the form `d(Expression, Derivative)`, where the first argument is ground.

```
d(der(t), 1).
d(der(A), 0):-number(A).
d(der(X + Y), DX + DY):-d(der(X), DX), d(der(Y), DY).
d(der(X * Y), X * DY + Y * DX):-d(der(X), DX), d(der(Y), DY).
d(der(der(X)), DDX):-d(der(X), DX), d(der(DX), DDX).
```

2. Consider McCarthy's 91 function [Knu91] with the query `mc_carthy_91(Argument, Value)`, where the first argument is an integer.

```
mc_carthy_91(X, Y)  :-  X > 100, Y is X - 10.
mc_carthy_91(X, Y)  :-  X =< 100, Z1 is X + 11,
                        mc_carthy_91(Z1, Z2), mc_carthy_91(Z2, Y).
```

For programs, such as repetitive derivations, we can show termination by considering *recursively definable sizes,* i.e., the size of a term depends only on its functor and the sizes of its arguments. These recursively definable sizes, which are not necessarily numerical, are a proper extension of the sizes induced by linear norms. For instance, the depth of a term is a recursively definable size, but it cannot be expressed by any linear norm. Norms that are based on recursively definable sizes can be incorporated into the framework of *TermiLog* and similar systems [LSS97, CT97].

For programs, such as McCarthy's 91 function, termination depends on arithmetic calculations, and showing termination is difficult, since the usual order for the integers is not well-founded. For

[*]Tel-Aviv University, Tel-Aviv, Israel, e-mail:nachumd@math.tau.ac.il
[†]Hebrew University, Jerusalem, Israel, e-mail:{naomil,sagiv,alicser}@cs.huji.ac.il
[‡]Contact author

such programs, we have developed the following method for showing termination. First, we deduce automatically a finite abstract domain representing the integers. For example, the integer abstractions for the 91 function consists of the following intervals: $\{X \leq 89, 89 < X \leq 100, X > 100\}$. Using this abstraction, we can apply an abstract interpretation to infer a finite number of atoms abstracting answers of the program. These abstract atoms serve as a basis for extending the technique of the query-mapping pairs [LS97]. Specifically, for each query-mapping pair that is suspected of expressing a non-terminating behavior, we guess a bounded integer-valued termination function. If, while traversing the pair, the termination function decreases monotonically, termination is shown. In our approach, it is sufficient to guess a simple termination function for each suspicious query-mapping pair, and that gives our approach an edge over the classical approach of using a *single* termination function, which inevitably has to be more complicated and is harder to guess automatically. Thus, our approach of combining a finite abstraction of the integers with the technique of the query-mapping pairs is essentially capable of dividing a termination proof into several cases, such that a simple termination function suffices for each case, and the whole process of proving termination can be done automatically in the framework of *TermiLog* and similar systems [LSS97, CT97].

References

[Apt97] Krzysztof R. Apt. *From Logic Programming to Prolog.* Prentice-Hall International Series in Computer Science. Prentice Hall, 1997.

[CT97] Michael Codish and Cohavit Taboch. A semantic basis for termination analysis of logic programs and its realization using symbolic norm constraints. In Michael Hanus, Jan Heering, and Karl Meinke, editors, *Algebraic and Logic Programming, 6th International Joint Conference, ALP '97 - HOA '97*, pages 31–45. Springer Verlag, 1997. Lecture Notes in Computer Science, volume 1298.

[DSD94] Danny De Schreye and Stefaan Decorte. Termination of Logic Programs: The Never-Ending Story. *The Journal of Logic Programming*, 19/20:199–260, May/July 1994.

[Knu91] Donald E. Knuth. Textbook Examples of Recursion. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation. Papers in Honor of John McCarthy*, pages 207–229. Academic Press, Inc., 1991.

[LS97] Naomi Lindenstrauss and Yehoshua Sagiv. Automatic termination analysis of logic programs. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 63–77. MIT Press, July 1997.

[LSS97] Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. *TermiLog*: A system for checking termination of queries to logic programs. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference*, pages 63–77. Springer Verlag, June 1997. Lecture Notes in Computer Science, volume 1254.

[Plü90] Lutz Plümer. *Termination Proofs for Logic Programs.* Lecture Notes in Artificial Intelligence, volume 446. Springer Verlag, 1990.

[UvG88] Jeffrey D. Ullman and Allen van Gelder. Efficient Tests for Top-Down Termination of Logical Rules. *Journal of the Association for Computing Machinery*, 35(2):345–373, April 1988.

On Proving Termination through Transfer Functions

Oliver Theel and Felix C. Gärtner*

Darmstadt University of Technology
Department of Computer Science
D-64283 Darmstadt, Germany

Phone: (+49)-6151-16-5306/3908
Fax: (+49)-6151-16-5410
Email: {theel|felix}@informatik.tu-darmstadt.de

Notions of termination appear in nearly every area of computer science. We have encountered one of these notions in the context of self-stabilizing distributed algorithms. An algorithm is said to be *self-stabilizing* if, starting from any state, it is guaranteed to reach a pre-defined set of legal states in finite time. This is called the *convergence* property of such algorithms. Together with the additional requirement that the set of legal states is closed under normal system operation, this ability can be seen as a very powerful method of achieving fault-tolerance: self-stabilizing algorithms tolerate *any* kind of transient failure in a non-masking way, even those failures which were unforseeable at design time.

Proving convergence of self-stabilizing algorithms is an instance of proving termination, and so all the methods that exist in the literature can be applied. However, the existing methods in computer science seemingly all boil down to a well-foundedness argument: basicly, one has to devise a well-founded ordering relation on system states and then show that in every state transition of the algorithm either a legal state is reached or the transition decreases the "order value" of the states involved. An equivalent proof method is to exhibit a corresponding *termination function* (commonly called *convergence function* in the self-stabilization literature). However, just like in the area of termination, devising such a termination function requires some creativity. In self-stabilization finding such a function is especially difficult since it must consider *all* states of the algorithm and capture the very "essence" of the convergence property.

Currently we are investigating methods to prove termination by adopting results from *control theory*. The objective of this part of electrical engineering is to "stabilize" real-world processes (like motors, power supplies and other electric or electronic devices) by devising special control components around the process (i.e., a control circuit) to guarantee that the process' output will stabilize to

*This author's work was supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the "Graduiertenkolleg ISIA" at Darmstadt University of Technology.

a certain value (like speed, voltage etc.) whenever external influences disturb normal operation.

The central part of proving the stability of such a control circuit is to calculate the input-to-output behavior of the process alone (its so called *transfer function*) and then "plug in" controller components that influence this transfer function in predefined ways. By adding these components, the transfer function of the overall control circuit is changed. Basic theorems of control theory offer a set of sufficient conditions proving that the overall transfer function will converge to some predified value.

The similarities in the problem statement are obvious. No so obvious are the similarities in proof techniques. While the traditional approach of convergence functions uses a state-based reasoning approach based on some form of logic, the control theory approach can be characterized as investigating the limits behavior of the overall transfer function. This involves solving non-linear differential equations, calculating function transformations and dealing with residues and complex numbers. While other proof methods similar to termination functions exist in control theory, the usual approach which we will present in the talk seems to have very little resemblance to the standard well-foundedness approach of computer science. For example, while the traditional termination functions are necessarily monotonic and stop decreasing at some finite point in time, the transfer functions are not necessarily monotonic (i.e., they can oscillate with decreasing amplitude around the target value) and converge only in the limit.

In the talk, we will present the control theory approach as sketched in this abstract. We are interested in obtaining some expert comments from the termination community on the above and some other open questions we have. We plan to illustrate the topic with an example algorithm which can be easily proved correct using the control theory approach but which we have found no easy way of proving using a well-foundedness argument.

Termination proofs and computational complexity classes

J.Y. Marion

Loria, Calligramme project,
B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France
Jean-Yves.Marion@loria.fr.

Abstract

Termination orderings on rewrite systems delineate classes of total functions such as the primitive recursive functions which are characterised by MPO [Hof92], multiple recursive functions [Pét66] which are characterised by LPO [Wei95]. Subrecursive hierarchy classifications has been extensively investigated by assigning ordinals to recursive definitions, see for example [BEW94,HW94]. The main issue is to express the computational complexity of a program from its termination proof. For this, we incorporate ideas involved in predicative analysis of recursive definitions which can be found in the works of [Sim88,BC92,Lei94] and also exploit more traditional methods from termination proofs. We consider here ways of specifying the computational complexity of algorithms which is an intensional property. On the other hand, the traditional meta-theory of program reasoning deals with the verification of extensional properties. The study of intensional properties of programs turns out to be much more challenging. For example, the class of all algorithms running, say, in polynomial time is not recursively enumerable with respect to an enumeration of partial recursive algorithms. But, imposing an intentional property on program constructions is a relevant question which is considered for example in Nuprl development as explained by Constable in [Con95]. One might wonder how relevant is a program whose proof termination is certified but runs in doubly exponential time.

We shall investigate the computational complexity of algorithms provided by rewrite systems with termination proofs based on polynomial and exponential interpretations. Termination proofs by polynomial interpretations were introduced by Lankford [Lan79]. Hofbauer and Lautemann [HL88] established that derivations admit a doubly exponential length. The background for our study really goes back to the work initiated by Cichon and Lescanne [CL92]. It was shown that a particularly important aspect was the interpretation of the constructors. More recently, we showed in [BCMT98] that the functions whose termination proofs involve a particular kind of successor interpretation, are exactly the functions computed in polynomial-time (PTIME), linear exponential time (ETIME $=$ DTIME$(2^{0(n)})$) and linear doubly exponential time (E$_2$TIME $=$ DTIME$(2^{2^{0(n)}})$).

While in [BCMT98] functions were just defined by a confluent system on strings, we now consider general rewrite systems with polynomial interpretations. Data structures are now lists, trees, ... So, we are somewhat closer

to real programming languages such as the ones designed in POLO [Gie95], ORME [Orm], LARCH [Lar]. Next, we present a notion of functions computed by non-confluent systems which appears in Krentel's work [Kre88] and which seems appropriate and robust as discussed by Grädel & Gurevich [GG95]. The whole purpose is to provide a basis whereby non-confluent rewrite systems are construed to non-deterministic computations. As in [BCMT98], we exhibit three kinds of polynomial interpretations of constructors. We analyse the computational complexity of algorithms with respect to the kind of interpretation given to constructors. Hence, we obtain three classes of functions which characterise exactly PTIME, ETIME, E_2TIME when systems are confluent. When systems are non-confluent, we capture the non-deterministic counterpart, that is the class of functions computable in non-deterministic polynomial time (NPTIME), non-deterministic exponential time (NETIME), and lastly non-deterministic doubly exponential time (NE_2TIME).

Machine independant characterisations of complexity classes were originated by Cobham [Cob62]. His approach is by means of 'bounded recursion on notation' in which rates of growth of functions are limited by functions already defined in the class. In contrast, in our work, polynomial interpretations just impose, via a reduction ordering, a local condition on each rewrite rule. In fact, the different kinds of polynomial interpretations of successors are somewhat akin to the notion of data tiering [BC92,Lei94]. Beckmann & Weiermann reformulate those ideas into a rewriting setting, see also [Cas96]. It is also worth mentioning the characterisation of PTIME functions over finite models in [Saz80,Gur83], since basically the same system is considered : the Herbrand-Gödel equations.

We briefly summarise the main results

n-ary constructor interpretation	Confluent	Non-Confluent
Kind 0 : $\sum_{i=1}^{n} X_i + \gamma$	PTIME	NPTIME
Kind 1 : $\sum_{i=1}^{n} \alpha_i X_i + \gamma$	ETIME	NETIME
Kind 2 : $\sum_{i=1}^{n} \alpha_i X_i^{\beta_i} + \gamma$	E_2TIME	NE_2TIME
Kind 3 : $\sum_{i=1}^{n} 2^{X_i} + \gamma$	DTIME($\exp_\infty$(n))	DTIME($\exp_\infty$(n))

$$\text{where } \exp_\infty(0) = 2 \text{ and } \exp_\infty(n+1) = 2^{\exp_\infty(n)}$$

References

[BC92] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.

[BCMT98] G. Bonfante, A. Cichon, J.Y Marion, and H. Touzet. Complexity classes and rewrite systems with polynomial interpretation. In *CSL*, 1998. Technical Report LORIA 98-R-060, `http://www.loria.fr/~marionjy`.

[BEW94] W. Buchholz, E.A.Cichon, and A. Weiermann. A uniform approach to fundamental sequences and hierarchies. *Mathematical Logic Quarterly*, 40, 1994.

[Cas96] V-H Caseiro. An equational characterization of the poly-time functions on any constructor data strucure. Technical Report 226, University of Oslo, Dept. of informatics, December 1996. `http://www.ifi.uio.no/~ftp/publications`.

[CL92] E.A Cichon and P. Lescanne. Polynomial interpretations and the complexity of algorithms. In *CADE'11*, pages 139–147, 1992.

[Cob62] A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science*, pages 24–30. North-Holland, Amsterdam, 1962.

[Con95] R. Constable. Expressing computationnal complexity in constructive type theory. In D. Leivant, editor, *LCC'94*, number 960 in LNCS, pages 131–144, 1995.

[GG95] Erich Grädel and Yuri Gurevich. Tailoring recursion for complexity. *J. Symbolic Logic*, 60(3):952–969, Sept. 1995.

[Gie95] J. Giesl. Generating polynomial orderings for termination proofs. In *RTA*, number 914 in Lecture Notes in Computer Science, pages 427–431, 1995.

[Gur83] Y. Gurevich. Algebras of feasible functions. In *Twenty Fourth Symposium on Foundations of Computer Science*, pages 210–214. IEEE Computer Society Press, 1983.

[HL88] D. Hofbauer and C. Lautemann. Termination proofs and the length of derivations. In *RTA*, number 355 in Lecture Notes in Computer Science, 1988.

[Hof92] D. Hofbauer. Termination proofs with multiset path orderings imply primitive recursive derivation lengths. *Theoretical Computer Science*, 105(1):129–140, 1992.

[HW94] W.G. Handley and S.S. Wainer. Equational derivation vs. computation. *Annals of Pure and Applied Logic*, 70:17–49, 1994.

[Kre88] M. Krentel. The complexity of optimization problems. *Journal of computer and system sciences*, 36:490–519, 1988.

[Lan79] D.S. Lankford. On proving term rewriting systems are noetherien. Technical Report MTP-3, louisiana Technical University, 1979.

[Lar] *Larch*. `http://www.sds.lcs.mit.edu/spd/larch/index.html`.

[Lei94] Daniel Leivant. Predicative recurrence and computational complexity I: Word recurrence and poly-time. In Peter Clote and Jeffery Remmel, editors, *Feasible Mathematrics II*. Birkhäuser, 1994.

[Orm] *Orme*. `http://www.ens-lyon.fr/~plescann/publications.html`.

[Pét66] Rósza Péter. *Rekursive Funktionen*. Akadémiai Kiadó, Budapest, 1966. English translation: *Recursive Functions*, Academic Press, New York, 1967.

[Saz80] Vladimir Sazonov. Polynomial computability and recursivity in finite domains. *Electronische Informationsverarbeitung und Kybernetik*, 7:319–323, 1980.

[Sim88] Harold Simmons. The realm of primitive recursion. *Archive for Mathematical Logic*, 27:177–, 1988.

[Wei95] A. Weiermann. Termination proofs by lexicographic path orderings yield multiply recursive derivation lengths. *Theoretical Computer Science*, 139:335–362, 1995.

Complexity classes within KBO

Guillaume Bonfante

LORIA, Projet Calligramme,
B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France,
bonfante@loria.fr.

Abstract

Termination proofs of rewrite systems using the Knuth-Bendix ordering, KBO, imply multiply-recursive bounds on the lengths of derivations. Hofbauer has studied two restrictions of KBO. The first is a restriction to the use of positive weights, the second is a restriction on the signature which is only allowed to contain at most unary symbols. In both cases, Hofbauer proves the derivation lengths to be bounded exponentially. We prove that in the first case computations are LINSPACE and that in the second computations are ETIME.

The theme of our investigations is that of complexity characterisations of classes of functions computed by rewrite systems. Comparisons are achieved with respect to computations with a Turing Machine. Actually, we concentrate on classes of feasible complexity.

There are several examples where a fine analysis of termination proofs gives upper bounds on the complexity of the computed function. Concerning path orderings for instance, see Hofbauer [Hof92] for the case of the *multiset path ordering*, MPO, and Weiermann [Wei95] for the case of the *lexicographic path ordering*, LPO. In both cases, the authors answer the question as to what complexity restrictions are imposed by the termination proof method.

We are concerned here with a similar point of view. Given a termination proof method and a complexity class, are all functions of this class computed by a rewrite system whose termination can be established by the method? Following on from work of Cichon and Lescanne in [CL92], Bonfante, Cichon, Marion and Touzet, in [BCMT98], have analysed the situation for termination proofs using polynomial interpretations in terms of time complexity classes. We consider the same problem for the Knuth-Bendix ordering, KBO.

KBO is a syntactic ordering on terms invented by Knuth and Bendix [KB70] in the late 1960's and it has been proved decidable whether or not a rewrite system admits a KBO ordering termination proof ([Lan79, Mar87, Alt89]). As a consequence, KBO can be used by automatic proof search systems.

In [HL89], Hofbauer shows that the upper bounds on derivation lengths of rewrite systems with a KBO termination proof are not primitive recursive. In fact, they are multiply-recursive [Hof9-, Tou97]. However, there are two natural restrictions of the order for which the bounds are exponential. The first requires that the weights be strictly positive (which gives a restriction on size) and the second restricts to computations on words (or strings).

¿From our point of view, these two restrictions lead to very different classes of functions. We show that the restriction on weights leads to a characterisation of LINSPACE and that the restriction on the signature leads to a characterisation of ETIME.

References

[Alt89] E. Altendorf. Termination von Termersetzungssystemen: Theorische Untersuchungen und Implementierungen von KBO, RPO und KNS. PhD thesis, TU Berlin, 1989.

[BCMT98] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet. Complexity classes and rewrite systems with polynomial interpretation. In CSL'98, Brno, Czech Republic, 1998.

[CL92] E.A. Cichon and P. Lescanne. Polynomial Interpretations and the Complexity of Algorithms. Proceedings of the 11^{th} International Conference on Automated Deduction. Springer Lecture Notes in Artificial Intelligence 607, 1992.

[HL89] D. Hofbauer and C. Lautemann. Termination Proofs and the Length of Derivation. In 3rd RTA, volume 355 of Lecture Notes in Computer Science, pages 167–177, 1989.

[Hof9-] D. Hofbauer. Termination Proofs and Derivation Lengths in Term Rewriting Systems. PhD thesis, Technische Universität Berlin.

[Hof92] D. Hofbauer. Termination proofs with multiset path orderings imply primitive recursive derivation lengths. Theoretical Computer Science, 105(1):129–140, 1992.

[KB70] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. Computational problems in abstract algebra. Ed. J. Leech, Pergamon, 1970.

[Lan79] D.S. Lankford. On proving term rewriting systems are nœtherian. Technical report, Louisiana Technical University, 1979.

[Mar87] U. Martin. How to chose the weights in the Knuth-Bendix ordering. In 2nd RTA, volume 256 of Lecture Notes in Computer Science, pages 42–53, 1987.

[Tou97] H. Touzet. Propriétés combinatoires pour la terminaison de systèmes de réécriture. PhD thesis, Université Henri Poincaré - Nancy 1, September 1997.

[Wei95] A. Weiermann. Termination proofs by lexicographic path orderings yield multiply recursive derivation lengths. Theoretical Computer Science, 139(1):355–362, 1995.

Recursive Upper-Bounds for Overlay Term Rewrite Systems

Elias TAHHAN BITTAR
Dpto. de Matemáticas Púras y Aplicadas.
Universidad Simón Bolívar
Caracas, VENEZUELA.
etahhan@usb.ve
http://thor.ma.usb.ve/∼etahhan/

January 30, 1999

In [Gra], Gramlich extended an O'Donnell's result by proving that finite innermost terminating overlay term rewrite systems with joinable critical pairs are terminating. We study the complexity aspect of this result and we prove that :

Theorem Given a finite overlay term rewrite system $\mathcal{R}$ and a term $t = f(u_0, \ldots, u_m)$ such that :

- each immediate sub-term u_i of t is strongly uniquely normalising (we denote $\downarrow(u_i)$ its unique normal form),
- and if $\downarrow_i(t) := f(\downarrow(u_0), \ldots, \downarrow(u_m))$ is strongly uniquely normalising,

then t is strongly uniquely normalising and there is a constant M such that

$$\lambda(t) \leq \lambda(\downarrow_i(t)) + M^{\lambda(\downarrow_i(t))} \times (\lambda(u_0) + \cdots + \lambda(u_m))$$

where $\lambda(v)$ denotes the length of a worst derivation of a strongly normalising term v.

This theorem implies Gramlich result stated above and extends a result proved in [Tah] for constructor orthogonal rewrite systems.

This result allows to estimate upper bounds for primitive recursive functions and to give a syntactic characterisation of the Grzegorczyk hierarchy as done in [Cich-Tah].

References

[Gra] B. Gramlich. Relating innermost, weak, uniform and modular termination of term rewriting systems. LNAI, pp 285–296. Springer-Verlag, 1992.

[Cich-Tah] E. A. Cichon & E. Tahhan Bittar. Strictly orthogonal left linear rewrite systems and primitive recursion. Submitted 1998.

[Tah] E. TAhhan Bittar. Bornes supérieures de terminaison de sistèmes de réécriture strictement orthogonaux. Prépublication 17, LLAIC1. Université d'Auvergne (FRANCE) 1993.

List of Participants

Andreas Abel

LMU München
TCS
Knappertsbuschstr. 21
81927 München
Germany

Phone: +49 89 93930657
abel@informatik.uni-muenchen.de

Elias Tahhan Bittar

Universidad Simon Bolivar
Dpto. de Matematicas
Caracas Sartenejas, Baruta
P.O. Box 89000, Caracas 1080-A
Venezuela

Phone: +58 2 9063282
Fax: +58 2 9063278
etahhan@usb.ve

Thorsten Altenkirch

LMU München
Oettingenstr. 67
D-80538 München
Germany

Phone: +49 89 2178 2209
Fax: +49 89 2178 2238
alti@informatik.uni-muenchen.de

Frederic Blanqui

LRI
Bat 490
Universite Paris 11
F-91405 Orsay Cedex
France

Phone: +33 1 69 15 64 85
Fax: +33 1 69 15 65 86
Frederic.Blanqui@lri.fr

Jamie Andrews

University of Western Ontario
Dept. of Computer Science
London, Ontario
N6A 5B7
Canada

Phone: (519)679-2111 6856
Fax: (519)661-3515
andrews@csd.uwo.ca

Guillaume Bonfante

LORIA Equipe Calligramme
BP 239
Villers-les-Nancy 54506
France

Phone: +33 3 83 59 20 22
bonfante@loria.fr

Thomas Arts

Ericsson Utvecklings AB
Box 1505
125 25 Älvsjö
Sweden

Phone: +46 8 719 9514
Fax: +46 8 719 8988
thomas@cslab.ericsson.se

Cristina Borralleras

University of Vic
Sagrada Familia 7
08500 - Vic
Barcelona
Spain

Phone: 93-8861222
Fax: 93-8856900
cristina.borralleras@uvic.es

Adam Cichon

LORIA-UHP
BP 239
Vandoeuvre-les-Nancy Cedex
France

Phone: +33 3 83 59 20 20
Fax: +46 8 719 8988
`cichon@loria.fr`

Arne John Glenstrup

DIKU
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen O
Denmark

Phone: 35 32 14 00
Fax: 35 32 14 01
`panic@diku.dk`

Evelyne Contejean

LRI
Bat 490
Universite Paris 11
F-91405 Orsay Cedex
France

Phone: +33 1 69 15 64 76
Fax: +33 1 69 15 65 86
`Evelyne.Contejean@lri.fr`

Bernhard Gramlich

Technische Universität Wien
Resselgasse 3
A-1040 Wien
Austria

Phone: +43 1 58801 18544
Fax: +43 1 58801 18597
`gramlich@logic.at`

Nachum Dershowitz

Tel-Aviv University
Ramat-Aviv 69978
Israel

Phone: 972-3-640-9621
Fax: 972-3-640-9357
`nachumd@cs.tau.ac.il`

Dieter Hofbauer

Universität GH Kassel
Fachbereich 17, Mathematik/Informatik
34109 Kassel
Germany

Phone: +49 29 561 804 4481
Fax: +49 29 561 804 4008
`dieter@theory.informatik.uni-kassel.de`

Jürgen Giesl

Dept. of Computer Science
Darmstadt University of Technology
Alexanderstr. 10
64283 Darmstadt
Germany

Phone: +49 6151 164494
Fax: +49 6151 166214
`giesl@informatik.tu-darmstadt.de`

Ingo Lepper

Westfälische Wilhelms-Universität Münster
Institut für mathematische Logik
Einsteinstraße 62
48149 Münster
Germany

Phone: +49 29251-833 37 67
Fax: +49 29251-833 30 78
`lepper@uni-muenster.de`

Pierre Lescanne

Ecole Normale Superieure de Lyon
46 allee d'Italie
F69364 Lyon Cedex 07
France

Phone: +33 4 72 72 86 83
Fax: +33 4 72 72 80 80
Pierre.Lescanne@ens-lyon.fr

Ursula Martin

University of St Andrews
Computer Science Department
St Andrews, Fife KY16 6SS,
Scotland

Phone: +44-1334-463252
Fax: +44-1334-463278
um@dcs.st-and.ac.uk

Naomi G. Lindenstrauss

Hebrew University
Dept of Computer Science
Givat Ram
Jerusalem
Israel

Phone: 972-2-6522762
Fax: 972-2-6585439
naomil@cs.huji.ac.il

Aart Middeldorp

University of Tsukuba
Institute of Information Sciences and Electronics
Tsukuba 305-8573
Japan

Phone: +81 298 53 5538
Fax: +81 298 53 5206
ami@is.tsukuba.ac.jp

Claude Marche

LRI
Bat 490
Universite Paris 11
F-91405 Orsay Cedex
France

Phone: +33 1 69 15 64 85
Fax: +33 1 69 15 65 86
Claude.Marche@lri.fr

Monica Nesi

Universita degli Studi di L'Aquila
Dipartimento di Matematica Pura ed Applicata
Via Vetoio
67010 Coppito (L'Aquila)
Italy

Phone: +39 0862 433728
Fax: +39 0862 433180
monica@univaq.it

Jean-Yves Marion

Loria Universite Nancy 2
BP 239
54506 Vandoeuvre les Nancy
France

Phone: +33 3 83 59 20 18
Fax: +33 3 83 27 83 19
Jean-Yves.Marion@loria.fr

Enno Ohlebusch

Universität Bielefeld
Technische Fakultät
AG Praktische Informatik
Postfach 10 01 31
D-33501 Bielefeld
Germany

Phone: +49 521 106 2907
Fax: +49 521 106 6411
enno@TechFak.Uni-Bielefeld.DE

Hitoshi Ohsaki

Electrotechnical Laboratory
Rewriting Group
Tsukuba 305-8568
Japan

Phone: +81 298 54 5890
Fax: +81 298 54 3342
ohsaki@etl.go.jp

Alexander Serebrenik

Hebrew University of Jerusalem
Institute of Computer Science
Jerusalem 91904
Israel

Phone: 02-6586776
alicser@cs.huji.ac.il

Laurence Puel

LRI
Bat 490
Universite Paris-Sud
F-91405 Orsay Cedex
France

Phone: +33 1 69 15 66 37
puel@lri.fr

Jan-Georg Smaus

University of Kent at Canterbury
Computing Lab
Canterbury
Kent CT2 7NF
United Kingdom

Phone: +44 1227 827553
Fax: +44 1227 762811
jgs5@ukc.ac.uk

Anabela Lopes Ribeiro

Universidade Nova de Lisboa
Dep. Informatica
Faculdade de Ciencias e Tecnologia
Quinta da Torre
2825-114 Caparica
Portugal

Phone: +351 1 2948536
Fax: +351 1 2948541
ar@di.fct.unl.pt

Oliver Theel

TU Darmstadt
Department of Computer Science
Alexanderstraße 10
64283 Darmstadt
Germany

Phone: +49 6151-16-5306
Fax: +49 6151-16-5410
theel@informatik.tu-darmstadt.de

Albert Rubio

Universitat Politecnica de Catalunya
Dept. LSI
Modul C6
C/Jordi Girona, 1-3
08034 Barcelona
Spain

Phone: +34 93 4017988
Fax: +34 93 4017014
rubio@lsi.upc.es

Xavier Urbain

LRI
Bat 490
Universite Paris 11
F-91405 Orsay Cedex
France

Phone: +33 1 69 15 64 85
Fax: +33 1 69 15 65 86
Xavier.Urbain@lri.fr

Sofie Verbaeten

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200 A
3001 Heverlee
Belgium

Phone: +32 280 2916-327555
Fax: +32 280 2916-327996
sofie.verbaeten@cs.kuleuven.ac.be

Andreas Weiermann

Universität Münster
Einsteinstr. 62
48149 Münster
Germany

Phone: +49 251 8333762
Fax: +49 251 8333078
weierma@math.uni-muenster.de

Fer-Jan de Vries

Electrotechnical Laboratory
Rewriting Group
Tsukuba 305-8568
Japan

Phone: +81 298 54 3382
Fax: +81 298 54 3342
ferjan@etl.go.jp

Hans Zantema

Utrecht University
Department of Computer Science
Padualaan 14
3508 TB Utrecht
The Netherlands

Phone: 3130 2534116
Fax: 3130 2513791
hansz@cs.uu.nl

Johannes Waldmann

Universität Leipzig
Institut für Informatik
Augustusplatz 10
04109 Leipzig
Germany

Phone: +49 280341 29 97 32 204
Fax: +49 280341 29 97 32 209
joe@informatik.uni-leipzig.de

International Workshop on Termination (WST '99)

Monday, May 10, 1999

Time			*Speaker/Chair*	*Title*
8:50	–	9:00	*Jürgen Giesl*	Opening of WST '99
9:00	–	9:45	Hans Zantema	The termination hierarchy for term rewriting (Tutorial talk)
9:45	–	10:15	Nachum Dershowitz	Undecidability results that follow from results in recursion theory
10:15	–	10:40	Break	
			Nachum Dershowitz	
10:40	–	11:10	Jamie Andrews	Termination semantics of logic programs with cut and related features
11:10	–	11:40	Sofie Verbaeten & Danny De Schreye	Termination of simply moded well-typed logic programs under tabled execution mechanism
11:40	–	12:10	Jan-Georg Smaus	Well-terminating, input-driven logic programs
12:10	–	13:40	Lunch	
			Hans Zantema	
13:40	–	14:10	Aart Middeldorp & Jürgen Giesl	Transforming context-sensitive rewrite systems
14:10	–	14:40	Maria Ferreira & Anabela Lopes Ribeiro	Context-sensitive AC-rewriting
14:40	–	15:10	Albert Rubio	A fully syntactic AC-RPO
15:10	–	15:40	Cristina Borralleras, Maria Ferreira & Albert Rubio	Monotonic semantic path orderings
15:40	–	16:20	Break	
			Bernhard Gramlich	
16:20	–	16:50	Jean-Pierre Jouannaud & Albert Rubio	Higher-order recursive path orderings
16:50	–	17:20	Frederic Blanqui, Jean-Pierre Jouannaud & Mitsuhiro Okada	A terminating schema for higher-order rewrite systems
17:20	–	17:50	Andreas Abel & Thorsten Altenkirch	A semantical analysis of structural recursion
18:00			Dinner	

International Workshop on Termination (WST '99)

Tuesday, May 11, 1999

Time			*Speaker/Chair*	*Title*
			Adam Cichon	
8:45	–	9:30	Dieter Hofbauer	On termination and derivation lengths for ground rewrite systems (Tutorial talk)
9:30	–	10:00	Andreas Weiermann	A slow growing analysis of the canonical rewrite system for the Ackermann function
10:00	–	10:30	Ingo Lepper	A totally terminating rewrite system whose complexity is not Γ_0 recursive
10:30	–	10:45	Break	
			Pierre Lescanne	
10:45	–	11:15	P. Inverardi & Monica Nesi	Translating TRSs into OS-TRSs: A result on modularity of termination
11:15	–	11:45	Jürgen Giesl, Thomas Arts & Enno Ohlebusch	Modularity results for termination proofs using dependency pairs
11:45	–	12:15	Thomas Arts & Jürgen Giesl	Verification of Erlang processes
12:15	–	13:40	Lunch	
13:40	–	22:00	Excursion	

International Workshop on Termination (WST '99)

Wednesday, May 12, 1999

Time	*Speaker/Chair*	*Title*
	Aart Middeldorp	
9:00 – 9:45	Ursula Martin	On assigning invariants to term orderings (Tutorial talk)
9:45 – 10:15	Bernhard Gramlich	On some abstract termination criteria
10:15 – 10:40	Break	
	Ursula Martin	
10:40 – 11:10	Johannes Waldmann	Top termination of $CL(S)$
11:10 – 11:40	Jürgen Giesl, Vincent von Oostrom & Fer-Jan de Vries	Strong convergence of term rewriting using strong dependency pairs
11:40 – 12:10	Hitoshi Ohsaki, Aart Middeldorp & Jürgen Giesl	Equational termination by semantic labelling
12:10 – 14:00	Lunch	
	Thomas Arts	
14:00 – 14:30	Enno Ohlebusch	Automatic termination proofs of logic programs via rewrite systems
14:30 – 15:00	Nachum Dershowitz, Naomi Lindenstrauss, Yehoshua Sagiv & Alexander Serebrenik	When linear norms are not enough
15:00 – 15:30	Oliver Theel & Felix Gärtner	On proving termination through transfer functions
15:30 – 16:20	Break	
	Dieter Hofbauer	
16:20 – 16:50	Jean-Yves Marion	Termination proofs and computational complexity classes
16:50 – 17:20	Guillaume Bonfante	Complexity classes within KBO
17:20 – 17:50	Elias Tahhan Bittar	Recursive upper-bounds for overlay term rewrite systems
18:00	Dinner	