Extended Abstracts of the Fifth International Workshop on Termination (WST '01)

Nachum Dershowitz, editor

Utrecht, May 20-21, 2001

Logic Group
Preprint Series
No. 209
May 2001

ZENO Institute of Philosophy

ISBN 90-393-2765-3
ISSN 0929-0710
Prof.dr. A. Visser, Editor

Extended Abstracts of the

Fifth International Workshop on Termination

(WST '01)

Utrecht, The Netherlands

Nachum Dershowitz, editor

May 20–21, 2001

Contents

Preface

After the successful international workshops on termination held in St. Andrews (1993), La Bresse (1995), Ede (1997), and Dagstuhl (1999) a fifth workshop was held in Utrecht, on the campus of Universiteit Utrecht, in conjunction with the *Twelfth International Conference on Rewriting Techniques and Applications* (RTA 2001) and the *Fourth International Workshop on Explicit Substitutions: Theory and Applications to Programs and Proofs* (WESTAPP 2001).

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

Areas of interest to this workshop, include, but are not limited to, the following:

Well-quasi-order theory and ordinal notations
Ordinals and termination orderings
Fast and slow growing hierarchies
Strong normalization of lambda calculi
Termination of programs, of rewriting, and of logic programs
Hard termination problems and proofs
Termination methods for theorem provers and verification systems
Implementations and applications of termination methods

The program committee for the fifth workshop consisted of

Nachum Dershowitz, Tel-Aviv (chair)
Danny De Schreye, Leuven
Jürgen Giesl, Aachen
Pierre Lescanne, Lyon
Albert Rubio, Barcelona
Stephen Simpson, Pennsylvania
Hans Zantema, Eindhoven

The local arrangements chair was Vincent van Oostrom. Sponsors of the events included Centrum voor Wiskunde en Informatica, Instituut voor Programmatuurkunde en Algoritmiek, Department of Philosophy at Universiteit Utrecht, International Federation for Information Processing, Leiden-Utrecht Research Institute, University of Amsterdam Informatics Institute, and University of Tsukuba.

Nachum Dershowitz
Tel Aviv

Apparent Causality for Distributed Termination Detection

Farhad Arbab
email: farhad@cwi.nl
CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Termination detection in a system of distributed processes is a classical problem in distributed computing. Distributed Termination Detection (DTD) has been extensively studied in the past twenty years and it is known to be a difficult problem to solve efficiently, because it involves properties of the global state of a distributed system. DTD is an especially important basic form of coordination. Appropriately general DTD algorithms can also be adapted for certain other coordination tasks in parallel and distributed systems, such as barrier synchronization, consensus, etc.

Many DTD algorithms exist and a recent survey of 35 of them [1] introduces a taxonomy and identifies 8 different characteristics for their classification and evaluation. This survey concludes by remarking that "[an] algorithm [with favorable ranking in all 8 dimensions,] if one exists, would be a huge development in this field." As difficult as DTD is in its classical setting, considerations for dynamicity and mobility in a distributed system further complicate the DTD problem and render most existing DTD algorithms non-applicable.

It is customary to assume that the only means of communication among the processes in a distributed system is message passing. Furthermore, it is customary to distinguish between *normal messages* exchanged among the processes in a system, and the *control messages* required by a DTD algorithm. One of the most interesting properties of a DTD algorithm is the maximum number of control messages it requires to detect the termination of a (run of a) system in which a total of n normal messages are exchanged. This measure is sometimes called the *message complexity* of a DTD algorithm. The message complexity of reasonably general DTD algorithms for a system consisting of m processes exchanging n normal messages is typically of the order of $m \times n$. Furthermore, the average performance of such algorithms is not significantly better than their worst case.

We introduce the notion of *Apparent Causality* as a relation among the messages in a system of distributed processes, from which we derive the concepts of message *histories* and *futures*. Apparent causality and message histories are inherently local properties which can be evaluated at the level of each process, whereas message futures are inherently global system-level properties. Histories and futures of messages are examples of histories and futures of more general observables in a distributed system. We propose Back To The Future (BTTF) as a generic method for computing futures from histories, and use this technique to construct three different symmetric algorithms:

1. BTTF Transitory Quiescence (BTTF-TQ) is a generic, efficient algorithm that leads a distributed system to a state containing the distributed knowledge that there are no pending messages;
2. Yet Another Wave Algorithm (YAWA) uses the BTTF technique to implement a generic DTD wave algorithm with certain interesting properties of its own; and
3. BTTF Wave is our main algorithm, which combines BTTF-TQ and YAWA to obtain a general symmetric DTD algorithm that is equally suitable for classical settings as for dynamic systems of distributed mobile processes.

A key concept that underlies all three algorithms is the fact that the real cost of communication in a distributed computing system is essentially a function of the number of transmitted messages. In other words, within "reasonable limits" sending a longer message costs the same as sending a shorter one. Furthermore, the occasion that a message uses the full capacity allowed by those "reasonable limits" for each transmission in a system is indeed very rare. Intuitively, our algorithms take advantage of this fact to reduce the number of control messages they need. Our algorithms utilize the unused capacity that is

collectively provided by the transmission of all normal messages in a system to piggy-back and transmit a good part of the control information they need to run.

Apparent causality is the key relation that creates this control information in terms of message histories. The BTTF scheme ensures that message histories reach a select subset of message senders, who use this information to reconstruct the futures of their respective messages. The significance of message futures is in the fact that they reveal certain global properties of a distributed system. The BTTF-TQ algorithm uses message futures to extract (partial) information about quiescence in a distributed system. The YAWA algorithm uses message futures to determine the coverage of the rounds of its token waves. The BTTF Wave algorithm uses message futures for both of these purposes in order to reduce the number of rounds of token waves as compared with YAWA.

Messages A message has a unique identity and contains some information. A message is sent by a single unique process (its producer or sender) to a single unique process (its target or receiver). The most important information contained in a message is its *value*, which is what the sender intends to communicate to the receiver. A message may contain extra information in addition to its value. Specifically, we require each message to also carry its own identifier and (a part of) its own history.

The value of a message is either *normal data* or *control data*. Normal data is the set of all application data values exchanged through messages. Messages that contain normal data as their values are called *normal messages*. Control data is the set of values required by a DTD algorithm to be exchanged through messages. Messages that contain control data as their values are called *control messages*. A distinguished subset of control data is called *token values* and messages that contain such values are called *tokens*.

Apparent Causality Consider the externally observable behavior of a process, P, in a parallel or distributed system. Every message has a unique sender and a unique receiver. We can observe that at certain points in time P actually reads (i.e., consumes) one of the messages it has received, and we can also see that at certain other points in time, P sends messages to one (single-cast) or more (multi-cast) processes in the system. Taking P as a black-box, there is no way for us to know what is the true relationship between any of its input and output messages, or indeed if they are related at all. Nevertheless, we can discern a certain *apparent causal* relationship among the input and output messages of P, expressed in terms of a formal relation called *Apparent Causal Precedence* (ACP).

Intuitively, every time a process P produces a message m_i, we associate a (possibly empty) set of messages, m_j, $j \geq 0$, that P has read during the period ending with the production of m_i, as the apparent causal precedents of m_i. The decisions about the starting point of the relevant period and exactly which m_j messages read during this period are related to m_i as its apparent causal precedents, both depend on the type of m_i. If m_i is a token, then the period starts at the time when P sent its last token, or the creation time of P. The messages that are related to m_i, in this case, are all the token messages read during this period. If m_i is not a token message, then the period starts at the time when P sent its last non-token message, the last time P became passive, or the creation time of P. The messages that are related to m_i, in this case, are all normal messages read during this period.

A message that has no apparent causal precedent is called an *initial message*. A message that is the apparent causal precedent of only non-token control messages is called a *final message*.

Causal Chains A sequence of messages m_i, for $i \geq 1$, such that $m_i \prec m_{i+1}$, is called a *causal chain* (of m_1). A precondition for (proper) termination in a distributed system is that after a certain point in time, no process sends a message. Thus, we are interested only in systems wherein all causal chains are finite.

A message can be the apparent causal precedent of more than one other message. Therefore, a single message can be the head of more than one causal chain of messages. Because a causal chain cannot contain a cycle, the set of all causal chains of all messages in a system forms a directed acyclic graph.

Histories Consider the set W of the maximal length causal chains of all messages in a run of a system that end with a particular message, m. Other causal chains that end with m are shorter, are completely overlapped by the longer chains in W, and contain only redundant information. The set of longest non-empty proper prefixes of chains in W is called the history of the message m. In other words, the history of m consists of all non-empty chains that result from chopping m off the end of all chains in W. Observe that (only) initial messages have empty histories and that every chain in the history of a message always starts with an initial message.

A process that never produces an initial message is called a **reactive process**. A process that produces one or more initial messages is called a **pro-active process**.

Futures Consider the set W of the maximal length causal chains of a particular message, m. Other causal chains of m are shorter, are completely overlapped by the longer chains in W, and contain only redundant information. The set of longest non-empty proper suffixes of chains in W is called the future of the message m. In other words, the future of m consists of all non-empty chains that result from chopping m off the beginning of all chains in W. Observe every chain in the future of a message always ends with a final message.

Back To The Future The future of a message contains interesting global information for DTD. For instance, information about the possibility of pending messages and the coverage of a token wave can be extracted from message futures. Because the futures of the messages of reactive processes are totally subsumed by the futures of the messages of pro-active processes, a DTD algorithm needs to be concerned only with pro-active processes. Furthermore, because the futures of initial messages subsume the futures of non-initial messages, a DTD algorithm needs to be concerned with the futures of initial messages only.

While apparent causality is a local property and message histories can be computed incrementally and locally, the futures of initial messages are not readily available to their senders. Back To The Future is a simple scheme through which the future of an initial message can be derived from the histories of other messages. This scheme works by ensuring that every chain in the future of an initial message ends with a (control) message (if necessary) sent to the sender of that initial message.

Reconstruction of the Future The future of a message is represented as a tree. As the relevant chains in the histories of other messages become available to the sender of an initial message, it dynamically reconstructs the tree representation of the future of that initial message by *grafting* those chains onto this tree. Trees that represent complete futures are easily recognizable in this scheme. When no pro-active process has an incomplete future tree, there can be no pending messages in the system. When the future tree of an initial token is complete, the token wave initiated by that token has covered all reachable processes and its round is complete.

Implementation and Analysis The implementation of the above-mentioned three algorithms provides an library of interface functions for the computation layer of a distributed application. The primary functions in this library support (multi-cast) send and receive operations for inter-process communication. All aspects of the DTD protocol are transparently handled by these high-level functions which isolate the application code from the details of the DTD algorithm.

The BTTF-TQ algorithm is best regarded as a more control-message-efficient replacement for the commonly used explicit acknowledgment scheme. In fact, in a distributed system where n messages are exchanged asynchronously and their order is not preserved, reaching a state wherein (transient) quiescence is detectable requires n control messages in the worst case. In the same setup, BTTF-TQ is an optimal algorithm that requires only n_c control messages, where $0 \leq n_c \leq \frac{f}{l} \times n, \quad 0 < \frac{f}{l} \leq 1, \quad \log f \leq \frac{\log n}{l-1}$, f is the average frequency of a message in history chains, and l is the average length of a history chain in a final message.

While not intended as a general DTD algorithm, in some special cases (that are not so uncommon in practice) BTTF-TQ is sufficient to detect termination. In these special cases, the message complexity of BTTF-TQ is dramatically superior to other DTD algorithms.

Because the YAWA algorithm relies only on reconstructed (token) message futures to determine the progression of its rounds of token waves, it is independent of any assumptions about the topology of a distributed system. Specifically, this topology can even undergo drastic dynamic changes (e.g., due to mobility) without affecting the execution of YAWA.

The BTTF Wave algorithm ranks quite favorably in the characterization scheme of [1]. Furthermore, it is generic and is suitable for dynamic and mobile systems at no extra cost. The message complexity of this algorithm is $n_c \leq \frac{f}{l} \times n + 2 \times n_r$, where f and l are as above and n_r is the number of wave rounds, which in the worst case, is of the order of m^2 for a system of m processes.

1 References

[1] J. Matocha and T. Camp, "A taxonomy of distributed termination detection algorithms," *The Journal of Systems and Software*, pp. 207–221, 1998.

A monotonic Higher-Order Semantic Path Ordering

Cristina Borralleras[1] and Albert Rubio[2]

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

There is an increasing use of higher-order rewrite rules in many programming languages and logical systems. As in the first-order case, termination is a fundamental property of most applications of higher-order rewriting. Thus, there exists a need to develop for the higher-order case the kind of semi-automated termination proof techniques that are available for the first-order case.

There have been several attempts at designing methods for proving strong normalization of higher-order rewrite rules based on ordering comparisons. Recently, in [JR99], Dershowitz's recursive path ordering has been extended to a higher-order setting by defining a higher-order recursive path ordering (HORPO) on terms of a typed lambda-calculus generated by a signature of polymorphic higher-order function symbols. This ordering is powerful enough to deal with many non-trivial examples and can be automated. HORPO is the first method which operates on arbitrary higher-order terms, therefore applying to higher-order rewriting based on plain pattern matching, where β-reduction is considered as any other rewrite rule. Furthermore, HORPO can operate as well on terms in η-long β-normal form, and hence it provides as well a method for provig strong normalization of higher-order rewriting "à la Nipkow", based on higher-order pattern matching modulo $\beta\eta$.

However, HORPO inherits the same weaknesses that RPO has in the first-order case. RPO is a *simplification ordering* (a monotonic ordering including the subterm relation), which extends a precedence on function symbols to an ordering on terms. It is simple and easy to use, but unfortunately, it turns out, in many cases, to be a weak termination proving tool. First, there are many term rewrite systems (TRSs) that are terminating but are not contained in any simplification ordering, i.e. they are not *simply terminating*. Second, in many cases the head symbol does not provide enough information to prove the termination of the TRS. Therefore, since HORPO follows the same structure and the same use of a precedence as in RPO (in fact, it reduces to RPO when restricted to first-order case), it is easy to expect that similar weaknesses will appear when proving termination of higher-order rewriting.

To avoid this weakness, in the first-order case, many different so-called *transformation methods* have been developed. By transforming the TRS into a set of ordering constraints, the Arts and Giesl's *dependency pair* method has become a successful general technique for proving termination of (non-simply terminating) TRSs.

As an alternative to transformation methods, more powerful term orderings, like Kamin and Levy's *semantic path ordering* (SPO), can be used. SPO generalizes RPO by replacing the precedence on function symbols by any (well-founded) underlying (quasi-)ordering involving the whole term and not only its head symbol. Although the simplicity of the presentation is kept, this makes the ordering much more powerful. Unfortunately, SPO is not so useful in practice, since, although it is well-founded, it is not, in general, monotonic. Hence, in order to ensure termination, apart from checking that the rules of the rewrite system are included in the ordering, in addition the monotonicity for contexts of the rewrite rules has to be proved.

In a recent work [BFR00], a monotonic version of SPO, called MSPO, has been presented. MSPO overcomes the weaknesses of RPO, is automatable and it is shown to generalize other existing transformation methods.

Due to the fact that RPO and SPO share the same "path ordering nature", our aim is to obtain for SPO and MSPO the same kind of extensions to the higher-order case as it was done for RPO.

In this work we introduce the *higher-order semantic path order* (HOSPO), which generalizes HORPO by replacing the precedence on algebraic function symbols by any underlying (quasi-)ordering on higher-order terms. Then a monotonic version of HOSPO, called MHOSPO, is obtained, which provides, a powerful method for proving termination of higher-order rewriting. The ingredients for MHOSPO are still (quasi-)ordering on higher-order terms, but to ensure that MHOSPO is adequate to prove termination (i.e. it is a higher-order reduction ordering) several properties on these quasi-orderings have to be required. In order to make MHOSPO useful in practice, and even make it automatable, we have analyzed possible cadidates for these ingredients. To illustrate the power of the resulting method several non-trivial examples, which cannot be proved by any HORPO, are shown to be terminating.

References

[BFR00] C. Borralleras, M. Ferreira, and A. Rubio. Complete monotonic semantic path orderings. In David McAllester, ed., *Proc. of the 17th Int. Conf. on Automated Deduction (CADE-17)*, LNAI 1831, pp. 346–364, Springer-Verlag, 2000.

[JR99] J.-P. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *14th IEEE Symposium on Logic in Computer Science (LICS)*, pp. 402–411, 1999.

Strong normalization by reducibility of the weak λ_P calculus

Julien FOREST
LRI (CNRS UMR 8623)
Bât 490, Université Paris-Sud
91405 Orsay Cedex
France
e-mail:forest@lri.fr

Pattern calculi [2, 3] were proposed as a theoretical modelization of programming languages with function definitions by cases using pattern matching such as CAML [1], Haskell [6], ML [7], etc.
In this paper we study strong normalization of a new pattern calculus called λ_P. The main differences between terms of λ_P and those of classical λ-calculus are:

- The construction $\lambda x.M$ of the λ-calculus is replaced by $\lambda P.M$ where P is a pattern which belongs to a given grammar including a notion of choice between two different patterns to denote patterns of sum types. Thus for example one can write terms such as $A = \lambda(nil \mid_\xi cons(h,t)).[nil|_\xi t]$ which denotes a function which takes a list and returns another list: when the argument is an empty list nil then the function A returns an empty list nil, and when A is applied to a non-empty list $cons(h,t)$, it returns the tail t of this list $cons(h,t)$.
- A set of variables (called choice variables) dealing with the notion of choice between two patterns is added. For example, the variable ξ of the previous example A belongs to this new set.

The main differences between the pattern calculi in [2, 3] and λ_P are:

- λ_P treats explicitly both pattern matching and substitutions (including substitutions generated by the choice of patterns).
- Our grammar is much more in the classical lambda calculus style.
- A system of substitution with concatenation (such as the one used in λ_σ [4]) is introduced to preserve confluence.

The calculus λ_P is weak (i.e. substitutions can not cross lambdas [5]) and is a typed calculus which verifies subject reduction and confluence.
The λ_P calculus can be seen as a generalization of the well-known weak λ_σ calculus (but in named notation) denoted here $\lambda_{\sigma w}$. For this reason we first

give a proof of strong normalization for $\lambda_{\sigma w}$, which is then extended to deal with patterns, thus obtaining a proof of strong normalization for λ_P.
The proof for $\lambda_{\sigma w}$ is inspired by the proof of termination of Ritter [8] which is based on the reducibility technique and which can be summarized as follows:
We first define a congruence $\equiv$ on the $\lambda_{\sigma w}$ term. We then define a reduction relation, denoted $\rightarrow_{\equiv}$, on the quotient of the set of terms. We then prove that the relation $\rightarrow_{\equiv}$ is strong normalizing as follows:

1. We define reducible terms and reducible substitutions for a given environment.
2. We prove that if a term or a substitution is reducible in an environment then it is $\rightarrow_{\equiv}$-strong normalizing.
3. We prove that every term or substitution which is typable in an environment is also reducible.

Finally, we show by a technical Lemma that strong normalization of $\rightarrow_{\equiv}$ implies strong normalization of $\lambda_{\sigma w}$.
We extend the proof of strong normalization for $\lambda_{\sigma w}$ to λ_P using the same ideas. We redefine the previous notions in the new context of both pattern and multiple sorts of variables (usual and choice variables). This proof is much more technical that the $\lambda_{\sigma w}$ but follows the same scheme than the one for $\lambda_{\sigma w}$.

References

[1] The Objective Caml language. `http://caml.inria.fr/`.

[2] Val Breazu-Tannen, Delia Kesner, and Laurence Puel. A typed pattern calculus. In *Proceedings of the Eigth Annual IEEE Symposium on Logic in Computer Science*, pages 262–274. IEEE Comp. Soc. Press, 1993.

[3] Serenella Cerrito and Delia Kesner. Pattern matching as cut elimination. In Giuseppe Longo, editor, *Fourteenth Annual IEEE Symposium on Logic in Computer Science*, Trento, Italy, July 1999. IEEE Comp. Soc. Press.

[4] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, March 1996.

[5] Therese Hardin, Luc Maranget, and Bruno Pagano. Functional back-ends within the lambda-sigma calculus. Technical Report RR-3034, Inria, Institut National de Recherche en Informatique et en Automatique.

[6] P. Hudak, S. Peyton-Jones, and P. Wadler (editors). Report on the programming language haskell, a non-strict, purely functional language (version 1.2). Sigplan Notices, 1992.

[7] Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[8] E. Ritter. Normalisation for typed lambda calculi with explicit substitution. *Lecture Notes in Computer Science*, 832:295–??, 1994.

Decidability of Termination of Certain One-Rule String Rewriting Systems Extended Abstract

Alfons GESER*

ICASE, NASA Langley Research Center, Hampton, VA.

February 20, 2001

Despite its simple appearance, the termination problem of single string rewriting rules (given a string rewriting rule $\ell \to r$, do all $\ell \to r$ derivations terminate?) is neither known solvable nor known unsolvable. This is so even in the case of rules $\ell \to r$ where ℓ has no self-overlap. The first promising approach for decidability has been introduced by McNaughton.

First one determines the sets $A = \mathrm{OVL}(\ell, r)$ and $B = \mathrm{OVL}(r, \ell)$ of left and right overlaps, respectively, between ℓ and r. For any $\alpha \in A$ let ℓ_α and r_α be defined by $\alpha\ell_\alpha = \ell$ and $r = r_\alpha\alpha$. Symmetrically one defines strings ℓ_β and r_β for every $\beta \in B$. Now the absence of certain rewrite steps can be expressed by the absence of certain patterns of decompositions of ℓ and r. For instance $r = \beta\ell_{\alpha_1} \ldots \ell_{\alpha_n} w \ell_{\beta_1} \ldots \ell_{\beta_m}\alpha$ is a decomposition which indicates that r may be consumed, up to a trunk w, by $n+1$ rewriting steps at the left and $m+1$ rewriting steps to the right.

McNaughton and later Kobayashi et al. distinguish ascending levels of complication that may be encountered during rewriting:

$$\text{well-behaved} \subsetneq \text{gentle} \subsetneq \text{tame} \subsetneq \text{arbitrary one-rule SRS}$$

*This work has been carried out at the Arbeitsbereich Symbolisches Rechnen, University of Tübingen. Partially supported by grant Ku 966/3-1 of the Deutsche Forschungsgemeinschaft (DFG) within the Schwerpunkt Deduktion at the University of Tübingen. Current address: ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23681. E-mail: geser@icase.edu Phone: +1 757 864-8003.

In a well-behaved rewriting derivation, any occurrence of the right hand side r of the rule cannot be totally consumed by rewriting steps. In other words, the trunk w cannot occur in a rewrite redex. Consequently the domains of rewriting within a string are separated by "dead" strings. Using this property McNaughton solved the termination problem in the class of well-behaved rules (even for the overlapping case).

For the next level in the hierarchy, gentle rules, there is as yet no decision procedure. In a gentle derivation, r may indeed by consumed, but in a strongly restricted way. First a left portion and a right portion of r is consumed, which leaves a trunk w. Then a rewrite step applies to the occurrence $\ell = \alpha' w \beta'$ of ℓ where $\alpha' \in A$ and $\beta' \in B$. This slight change accounts for a substantial complication.

Kobayashi et al. introduce a translation from non-overlapping gentle rules into different string rewriting systems with the aim that (non-)termination of the latter is easier to prove.

We present a decidability result for non-overlapping, gentle rules with one left overlap and only regular right overlaps. Regular right overlaps means that the set of right overlaps is of the form $B = \{\beta(v\beta)^{j-1} \mid 1 \leq j \leq n\}$ for some nonempty string β, string v, and $n \geq 1$.

We prove that every such rule is either left or right barren, or is well-behaved (in which cases termination is known to be decidable), or it has one of five forms that can effectively be checked. For each of the five forms we apply Kobayashi et al.'s transformation. Each of the five resulting string rewriting systems is either proven terminating by a recursive path order or proven non-terminating by a loop argument.

Reducibility method for termination properties of typed lambda terms

Silvia Ghilezan
Faculty of Engineering, University of Novi Sad, Novi Sad, Yugoslavia
Computing Science Department, Catholic University, Nijmegen, The Netherlands
e-mail: silviagh@cs.kun.nl

Viktor Kunčak
Laboratory of Computer Science, MIT, Cambridge, USA
e-mail: vkuncak@mit.edu

Silvia Likavec
Faculty of Engineering, University of Novi Sad, Novi Sad, Yugoslavia
e-mail: likavec@uns.ns.ac.yu

The reducibility method is a generally accepted way for proving the strong normalization property of various type systems. The substantial idea of the reducibility method is to interpret types by suitable sets of lambda terms which satisfy certain realizability properties and then to develop semantics in order to obtain the soundness of the type assignment. A consequence of soundness, the fact that every term typeable by a type in the type system belongs to the interpretations of that type, leads to the fact that terms typeable in the type system satisfy the required property, since the type interpretations are built up in that way.

This method was introduced by Tait for proving the strong normalization property for the simply typed lambda calculus and further developed by Girard and Tait for proving the strong normalization property for polymorphic lambda calculus.

This method is also refered to as the logical relations and it is discussed by Mitchell that apart from the strong normalization this method can be used for the proof of the confluence of (Church-Rosser property) and other basic results of the simply typed lambda calculus. The original proof of the Church-Rosser property of the simply typed lambda calculus using logical relations and the reducibility method is due to Statman and Koletsos.

The reducibility method is applied by Krivine and later by Ghilezan in order to characterize all and only the strongly normalizing lambda terms in lambda calculus with intersection types. The reducibility method is also used by Gallier for characterizing some special classes of lambda terms such as strongly normalizing terms, normalizing terms, head normalizing terms, and weak head normalizing terms by their typeability in the intersection type systems. Dezani et al. applied this method for characterizing both the mentioned terms and their persistent versions.

This work presents the reducibility method as a general framework for proving reduction properties of simply typed lambda terms and lambda terms typeable by intersection types. We distinguish two different kinds of type interpretation with respect to a given set $\mathcal{P} \subseteq \Lambda$. and two different types of conditions which the given set $\mathcal{P} \subseteq \Lambda$ has to satisfy. By combining different type interpretations with different conditions on $\mathcal{P} \subseteq \Lambda$ we build up semantics and prove soundness in both cases. The method with weaker conditions on $\mathcal{P}$ and the corresponding stronger type interpretation leads to uniform proofs of the confluence of β-reduction on typed lambda terms, the standardization property, and the unique (β-)normal form property of typed lambda terms. The method with stronger conditions on $\mathcal{P}$ and the corresponding type interpretation leads to uniform proofs of the concluence of $\beta\eta$-reduction on typed lambda terms, the unique normal $\beta\eta$-form property, the ter-

mination of the leftmost reduction, and some other properties of typed lambda terms. The strong normalization of typed lambda terms can be proved by both combinations.

Lambda term, types and type systems are defined in the following way.

Definition 1

- *The set* Λ *of lambda terms is defined by the following abstract syntax.*

$$\begin{array}{lcl} \Lambda & = & \mathtt{var} \mid \Lambda\Lambda \mid \lambda \mathtt{var}.\Lambda \\ \mathtt{var} & = & x \mid \mathtt{var}' \end{array}$$

- *The set* type *of types is defined as follows.*

$$\begin{array}{lcl} \mathtt{type} & = & \mathtt{atom} \mid \mathtt{type} \to \mathtt{type} \mid \mathtt{type} \cap \mathtt{type} \\ \mathtt{atom} & = & \alpha \mid \mathtt{atom}' \end{array}$$

- $\Gamma \vdash P : \varphi$ *is derivable in* $\lambda\cap$, *if* $\Gamma \vdash P : \varphi$ *can be generated by the following axiom-scheme and rules.*

$$\Gamma, x : \sigma \vdash x : \sigma \ \ (ax)$$

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

$$\frac{\Gamma \vdash M : \sigma \cap \tau}{\Gamma \vdash M : \sigma \quad \Gamma \vdash M : \tau}(\cap E) \qquad \frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash M : \tau}{\Gamma \vdash M : \sigma \cap \tau}(\cap I)$$

The simply typed lambda caluclus $\lambda \to$ is generated by (ax), $(\to E)$, and $(\to I)$.

In order to develop the reducibility method we consider Λ as the *applicative structure* whose domain are the lambda terms and where the application is just the application of terms. Let us define the *interpretation of types* with respect to a fixed subset $\mathcal{P} \subseteq \Lambda$ in the following way.

Definition 2 *Let* $\mathcal{P} \subseteq \Lambda$. *The map* $[\![-]\!]_{\mathcal{P}} : \mathtt{type} \to 2^{\Lambda}$ *is defined by:*

(I1) $[\![\alpha]\!]_{\mathcal{P}} = \mathcal{P}$, α *is an atom;*

(I2) $[\![\tau \cap \sigma]\!]_{\mathcal{P}} = [\![\tau]\!]_{\mathcal{P}} \cap [\![\sigma]\!]_{\mathcal{P}}$;

(I3) $[\![\tau \to \sigma]\!]_{\mathcal{P}} = [\![\tau]\!]_{\mathcal{P}} \to [\![\sigma]\!]_{\mathcal{P}} = \{M \mid \forall N \in [\![\tau]\!]_{\mathcal{P}} \quad MN \in [\![\sigma]\!]_{\mathcal{P}}\}$;

(I3$^+$) $[\![\tau \to \sigma]\!]_{\mathcal{P}} = ([\![\tau]\!]_{\mathcal{P}} \to [\![\sigma]\!]_{\mathcal{P}}) \cap \mathcal{P}$.

The *type interpretation* defined by $(I1)$, $(I2)$ and $(I3)$ will be denoted by $[\![-]\!]$, whereas the type interpretation defined by $(I1)$, $(I2)$ and $(I3^+)$ will be denoted by $[\![-]\!]_{\mathcal{P}}$ and called the *strong interpretation.*

Let us further define the *valuation of terms* $[\![-]\!]_{\rho} : \Lambda \to \Lambda$ and the *semantic satisfiability relations* $\models$ and $\models_{\mathcal{P}}$ which connects the type interpretations and the term valuations as follows.

Definition 3 *Let* $[\![-]\!]_{(\mathcal{P})} : \mathtt{type} \to 2^{\Lambda}$ *be the (strong) type interpretation for a given* $\mathcal{P} \subseteq \Lambda$ *and let* $\rho : \mathtt{var} \to \Lambda$ *be a valuation of term variables in* Λ. *Then*

1. $[\![-]\!]_{\rho} : \Lambda \to \Lambda$ *is defined by*
   $[\![M]\!]_{\rho} = M[x_1 := \rho(x_1), \ldots, x_n := \rho(x_n)]$, *where* $FV(M) = \{x_1, \ldots, x_n\}$;
2. $\rho \models_{(\mathcal{P})} M : \varphi$ *iff* $[\![M]\!]_{\rho} \in [\![\varphi]\!]_{(\mathcal{P})}$;
3. $\rho \models_{(\mathcal{P})} \Gamma$ *iff* $(\forall (x : \varphi) \in \Gamma) \quad \rho \models_{(\mathcal{P})} x : \varphi$;
4. $\Gamma \models_{(\mathcal{P})} M : \sigma$ *iff* $(\forall \rho \models_{(\mathcal{P})} \Gamma) \quad \rho \models_{(\mathcal{P})} M : \sigma$.

Let us consider the following conditions on $\mathcal{P} \subseteq \Lambda$.

Definition 4 *Let $\mathcal{P} \subseteq \Lambda$ be given. Then we define:*

($\mathcal{P}1$) $(\forall \varphi \in \mathbf{type})\ \mathbf{var} \subseteq [\![\varphi]\!]$;

($\mathcal{P}2$) $(\forall \varphi \in \mathbf{type})\ (\forall N \in \mathcal{P})\ M[x := N] \in [\![\varphi]\!] \Rightarrow (\lambda x.M)N \in [\![\varphi]\!]$;

($\mathcal{P}3$) $M \in \mathcal{P} \Rightarrow \lambda x.M \in \mathcal{P}$;

($\mathcal{P}3^+$) $Mx \in \mathcal{P} \Rightarrow M \in \mathcal{P}$.

Now we can prove the following *realizability property*, which is refered to as the soundness or the adequacy.

Proposition 5 *(Soundness)*

(i) If $\mathcal{P} \subseteq \Lambda$ satisfies ($\mathcal{P}1$), ($\mathcal{P}2$), and ($\mathcal{P}3^+$), then $\Gamma \vdash Q : \varphi \Rightarrow \Gamma \models Q : \varphi$.

(ii) If $\mathcal{P} \subseteq \Lambda$ satisfies ($\mathcal{P}1$), ($\mathcal{P}2$), and ($\mathcal{P}3$), then $\Gamma \vdash Q : \varphi \Rightarrow \Gamma \models_{\mathcal{P}} Q : \varphi$.

An immediate consequence of soundness is the following statement.

Proposition 6 *(i) Let $\mathcal{P}$ satisfy ($\mathcal{P}1$), ($\mathcal{P}2$) and ($\mathcal{P}3^+$). Then $\Gamma \vdash M : \varphi \Rightarrow M \in \mathcal{P}$.*

(ii) Let $\mathcal{P}$ satisfy ($\mathcal{P}1$), ($\mathcal{P}2$) and ($\mathcal{P}3$). Then $\Gamma \vdash M : \varphi \Rightarrow M \in \mathcal{P}$.

In order to prove that for a given $\mathcal{P} \subseteq \Lambda$ the properties ($\mathcal{P}1$) and ($\mathcal{P}2$) hold, we proceed by induction on the construction of a type, but then we need stronger induction hypotheses which are easier to prove. These stronger conditions actually unify the conditions for saturated and $\mathcal{P}$-saturated sets which are considered in reducibility methods by Krivine, Barendregt, Gallier, and Koletsos and Stavrinos.

Definition 7 *Let $\mathcal{P}$, $X \subseteq \Lambda$ be given. Then*

(i) $\mathcal{P}VAR(X)$ means $(\forall x \in \mathbf{var})\ (\forall n \geq 0)\ (\forall M_1, \ldots, M_n \in \mathcal{P})\ xM_1 \ldots M_n \in X$.

(ii) $\mathcal{P}SAT(X)$ means $(\forall M, N \in \mathcal{P})\,(\forall n \geq 0)\,(\forall M_1, \ldots, M_n \in \mathcal{P})\ M[x := N]M_1 \ldots M_n \in X \Rightarrow (\lambda x.M)NM_1 \ldots M_n \in X$.

We prove that $\mathcal{P}VAR(\mathcal{P}) \Rightarrow (\mathcal{P}1)$ and $\mathcal{P}SAT(\mathcal{P}) \Rightarrow (\mathcal{P}2)$.

The following statement presents the general reducibility method which will be applied in order to prove various reduction properties of the lambda terms typeable in $\lambda\cap$ and $\lambda \rightarrow$.

Proposition 8 *(i) Let $\mathcal{P} \subseteq \Lambda$ be such that $\mathcal{P}VAR(\mathcal{P})$, $\mathcal{P}SAT(\mathcal{P})$ and ($\mathcal{P}3^+$) hold. Then $\Gamma \vdash M : \varphi \Rightarrow M \in \mathcal{P}$.*

(ii) Let $\mathcal{P} \subseteq \Lambda$ be such that $\mathcal{P}VAR(\mathcal{P})$, $\mathcal{P}SAT(\mathcal{P})$ and ($\mathcal{P}3$) hold. Then $\Gamma \vdash M : \varphi \Rightarrow M \in \mathcal{P}$.

Proposition 8 (i) is applicable when $\mathcal{P}$ is:

$\mathcal{P} = CE = \{M \in \Lambda \mid \beta\eta\text{-reduction is confluent on } M\}$,

$\mathcal{P} = UE = \{M \in \Lambda \mid M \text{ has a unique } \beta\eta\text{-normal form}\}$,

$\mathcal{P} = N = \{M \in \Lambda \mid M \text{ is normalizing}\}$,

$\mathcal{P} = L = \{M \in \Lambda \mid \text{the leftmost reduction of M terminates}\}$.

Proposition 8 (ii) is applicable when $\mathcal{P}$ is:

$\mathcal{P} = C = \{M \in \Lambda \mid \beta\text{-reduction is confluent on } M\}$,

$\mathcal{P} = U = \{M \in \Lambda \mid M \text{ has a unique } (\beta\text{-})\text{normal form}\}$,

$\mathcal{P} = ST = \{M \mid \text{every reduction of } M \text{ can be done in a standard way}\}$.

Both Proposition 8 (i) and (ii) can be applied in case that $\mathcal{P} = SN = \{M \in \Lambda \mid \text{M is strongly normalizing}\}$.

Comparing Techniques for Automated Termination Proofs

Jürgen Giesl[1] and Aart Middeldorp[2]

[1] LuFG Informatik II
RWTH Aachen, Ahornstr. 55, 52074 Aachen, Germany
giesl@informatik.rwth-aachen.de

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

We evaluate and compare three recent transformational techniques for automated termination proofs of term rewrite systems, viz. the *dependency pair* technique of Arts and Giesl [1–3], the *dummy elimination* method of Ferreira and Zantema [5], and the *argument filtering transformation* of Kusakari, Nakamura, and Toyama [11].

Traditional methods to prove termination of term rewrite systems are based on simplification orders [4, 13]. However, the restriction to simplification orders represents a significant limitation on the class of rewrite systems that can be proved terminating. Indeed, there are numerous important and interesting rewrite systems which are not *simply terminating*, i.e., their termination cannot be proved by simplification orders. Transformation methods aim to prove termination by transforming a given term rewrite system into a term rewrite system whose termination is easier to prove. The success of such methods has been measured by how well they transform non-simply terminating rewrite systems into simply terminating rewrite systems, since simply terminating systems were the only ones where termination could be established automatically.

In recent years, the dependency pair technique of Arts and Giesl [1–3] emerged as the most powerful automatic method for proving termination of rewrite systems. For any given rewrite system, this technique generates a set of constraints which may then be solved by standard simplification orders. In this way, the power of traditional termination proving methods has been increased significantly, i.e., the class of systems where termination is provable mechanically by the dependency pair technique is much larger than the class of simply terminating systems. In light of this development, it is no longer sufficient to base the claim that a particular transformation method is successful on the fact that it may transform non-simply terminating rewrite systems into simply terminating ones. We compare two transformation methods, dummy elimination [5] and the argument filtering transformation [11], with the dependency pair technique. With respect to dummy elimination we obtain the following results:

(1) If dummy elimination transforms a given rewrite system $\mathcal{R}$ into a simply terminating rewrite system $\mathcal{R}'$, then the termination of $\mathcal{R}$ can also be proved by the most basic version of the dependency pair technique.
(2) If dummy elimination transforms a given rewrite system $\mathcal{R}$ into a *DP simply terminating* rewrite system $\mathcal{R}'$, i.e., the termination of $\mathcal{R}'$ can be proved by a simplification order in combination with the dependency pair technique, then $\mathcal{R}$ is also DP simply terminating.

These results are constructive in the sense that the constructions in the proofs are solely based on the termination proof of $\mathcal{R}'$. This shows that proving termination

of $\mathcal{R}$ directly by dependency pairs is never more difficult than proving termination of $\mathcal{R}$ via dummy elimination. The second result states that dummy elimination is useless as a preprocessing step to the dependency pair technique. Not surprisingly, the reverse statements do not hold. In other words, as far as automatic termination proofs are concerned, dummy elimination is no longer needed. (One should however remark that this observation only holds for ordinary term rewriting. There are also interesting extensions of both dummy elimination and dependency pairs to rewriting modulo equations (see [6, 7] and [9, 10, 12], respectively) and it remains to be seen how these extensions are related.)

The recent argument filtering transformation of Kusakari, Nakamura, and Toyama [11] can be viewed as an improvement of dummy elimination by incorporating ideas of the dependency pair technique. We show that the first result (1) above also holds for the argument filtering transformation. The second result (2) does not extend in its full generality, but we show that under a suitable restriction on the argument filtering applied in the transformation of $\mathcal{R}$ to $\mathcal{R}'$, DP simple termination of $\mathcal{R}'$ also implies DP simple termination of $\mathcal{R}$. For further details the reader is referred to [8].

Acknowledgements. Jürgen Giesl was partially supported by the DFG under grant GI 274/4-1 and by the JSPS under grant S-00236. Aart Middeldorp is partially supported by the Grant-in-Aid for Scientific Research C(2) 11680338 of the Ministry of Education, Science, Sports and Culture of Japan.

References

1. T. Arts and J. Giesl, *Automatically Proving Termination where Simplification Orderings Fail*, Proc. 7th TAPSOFT, Lille, France, LNCS 1214, pp. 261–273, 1997.
2. T. Arts and J. Giesl, *Modularity of Termination Using Dependency Pairs*, Proc. 9th RTA, Tsukuba, Japan, LNCS 1379, pp. 226–240, 1998.
3. T. Arts and J. Giesl, *Termination of Term Rewriting Using Dependency Pairs*, Theoretical Computer Science 236, pp. 133–178, 2000.
4. N. Dershowitz, *Termination of Rewriting*, Journal of Symbolic Computation 3, pp. 69–116, 1987.
5. M. C. F. Ferreira and H. Zantema, *Dummy Elimination: Making Termination Easier*, Proc. 10th FCT, Dresden, Germany, LNCS 965, pp. 243–252, 1995.
6. M. C. F. Ferreira, *Dummy Elimination in Equational Rewriting*, Proc. 7th RTA, New Brunswick, NJ, USA, LNCS 1103, pp. 78–92, 1996.
7. M. C. F. Ferreira, D. Kesner, and L. Puel, *Reducing AC-Termination to Termination*, Proc. 23rd MFCS, Brno, Czech Republic, LNCS 1450, pp. 239–247, 1998.
8. J. Giesl and A. Middeldorp, *Eliminating Dummy Elimination*, Proc. 17th CADE, Pittsburgh, PA, USA, LNAI 1831, pp. 309–323, 2000.
9. J. Giesl and D. Kapur, *Dependency Pairs for Equational Rewriting*, Proc. 12th RTA, Utrecht, The Netherlands, LNCS, 2001. To appear.
10. K. Kusakari and Y. Toyama, *On Proving AC-Termination by AC-Dependency Pairs*, Research Report IS-RR-98-0026F, School of Information Science, JAIST, Japan, 1998. Revised version in K. Kusakari, *Termination, AC-Termination and Dependency Pairs of Term Rewriting Systems*, PhD Thesis, JAIST, Japan, 2000.
11. K. Kusakari, M. Nakamura, and Y. Toyama, *Argument Filtering Transformation*, Proc. 1st PPDP, Paris, France, LNCS 1702, pp. 48–62, 1999.
12. C. Marché and X. Urbain, *Termination of Associative-Commutative Rewriting by Dependency Pairs*, Proc. 9th RTA, Tsukuba, Japan, LNCS 1379, pp. 241–255, 1998.
13. J. Steinbach, *Simplification Orderings: History of Results*, Fundamenta Informaticae 24, pp. 47–87, 1995.

Knowledge Based Simplification of Termination Proofs

Bernhard Gramlich*
Institut für Computersprachen, TU Wien
Favoritenstr. 9, E185-2, A-1040 Wien, Austria

February 2, 2001

Extended Abstract

There are many known approaches, methods and techniques to tackle termination proofs of rewrite systems. One of the less frequently used (abstract) approaches is to systematically investigate which knowledge can be used *a priori* to simplify the given termination proof task (before actually trying to prove termination) without sacrificing, of course, the essential logical and operational properties of the rewrite system under consideration. To this end, there exist at least two different approaches:[1]

(1) Using known general / abstract / structural results that reduce termination of a given system to a simpler termination property of (subsystems of) the same system.

(2) Transformational techniques that reduce termination of a given system to termination of a transformed (usually simpler) one.

These two abstract approaches as well as combinations thereof are particularly useful in the sense that one may use them as optional preprocessing steps that simplify the given termination proof tasks, and potentially render certain problems tractable that would have been intractable with the used underlying termination proof approach (e.g., using some class of precedence-based reduction orderings like *rpos*, recursive path ordering with status).

In previous works we had already obtained various results of type (1), including in particular several criteria which guarantee the equivalence of (general) termination and weakened termination properties like innermost or weak termination, cf. e.g. [5, 7, 6]. The underlying abstract and structural results[2] of [5, 7]

*email: gramlich@logic.at www: http://www.logic.at/staff/gramlich/

[1] Of course, the whole field of (non-)modularity analysis of termination properties of rewrite systems could explicitly be mentioned here, too. In fact, it is subsumed by (1).

[2] It seems that the first significant results of type (1) date back to **A. Church** [3] and **M.H.A. Newman** [12].

have turned out to be very powerful and useful, also in other contexts (cf. e.g. [10], [4], [1, 2], [11]).

Partially based on these results, more recent results along the line of (2) are presented in [9, 8]. These latter results provide the theoretical basis for sound (and automatic) preprocessing steps when proving termination (semi-completeness) of (orthogonal) non-overlapping rewrite systems and equational programs defined by such systems. Here, soundness means that the relevant logical and operational properties are preserved.

In this paper we investigate to what extent and how (some of) the above mentioned results of [5, 6, 7, 9] can be extended and generalized. In particular we focus on the following extensions / generalizations:

- Regarding the criteria for equivalence of termination and innermost (weak) termination in [5, 7, 6], we study the conditional case, i.e., consider 2-CTRSs and also the more general – and practically very important – case of 3-CTRSs (cf. e.g. [13],[14]) in which extra variables are allowed not only in conditions, but also in right-hand sides.

- Concerning the transformational approach of [9], where certain simplifications of right-hand sides and conditions were allowed, we show how to relax the conditions for simplifying right-hand sides, and moreover, instead of only 2-CTRSs we also consider 3-CTRSs.

In the talk we shall concentrate on the most significant progress made in the directions sketched above, as well as on substantial difficulties encountered.

References

[1] T. Arts and J. Giesl. Proving innermost normalisation automatically. In H. Comon, editor, *Proc. 8th Int. Conf. on Rewriting Techniques and Applications (RTA '97), LNCS 1232*, pages 157–171, Sitges, Spain, June 1997. Springer-Verlag.

[2] T. Arts and J. Giesl. Applying rewriting techniques to the verification of Erlang processes. In *Proc. Computer Science Logic (CSL'99), 8th Annual Conference of the EACSL, LNCS 1683*, pages 96–110, Madrid, Spain, Sept. 1999. Springer-Verlag.

[3] A. Church. The calculi of lambda conversion. In *Annals of Mathematical Studies*, volume 6. Princeton University Press, 1941.

[4] N. Dershowitz. Hierarchical termination. In N. Dershowitz and N. Lindenstrauss, editors, *Proc. 4th Int. Workshop on Conditional and Typed Rewriting Systems (CTRS'94), Jerusalem, Israel (1994), LNCS 968*, pages 89–105. Springer-Verlag, 1995.

[5] B. Gramlich. Abstract relations between restricted termination and confluence properties of rewrite systems. *Fundamenta Informaticae*, 24:3–23, 1995.

[6] B. Gramlich. On proving termination by innermost termination. In H. Ganzinger, editor, *Proc. 7th Int. Conf. on Rewriting Techniques and Applications (RTA'96)*, *LNCS 1103*, pages 93–107. Springer-Verlag, July 1996.

[7] B. Gramlich. *Termination and Confluence Properties of Structured Rewrite Systems.* PhD thesis, Fachbereich Informatik, Universität Kaiserslautern, Jan. 1996.

[8] B. Gramlich. On interreduction of semi-complete term rewriting systems. *Theoretical Computer Science.* 20 pages, accepted in June 1999, to appear in 2001.

[9] B. Gramlich. Simplifying termination proofs for rewrite systems by preprocessing. In M. Gabrielli and F. Pfenning, editors, *Proc. 2nd Int. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming (PPDP 2000)*, pages 139–150, Montreal, Canada, Sept. 2000. ACM Press. September 20 - 23, 2000, Montreal Canada,.

[10] M. Krishna Rao. Semi-completeness of hierarchical and super-hierarchical combinations of term rewriting systems. In P. Mosses, M. Nielsen, and M. Schwartzbach, editors, *Proc. 6th Int. Joint Conf. on Theory and Practice of Software Development*, *LNCS 915*, pages 379–393. Springer-Verlag, Aug. 1995.

[11] T. Nagaya and Y. Toyama. Decidability for left-linear growing term rewriting systems. In M. Rusinowitch and P. Narendran, editors, *Proc. 10th Int. Conf. on Rewriting Techniques and Applications (RTA'99)*, *LNCS 1631*, pages 256–270, Springer-Verlag, Trento, Italy, July 1999.

[12] M. Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, 43(2):223–242, 1942.

[13] T. Suzuki, A. Middeldorp, and T. Ida. Level-confluence of conditional rewrite systems with extra variables in right-hand sides. In J. Hsiang, editor, *Proc. 6th Int. Conf. on Rewriting Techniques and Applications (RTA'95)*, *LNCS 914*, pages 179–193, Kaiserslautern, Germany, Apr. 1995. Springer-Verlag.

[14] T. Yamada, J. Avenhaus, C. Loría-Sáenz, and A. Middeldorp. Logicality of conditional rewrite systems. *Theoretical Computer Science*, 236(1,2):209–232, 2000.

On Termination of Multiple Premise Ground Rewrite Systems

Dieter Hofbauer

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

String rewrite rules with an arbitrary number of premises have been studied by Post [7], and he has shown that any recursively enumerable language can be generated by such a Post system. Later Büchi proved that a syntactically restricted class of one-premise Post systems that corresponds to prefix string rewrite systems generates only regular languages [2], see also [3]. This fundamental result was then extended in various directions.

That it also holds for prefix string rules with an arbitrary number of premises was conjectured by Büchi [2] and later confirmed by Büchi and Hosken [4] and independently by Kratko [6], cf [3]. The natural extension of Büchi's result from strings to trees was considered by Brainerd [1]; he proved that one-premise ground term rewrite systems can only generate regular tree languages.

In this talk, I treat multiple premise ground term rewrite systems. It is shown that also this rather general class (effectively) preserves regularity of tree languages. The proof, however, is not based on Büchi's technique but uses a generalization of Engelfriet's derivation trees [5]. This result is then used to obtain decidability of termination and weak termination for multiple premise ground term rewrite systems.

References

1. W. S. Brainerd. Tree generating regular systems. *Information and Control*, 14(2):217–231, 1969.
2. J. R. Büchi. Regular canonical systems. *Archiv Math. Logik und Grundlagenforschung*, 6:91–111, 1964.
3. J. R. Büchi. *Finite Automata, Their Algebras and Grammars – Towards a Theory of Formal Expressions.* D. Siefkes, editor. Springer-Verlag, Berlin, 1989.
4. J. R. Büchi and W. H. Hosken. Canonical systems which produce periodic sets. *Math. Systems Theory*, 4:81–90, 1970.
5. J. Engelfriet. Derivation trees of ground term rewriting systems. *Information and Computation*, 152(1):1–15, 1999.
6. M. I. Kratko. A class of Post calculi. *Soviet Mathematics Doklady*, 6(6):1544–1545, 1965.
7. E. L. Post. Formal reductions of the general combinatorial decision problem. *Am. J. Math.*, 65:197–215, 1943.

Higher-Order Recursive Path Orderings à la carte*

Jean-Pierre Jouannaud
LRI, Bat. 490
CNRS/Univ. de Paris Sud
91405 Orsay, FRANCE
Jean-Pierre.Jouannaud@lri.fr
and
LIX, École Polytechnique
91400 Palaiseau, FRANCE

Albert Rubio
Edif. C6-219,
Univ. Politècnica de Catalunya
Pau Gargallo 5
08028 Barcelona, SPAIN
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. The use of rules in logical systems is subjected to three main meta-theoretic properties : subject reduction, local confluence, and strong normalization. The first two are usually easy. The last one is difficult, requiring the use of sophisticated proof techniques based, for example, on Tait and Girard's reducibility predicate technique. Our ambition is to remedy this situation by developping for the higher-order case the kind of semi-automated termination proof techniques that are available for the first-order case, of which the most popular one is Dershowitz's recursive path ordering.

Our contribution to this program is a reduction ordering for typed higher-order terms following a typing discipline including ML-like polymorphism and sort constructors, which conservatively extends β-reductions for higher-order terms on the one hand, and on the other hand Dershowitz's recursive path ordering for first-order unisorted 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 is bigger than t_i 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.

In a preliminary version of this work presented at the Federated Logic Conference in Trento, our ordering could only compare terms of equal types (after

*This work was partly supported by the RNRT project CALIFE and the CICYT project HEMOSS ref. TIC98-0949-C02-01.

identifying sorts such as Nat or List). In the present version, our ordering is capable of ordering terms of *decreasing types*, the ordering on types being simply a slightly weakened form of Dershowitz's recursive path ordering. This yields a very elegant presentation of the whole machinery by integrating both orderings into a single one operating on terms and types as well. This presentation should in turn be considered as the key missing stone on the way towards the definition of a recursive path ordering for dependent type calculi.

Several other improvements have been made to the preliminar version, which allow to prove a great variety of practical examples. To hint at the strength of our ordering, let us mention that the polymorphic version of Gödel's recursor for the natural numbers is easily oriented. And indeed, our ordering is polymorphic in the sense that a single comparison allows to prove the termination property of all monomorphic instances of a polymorphic rewrite rule. Many non-trivial examples can be carried out which examplify the expressive power of these orderings.

In the litterature, one can find several attempts at designing methods for proving strong normalization of higher-order rewrite rules based on ordering comparisons. These orderings are either quite weak, or need an important user interaction. Besides, they operate on terms in η-long β-normal form, hence apply only to the higher-order rewriting "à la Nipkow", based on higher-order pattern matching modulo $\beta\eta$. To our knowledge, our ordering is the first to operate on arbitrary higher-order terms, therefore applying to the other kind of rewriting, based on plain pattern matching and having β-reduction as an additional rule. On the other hand, a very simple modification of the ordering can be used to prove strong normalization of higher-order rewrite rules operating on terms in η-long β-normal forms, and indeed we show that this is true for any well-founded higher-order rewrite ordering containing β-reductions.

Is the *se*-calculus strongly normalising?*

Fairouz Kamareddine[1], and Alejandro Ríos[2]

[1] Computing and Electrical Engineering, Heriot-Watt Univ., Riccarton, Edinburgh EH14 4AS, Scotland, fairouz@cee.hw.ac.uk

[2] Department of Computer Science, University of Buenos Aires, Pabellón I - Ciudad Universitaria (1428) Buenos Aires, Argentina, rios@dc.uba.ar

Abstract. The $\lambda\sigma$-calculus (cf. [1]) reflects in its choice of operators and rules the calculus of categorical combinators (cf. [3]). The main innovation of the $\lambda\sigma$-calculus is the division of terms in two sorts: sort `term` and sort `substitution`. λs_e departs from this style of explicit substitutions in two ways. First, it keeps the classical and unique sort `term` of the λ-calculus. Second, it does not use some of the categorical operators, especially those which are not present in the classical λ-calculus. The λs_e introduces two new operators which reflect the substitution and updating that are present in the meta-language of the λ-calculus, and so it can be said to be closer to the λ-calculus from an intuitive point of view, rather than a categorical one.

The λs_e-calculus, like the $\lambda\sigma$-calculus, simulates β-reduction, is confluent (on open terms[1]) [9] and does not preserve PSN [6]. However, although strong normalisation (SN) of the σ-calculus (the substitution calculus associated with the $\lambda\sigma$-calculus) has been established, it is still unkown whether strong normalisation of the s_e-calculus (the substitution calculus associated with the λs_e-calculus) holds. Only weak normalisation of the s_e-calculus is known so far. This note, is a discussion of the status of strong normalisation of the s_e-calculus. Basically we show that the set of rules s_e is the union of two disjoint sets of rules A and B which are both SN but this does not lead us anywhere as commutation does not hold and hence modularity cannot be used to obtain SN of s_e. In addition, the distribution elimination [13] and recursive path ordering methods are not applicable and we remain unsure whether s_e is actually SN or not.

Strong normalisation of subcalculi of s_e

The last 15 years have seen an explosion in explicit substitution calculi (see [10] for a survey). As far as we know, almost all of them satisfy the property that the underlying calculus of substitutions terminate. For the λs_e-calculus [9], this property remains unsolved. This paper is to pose this problem in the hope that it can generate interest as a termination problem which at least for curiosity, needs to be settled. The answer can go either way. On the one hand, although the $\lambda\sigma$-calculus does not have PSN, the σ-calculus itself is SN. On the other hand, could the loss of PSN in the λs_e-calculus be due to the non-SN of the s_e-calculus? Are there termination techniques that we still have not explored and that could help us settle this problem? We would like to find out.

Let us summarize first the main problems that we face when trying to establish SN for s_e.

Problem 1: Unable to use recursive path ordering By taking a quick look at the s_e-rules (see Definition 22), it becomes obvious that the unfriendly rules, with respect to SN, are *σ-σ-transition* and to a lesser extent *φ-σ-transition*. These rules prevent us from establishing an order on the set of operators in order to solve the normalisation problem with a recursive path ordering.

Problem 2: Unable to use Zantema's distribution elimination lemma The s_e-rules "look like" associative rules but unfortunately they are not; e.g. in *σ-σ-transition* one could think of the σ^j-operator distributing over the σ^i-operator, but it is not a "true" distribution: σ^j changes to σ^{j+1} when acting on the first term and to σ^{j-i+1} when acting on the second. This prevents us from using Zantema's distribution elimination method [13] to obtain SN.

* This work was carried out under EPSRC grants GR/K25014, GR/L15685 and GR/L36963.

[1] The λs_e-calculus is confluent on the whole set of open terms whereas $\lambda\sigma$ is confluent on the open terms without metavariables of sort `substitution` as is shown in [12].

Problem 3: Unable to use modularity Another technique to show SN is modularity, i.e. establish SN for certain subcalculi and afterwards prove that these subcalculi satisfy a commutation property to conclude SN for the whole calculus. At the end of this note we will come back to this point and show that the necessary commutation results do not hold.

Let us say here that, even if σ-σ-*transition* seems responsible for the difficulties in establishing SN, Zantema succeded in establishing that the σ-σ-*transition* scheme on its own is SN (personal communication cited in [9]). Here we shall go a step further: we shall prove that σ-σ-*tr.*+φ-σ-*tr.* is SN and also that the calculus obtained with the rest of the rules is SN as well.

In this note we shall frequently use the following nomenclature:

Definition 1 *We define the following sets of rules:*
$*\varphi = \{\sigma\text{-}\varphi\text{-}tr.1,\ \sigma\text{-}\varphi\text{-}tr.2,\ \varphi\text{-}\varphi\text{-}tr.1,\ \varphi\text{-}\varphi\text{-}tr.2\}$,
$*\sigma = \{\sigma\text{-}\sigma\text{-}tr.,\ \varphi\text{-}\sigma\text{-}tr.\}$,
$*\varphi^{-} = \{\sigma\text{-}\varphi\text{-}tr.1,\ \varphi\text{-}\varphi\text{-}tr.2\}$, $*\varphi^{--} = \{\sigma\text{-}\varphi\text{-}tr.2,\ \varphi\text{-}\varphi\text{-}tr.1\}$.

Note that $s_e = (s + *\varphi) + *\sigma$. We shall prove in this note that both calculi generated by the set of rules $s + *\varphi$ (Theorem 4) and $*\sigma$ (Theorem 11) are SN. Unfortunately, these calculi do not possess the property of commutation needed to ensure that their union s_e is SN (see Example 14).

It is not difficult to prove that $s + *\varphi$ is SN by giving a weight that decreases through reduction. We begin by defining two weight functions we will need for the final weight:

Definition 2 *Let* $P : \Lambda s_{op} \to I\!N$ *and* $W : \Lambda s_{op} \to I\!N$ *be defined inductively by:*

$P(X) = P(\mathtt{n}) = 2$	$W(X) = W(\mathtt{n}) = 1$
$P(a\,b) = P(a) + P(b)$	$W(a\,b) = W(a) + W(b) + 1$
$P(\lambda a) = P(a)$	$W(\lambda a) = W(a) + 1$
$P(a\,\sigma^j b) = j * P(a) * P(b)$	$W(a\,\sigma^j b) = 2 * W(a) * (W(b) + 1)$
$P(\varphi^i_k a) = (k+1) * (P(a) + 1)$	$W(\varphi^i_k a) = 2 * W(a)$

Lemma 3 *For* $a, b \in \Lambda s_{op}$ *the following hold:*
1. *If* $a \to_{s+*\varphi} b$ *then* $W(a) \geq W(b)$.
2. *If* $a \to_{s+*\varphi^-} b$ *then* $W(a) > W(b)$.
3. *If* $a \to_{*\varphi^{--}} b$ *then* $P(a) > P(b)$.

PROOF: By induction on a: if the reduction is internal, the IH applies; otherwise, the theorem must be checked for each rule. ⊠

An immediate consequence of the previous lemma is:

Theorem 4 *The* $s + *\varphi$*-calculus is SN.*

PROOF: The previous lemma ensures that the ordinal $(W(a), P(a))$ decreases with the lexicographical order for each $s + *\varphi$-reduction. ⊠

Now, to prove SN for $*\sigma$ we are going to use the isomorphism presented in the appendix and the technique that Zantema used to prove SN for the calculus whose only rule is σ-σ-*transition* (cf. [9]). Following this isomorphism, the schemes σ-σ-*tr.* and φ-σ-*tr.* of λs_e both translate into the same scheme of $\lambda\omega_e$, namely σ-/-*transition* of Definition 28.

Zantema uses the following lemma (cf. [11]):

Lemma 5 *Any reduction relation* $\to$ *on a set* T *satisfying 1,2, and 3 is strongly normalising:*
1. $\to$ *is weakly normalising.*
2. $\to$ *is locally confluent.*
3. $\to$ *is increasing, i.e.,* $\exists$ *a function* $f : T \longrightarrow I\!N$ *where* $a \to b \Rightarrow f(a) < f(b)$.

We use the previous lemma to prove that the calculus whose only rule is σ-/-*transition*, let us call it σ-/-calculus, is strongly normalising. For the σ-/ $-$ *calculus*, 2 follows from a simple critical pair analysis and 3 can be easily established by choosing $f(a)$ to be the size of a. To show weak normalisation of the σ-/ $-$ *calculus* the technique used by Zantema (cf. [9]) can be adapted here:

Definition 6 *We say that* $c \in \Lambda\omega^t$ *is an* external normal form *if* $c = a[s_1]_{i_1} \cdots [s_n]_{i_n}$ *where* $a \neq c[d/]_k$ *and if* $s_k = b_k/$ *then* $i_k > i_{k+1}$. *We denote the set of external normal forms* ENF.

Lemma 7 *Let* $c = a[s_1]_{i_1} \cdots [s_n]_{i_n} \in ENF$ *and let* $i_n \leq i_{n+1}$ *and* $s_n = b_n/$ *then there exists a* σ-/*-derivation* $c \to^+ a[t_1]_{j_1} \cdots [t_{n+1}]_{j_{n+1}} \in ENF$ *such that* $j_{n+1} = i_n$ *and for every* r *with* $1 \leq r \leq n+1$ *we have either* $t_r = s_k$ *for some* $k \leq n+1$ *or* $t_r = (a_p[s_{n+1}])/$ *for some* $s_p = a_p/$ *with* $1 \leq p \leq n$.

PROOF: By induction on n. ⊠

Lemma 8 *Let* $c = a[s_1]_{i_1} \cdots [s_n]_{i_n}$ *such that* $a \neq c[d/]_k$. *There exists a* σ-/*-derivation* $c \twoheadrightarrow a[t_1]_{j_1} \cdots [t_n]_{j_n} \in ENF$ *such that for every* r *with* $1 \leq r \leq n+1$ *we have either* $t_r = s_k$ *for some* $k \leq n$ *or* $t_r = (a_{p_r 1}[s_{p_r 2}]_{k_2} \cdots [s_{p_r n}]_{k_n})/$ *with* $1 \leq p_{r1} \leq \cdots \leq p_{rn} \leq n$ *and with some* $s_p = a_{p_r 1}/$ $(1 \leq p \leq n)$.

PROOF: By induction on n, using the previous lemma. ⊠

Lemma 9 *The* σ-/*-calculus is weakly normalising.*

PROOF: Suppose there is a term c not having a normal form for which every term smaller (in size) than c admits a normal form. Let $c = a[s_1]_{i_1} \cdots [s_n]_{i_n}$ such that $a \neq c[d/]_k$. Applying Lemma 8, we get $c \to \to a[t_1]_{j_1} \cdots [t_n]_{j_n} \in ENF$. Note that $a, t_1, \cdots t_n$ are all smaller than c and hence admit a normal form. Now replacing each of them by its normal form in $a[t_1]_{j_1} \cdots [t_n]_{j_n}$ we have a normal form for c which is a contradiction. ⊠

Therefore we can finally apply Lemma 5 to conclude:

Theorem 10 *The* σ-/*-calculus is strongly normalising on* $\Lambda\omega^t$.

Now, using the isomorphism, since, as we mentioned before, both rule schemes in $*\sigma$ translate into the single σ-/ rule scheme, we have:

Theorem 11 *The* $*\sigma$*-calculus is strongly normalising.*

Now that $s + *\varphi$ and $*\sigma$ have been proved SN the question arises whether the whole system can be proved SN using a modularity result. The answer is negative for the classical modularity theorem of Bachmair-Dershowitz, which we recall here:

Definition 12 *A rewrite relation* R *commutes over* S *if whenever* $a \to_S b \to_R c$, *there is an alternative derivation* $a \to_R d \to_{R \cup S} c$.

Theorem 13 (Bachmair-Dershowitz-85) *Let* R *commute over* S. *The combined system* $R \cup S$ *is SN iff* R *and* S *both are SN.*

The following example shows that no commutation is possible between $s + *\varphi$ and $*\sigma$ and therefore the Bachmair-Dershowitz's Theorem cannot be applied to get SN for s_e.

Example 14 *Now, here is an example which shows that* $*\sigma$ *does not commute over* $s + *\varphi$: *Let* $k + i \leq j$, $h \leq j - i + 1$ *and* $h > k + 1$. *Let us consider the following derivation:*

$$(\varphi_k^i(a\,\sigma^h b))\,\sigma^j c \to_{*\varphi} \varphi_k^i((a\,\sigma^h b)\,\sigma^{j-i+1} c) \to_{\sigma-\sigma-tr} \varphi_k^i((a\,\sigma^{j-i+2} c)\,\sigma^h(b\,\sigma^{j-i-h+2} c))$$

But it is easy to see that $(\varphi_k^i(a\,\sigma^h b))\,\sigma^j c$ *does not contain any* $*\sigma$*-redex.*

On the other hand, $s + *\varphi$ *does not commute over* $*\sigma$ *either: Let* $i \leq j$ *and let us consider the following derivation:*

$$((\lambda a)\,\sigma^i b)\,\sigma^j c) \to_{\sigma-\sigma-tr} ((\lambda a)\,\sigma^{j+1} c)\,\sigma^i(b\,\sigma^{j-i+1} c) \to_s (\lambda(a\,\sigma^{j+2} c))\,\sigma^i(b\,\sigma^{j-i+1} c)$$

But reducing the only s*-redex in* $((\lambda a)\,\sigma^i b)\,\sigma^j c)$ *we get* $(\lambda(a\,\sigma^{i+1} b))\,\sigma^j c$ *which also has a unique* s*-redex. Reducing it we get* $\lambda((a\,\sigma^{i+1} b)\,\sigma^{j+1} c)$ *and now there is only the* σ-σ*-transition redex, whose reduction gives us* $\lambda((a\,\sigma^{j+2} c)\sigma^{i+1}(b\,\sigma^{j-i+1} c))$ *which has no further redexes. Therefore,* $(\lambda(a\,\sigma^{j+2} c))\,\sigma^i(b\,\sigma^{j-i+1} c)$ *cannot be reached from* $((\lambda a)\,\sigma^i b)\,\sigma^i c)$ *with an* s_e*-derivation beginning with an* s*-step.*

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit Substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.
2. Z. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. λv, a calculus of explicit substitutions which preserves strong normalisation. *Functional Programming*, 6(5), 1996.
3. P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Pitman, 1986. Revised edition : Birkhäuser (1993).
4. P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. Technical Report RR 1617, INRIA, Rocquencourt, 1992.
5. N. de Bruijn. Lambda-Calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indag. Mat.*, 34(5):381–392, 1972.
6. B. Guillaume. *Un calcul des substitutions avec etiquettes.* PhD thesis, Université de Savoie, Chambéry, France, 1999.
7. T. Hardin and A. Laville. Proof of Termination of the Rewriting System SUBST on CCL. *Theoretical Computer Science*, 46:305–312, 1986.
8. F. Kamareddine and A. Ríos. A λ-calculus à la de Bruijn with explicit substitutions. Proceedings of PLILP'95. *LNCS*, 982:45–62, 1995.
9. F. Kamareddine and A. Ríos. Extending a λ-calculus with explicit substitution which preserves strong normalisation into a confluent calculus on open terms. *Journal of Functional Programming*, 7(4):395–420, 1997.
10. F. Kamareddine and Alejandro Ríos. Relating the $\lambda\sigma$- and λs-styles of explicit substitutions. *Logic and Computation*, 10(3):349–380, 2000.
11. J.-W. Klop. Term rewriting systems. *Handbook of Logic in Computer Science*, II, 1992.
12. A. Ríos. *Contribution à l'étude des λ-calculs avec substitutions explicites.* PhD thesis, Université de Paris 7, 1993.
13. H. Zantema. Termination of term rewriting: interpretation and type elimination. *J. Symbolic Computation*, 17(1):23–50, 1994.

On termination of OBJ programs*

Salvador Lucas

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

Eager rewriting-based languages such as Lisp, OBJ*, CafeOBJ, ELAN, or Maude use innermost rewriting to evaluate initial expressions. A frequent problem here is nontermination. *Syntactic annotations* (i.e., associated to the arguments of symbols) have been used in OBJ2 [FGJM85], OBJ3 [GWMFJ00], CafeOBJ [FN97], or Maude [CELM96] as *replacement restrictions* to (hopefully) avoid nontermination. Within the program text, they are specified as sequences of integers in parentheses called *local strategies*. For instance, the following OBJ3 program:

```
obj EXAMPLE is
  sorts Sort .
  op 0 : -> Sort .
  op s : Nat -> Sort .
  op cons : Sort Sort -> Sort [strat: (1 0)] .
  op from : Sort -> Sort .
  op sel : Sort Sort -> Sort .
  var X Y L : Sort .
  eq sel(s(X),cons(Y,L)) = sel(X,L) .
  eq sel(0,cons(X,L)) = X .
  eq from(X) = cons(X,from(s(X)) .
endo
```

specifies a *explicit* local strategy for the list constructor `cons`. Symbols with no explicit strategy obtain a *default* strategy which is associated by the system (see [GWMFJ00]).

Local strategies serve to *completely guide* the *evaluation strategy* of OBJ programs[1]: when considering a function call $f(t_1, \ldots, t_k)$, only the arguments whose indices are present as *positive* integers in the list associated to the local strategy of f are evaluated (following the ordering which has been specified in the list). If an index 0 is found, then the reduction of the external function call is attempted. *Negative* indices indicate that the corresponding argument is to be evaluated 'on-demand', where a 'demand' is an attempt to match a pattern to the term that occurs in such an argument position [GWMFJ00,OF00].

According to this, the second argument of the list constructor `cons` will *never* be evaluated within a call to `cons`. The presence of such 'true' replacement restrictions is often invoked to justify that, even though the underlying execution mechanism is innermost rewriting, OBJ programs are able to achieve a *lazy* behavior thus avoiding nontermination [GWMFJ00]. For instance, with our OBJ3 program, the evaluation of `sel(s(s(0)),from(0))` should produce `s(s(0))`, even though the 'infinite list' `from(0)` is a part of the expression. The question is: can we (maybe automatically) *ensure* this? i.e., will the evaluation *terminate*? Unfortunately, no formal analysis about how a particular choice of replacement restrictions modify termination of OBJ (like) programs is available.

Term rewriting systems (TRSs) provide a suitable computational model for programs written in more sophisticated programming languages. Syntactic replacement restrictions can be associated to symbols f of a signature Σ by means of a *replacement map* $\mu : \Sigma \to \mathcal{P}(\mathbb{N})$ [Luc98] that discriminates the argument positions $\mu(f) \subseteq \{1, \ldots, ar(f)\}$ on which we *can* perform replacements. For instance, the previous OBJ3 program can be associated to TRS

* This work has been partially supported by CICYT TIC 98-0445-C03-01.

[1] As in [GWMFJ00], by OBJ we mean OBJ2, OBJ3, or CafeOBJ.

```
sel(s(x),y:l) → sel(x,l)
sel(0,x:l)    → x
from(x)       → x:from(s(x))
```

and replacement map μ given by $\mu(:) = \{1\}$ and $\mu(f) = \{1, \ldots, ar(f)\}$ for all other symbol f.

Negative integers in local strategies (allowed in OBJ3 or CafeOBJ programs) can be managed using an *additional* replacement map μ_D containing their absolute values. Context-sensitive rewriting (*CSR*) [Luc98] and lazy (graph) rewriting (*LR*) [FKW00,KW95] provide operational models for using replacement restrictions specified by a single replacement map μ. We define *on-demand rewriting* (using two replacement maps μ and μ_D) as a generalization of *LR* that also includes *CSR*. Termination of *CSR* has been studied in some extent in [GM99,Luc96,Zan97]. We connect termination of on-demand rewriting (*ODR*) and termination of *CSR*.

Semantics of OBJ programs is usually given as a recursive evaluation function *eval* (mapping terms into their computed values, if any) rather than specifying the concrete rewrite steps leading to computed values (see [OF00]). Expressing OBJ computations as rewritings enables the use of the theory of Term Rewriting to analyze termination. We show that *CSR* and *ODR* can be used to model computations of OBJ programs thus giving a tool for analyzing termination. For instance, by using the results of [Zan97], we can show that context-sensitive reductions with the previous TRS under the replacement map μ are terminating. Thus, our OBJ3 program is terminating.

References

[CELM96] M. Claver, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In *Proc. 1st International Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science, Elsevier Sciences, 1996.

[FGJM85] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages, POPL'85*, pages 52-66, ACM Press, 1985.

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

[FN97] K. Futatsugi and A. Nakagawa. An Overview of CAFE Specification Environment – An algebraic approach for creating, verifying, and maintaining formal specification over networks –. In *Proc. of 1st International Conference on Formal Engineering Methods*, 1997.

[GM99] J. Giesl and A. Middeldorp. Transforming Context-Sensitive Rewrite Systems. In P. Narendran and M. Rusinowitch, editors, *Proc. of 10th International Conference on Rewriting Techniques and Applications, RTA'99*, LNCS 1631:271-285, Springer-Verlag, Berlin, 1999.

[GWMFJ00] J.A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducig OBJ. In J. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: algebraic specification in action*, Kluwer, 2000.

[KW95] J.F.Th. Kamperman and H.R. Walters. Lazy Rewriting and Eager Machinery. In J. Hsiang, editor, *Proc. of the 6th International Conference on Rewriting Techniques and Applications, RTA'95*, LNCS 914:147-162, Springer-Verlag, Berlin, 1995.

[Luc96] S. Lucas. Termination of context-sensitive rewriting by rewriting. In F. Meyer auf der Heide and B. Monien, editors, *Proc. of 23rd. International Colloquium on Automata, Languages and Programming, ICALP'96*, LNCS 1099:122-133, Springer-Verlag, Berlin, 1996.

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

[OF00] K. Ogata and K. Futatsugi. Operational Semantics of Rewriting with the On-demand Evaluation Strategy. In *Proc of 2000 International Symposium on Applied Computing, SAC'00*, pages 756-763, ACM Press, 2000.

[Zan97] H. Zantema. Termination of Context-Sensitive Rewriting. In H. Comon, editor, *Proc. of 8th International Conference on Rewriting Techniques and Applications, RTA'97*, LNCS 1232:172-186, Springer-Verlag, Berlin, 1997.

Relating termination and infinitary normalization*

Salvador Lucas

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

Lazy languages admit giving *infinite values* as the meaning of expressions. Infinite values are limits of converging infinite sequences of *partially defined* values which are more and more defined. This can be formalized by using *strongly convergent infinitary rewrite sequences* [KKSV95]. These are Cauchy convergent rewrite sequences in which redexes are (ultimately) contracted deeper and deeper. We only consider sequences of length at most ω issued from finite terms. The existence of a strongly convergent rewrite sequence leading from a term to another is undecidable. However, we have the following.

Theorem 1. *Let $\mathcal{R}$ be a confluent, left-linear, right ground TRS over a finite signature, $t \in \mathcal{T}(\Sigma)$, and δ be a constructor (possibly infinite) term. It is decidable whether t reduces to δ.*

A TRS is *top-terminating* if no infinitary reduction sequence performs infinitely many rewrites at Λ [DKP91]. A TRS $\mathcal{R}$ is *strongly convergent* if all infinitary reductions in $\mathcal{R}$ are strongly convergent [GOV99]. Giesl et al. showed that *dependency pairs* [AG00] can be used for proving strong convergency of TRSs (without collapsing rules) [GOV99]. Kennaway et al. remarked that *top-termination* also ensures strong convergence of TRSs; this was implicit in the proof of Proposition 5.1 in [DKP91]. In fact, both notions coincide.

Theorem 2. *A TRS is top-terminating if and only if is strongly convergent.*

A TRS is *infinitary normalizing* if every (finite) term t admits a strongly convergent sequence starting from t and ending into a normal form (i.e., a term without redexes). Infinitary normalizing TRSs can be thought of as playing the role of normalizing TRSs in the infinitary setting, i.e., ensuring that every term has a meaning. The following fact easily follows from[1] [DKP91].

Theorem 3. *Every left-linear, top-terminating TRS is infinitary normalizing.*

Infinitary normalizing TRSs do not need to be top-terminating: Consider the TRS $\mathcal{R}$:

$$\mathtt{f(a)} \to \mathtt{f(f(a))} \qquad \mathtt{f(a)} \to \mathtt{a}$$

It is not difficult to see that $\mathcal{R}$ is infinitary normalizing; however, the infinite rewrite sequence

$$\underline{\mathtt{f(a)}} \to \mathtt{f}(\underline{\mathtt{f(a)}}) \to \underline{\mathtt{f(a)}} \to \cdots$$

performs infinitely many reductions at the top position.

We show that it is possible to prove top-termination of a left-linear TRS $\mathcal{R}$ by proving *termination* of the *canonical* context-sensitive rewrite relation associated to $\mathcal{R}$. In context-sensitive rewriting [Luc98], we (only) rewrite subterms at *replacing* arguments; a *replacement*

* This work has been partially supported by CICYT TIC 98-0445-C03-01.

[1] But note that the notion of infinitary normal form used in [DKP91] (a term t such that $t = t'$ whenever $t \to t'$) differs from the usual one.

map $\mu : \Sigma \to \mathcal{P}(\mathbb{N})$ discriminates the argument positions $\mu(f) \subseteq \{1, \ldots, ar(f)\}$, on which we *can* perform replacements. This is extended to positions of terms inductively. If the context-sensitive rewrite relation associated to a TRS $\mathcal{R}$ and a replacement map μ is terminating, we say that $\mathcal{R}$ is μ-terminating. A replacement map μ is *more restrictive* than μ', written $\mu \sqsubseteq \mu'$ if $\forall f \in \Sigma, \mu(f) \subseteq \mu'(f)$; $\mu_\top$ given by $\mu_\top(f) = \{1, \ldots, ar(f)\}$ for all $f \in \Sigma$ is the less restrictive map, imposing no restriction. Given a TRS $\mathcal{R}$, its *canonical* replacement map $\mu_{\mathcal{R}}^{can}$ is the most restrictive replacement map that permits replacements at every non-variable position of the *lhs*'s of $\mathcal{R}$. We have the following.

Theorem 4. *If a left-linear TRS $\mathcal{R}$ is $\mu_{\mathcal{R}}^{can}$-terminating, then $\mathcal{R}$ is top-terminating.*

Strongly convergent TRSs do not need to be $\mu_{\mathcal{R}}^{can}$-terminating: Consider the TRS $\mathcal{R}$:

```
f(a) → f(f(a))
```

It is not difficult to see that $\mathcal{R}$ is strongly convergent. However, it is not $\mu_{\mathcal{R}}^{can}$-terminating, since $\mu_{\mathcal{R}}^{can} = \mu_\top$ and $\mathcal{R}$ is not terminating.

Since μ'-terminating TRSs are also μ-terminating whenever $\mu \sqsubseteq \mu'$ (see [Luc96]), Theorem 4 can trivially be extended to replacement maps μ such that $\mu_{\mathcal{R}}^{can} \sqsubseteq \mu$. However, Theorem 4 does not necessarily hold if $\mu_{\mathcal{R}}^{can} \not\sqsubseteq \mu$: consider our first example; if $\mu(\mathtt{f}) = \varnothing$, then $\mathcal{R}$ is μ-terminating but it is not top-terminating. In this way, μ-termination criteria [GM99,Luc96,SX98,Zan97] can also be used for proving infinitary normalization. The advantage w.r.t. [GOV99] is that we can use available software and techniques for proving termination by just pre-processing the original TRS by using the transformations of [GM99,Luc96,SX98,Zan97].

References

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

[DKP91] N. Dershowitz, S. Kaplan, and D. Plaisted. Rewrite, rewrite, rewrite, rewrite, rewrite. *Theoretical Computer Science* 83:71-96, 1991.

[GM99] J. Giesl and A. Middeldorp. Transforming Context-Sensitive Rewrite Systems. In P. Narendran and M. Rusinowitch, editors, *Proc. of 10th International Conference on Rewriting Techniques and Applications, RTA '99*, LNCS 1631:271-285, Springer-Verlag, Berlin, 1999.

[GOV99] J. Giesl, V. van Oostrom, and F.J. de Vries. Strong convergence of term rewriting using strong dependency pairs (Extended abstract). In *Proc. of 4th International Workshop on Termination, WST'99*, pages 38-39, 1999.

[KKSV95] R. Kennaway, J.W. Klop, M.R. Sleep, and F.-J. de Vries. Transfinite reductions in Orthogonal Term Rewriting Systems. *Information and Computation* 119(1):18-38, 1995.

[Luc96] S. Lucas. Termination of context-sensitive rewriting by rewriting. In F. Meyer auf der Heide and B. Monien, editors, *Proc. of 23rd. International Colloquium on Automata, Languages and Programming, ICALP'96*, LNCS 1099:122-133, Springer-Verlag, Berlin, 1996.

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

[SX98] J. Steinbach and H. Xi. Freezing – Termination Proofs for Classical, Context-Sensitive and Innermost Rewriting. Institut für Informatik, T.U. München, January 1998.

[Zan97] H. Zantema. Termination of Context-Sensitive Rewriting. In H. Comon, editor, *Proc. of 8th International Conference on Rewriting Techniques and Applications, RTA '97*, LNCS 1232:172-186, Springer-Verlag, Berlin, 1997.

Approximating Dependency Graphs using Tree Automata Techniques — Extended Abstract —

Aart Middeldorp* and Seitaro Yuuki

1 Dependency Pairs

This paper is concerned with the dependency pair method of Arts and Giesl [1], a powerful and automatable method for proving termination of rewrite systems. In this method a rewrite system is transformed into a set of ordering constraints such that termination of the rewrite system is equivalent to the solvability of the constraints. The generated constraints are typically solved by standard techniques (polynomial interpretations, path orders), even when these techniques are not applicable to the original rewrite system.

The ordering constraints in the dependency pair method are generated by analyzing the cycles in the *dependency graph*. This graph summarizes the relationships between the dependency pairs of the rewrite system. More precisely, there is an arrow from $s \to t$ to $u \to v$ in the dependency graph if some instance of t rewrites to some instance of u. Since this is undecidable in general, the dependency graph has to be estimated by a decidable approximation. Arts and Giesl [1] proposed a simple algorithm for this purpose.

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

Example 1. For the TRS $\mathcal{R}$ consisting of the two rewrite rules

$$\begin{aligned} \mathsf{f}(\mathsf{a},\mathsf{b},x) &\to \mathsf{f}(x,x,x) \\ \mathsf{a} &\to \mathsf{c} \end{aligned}$$

there is one dependency pair $\mathsf{F}(\mathsf{a},\mathsf{b},x) \to \mathsf{F}(x,x,x)$ $(*)$. Since $\mathsf{F}(\mathsf{a},\mathsf{b},x)$ unifies with $\mathsf{REN}(\mathsf{CAP}(\mathsf{F}(x,x,x))) = \mathsf{F}(x_1,x_2,x_3)$, $\mathsf{EDG}(\mathcal{R})$ consists of the cycle $(*)$. This gives rise to the following ordering constraints:

$$\mathsf{F}(\mathsf{a},\mathsf{b},x) > \mathsf{F}(x,x,x) \qquad \mathsf{f}(\mathsf{a},\mathsf{b},x) \gtrsim \mathsf{f}(x,x,x) \qquad \mathsf{a} \gtrsim \mathsf{c}$$

with $>$ a well-founded order and $\gtrsim$ a weakly monotonic quasi-order compatible with $>$ (i.e., $\gtrsim \cdot > \subseteq >$ or $> \cdot \gtrsim \subseteq >$) such that both $>$ and $\gtrsim$ are closed under

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

substitutions. Since the first ordering constraint is not satisfied by any standard order used for proving termination, an automatic termination proof will fail. Note however that the cycle does not exist in the real dependency graph $\mathsf{DG}(\mathcal{R})$ as no instance of $\mathsf{F}(x,x,x)$ rewrites to an instance of $\mathsf{F}(\mathsf{a},\mathsf{b},x)$.

The approximation of Arts and Giesl often results in an unnecessarily large graph and hence a large number of constraints. Sometimes, as in the above example, this causes the failure of the termination proof. Our aim is to show that by using tree automata techniques we obtain a much better estimation of the dependency graph.

2 Tree Automata Techniques

Our approach is based on the following two ingredients:

1. The powerful framework of Durand and Middeldorp [2] for the study of decidable call-by-need computations in orthogonal term rewriting. This framework is parameterized by so-called approximation mappings. An approximation mapping abstracts from parts of the terms in the rewrite rules such that the set of terms that rewrite to a term in an arbitrary regular tree language is regular.
2. The folklore result that it is decidable whether the set of ground instances of an arbitrary term intersects with a regular tree language. This result is well-known for linear terms but it also holds for non-linear terms.

A set of ground terms is said to be regular if it is accepted by a (finite bottom-up) tree automaton. We write $\Sigma(t)$ for the set of ground instances of the term t. If $\mathcal{R}$ is a TRS over a signature $\mathcal{F}$ and $L \subseteq \mathcal{T}(\mathcal{F})$ then $(\to_{\mathcal{R}}^*)[L]$ denotes the set of all terms $s \in \mathcal{T}(\mathcal{F})$ such that $s \to_{\mathcal{R}}^* t$ for some term $t \in L$.

Definition 2. *An* approximation mapping *is a mapping α from TRSs to TRSs with the property that $\to_{\mathcal{R}} \subseteq \to_{\alpha(\mathcal{R})}^*$ for every TRS $\mathcal{R}$. In the following we write $\mathcal{R}_\alpha$ instead of $\alpha(\mathcal{R})$. We say that α is* regularity preserving *if $(\to_{\mathcal{R}_\alpha}^*)[L]$ is regular for all TRSs $\mathcal{R}$ and regular L.*

In this extended abstract we consider only the approximation mapping *nv* of Oyamaguchi. Let $\mathcal{R}$ be a TRS. The nv approximation $\mathcal{R}_{\mathrm{nv}}$ is obtained from $\mathcal{R}$ by replacing all occurrences of variables in the rewrite rules by fresh variables. It is well-known that nv is regularity preserving.

Definition 3. *Let $\mathcal{R}$ be a TRS and α an approximation mapping. The nodes of the* α-approximated *dependency graph $\mathsf{DG}_\alpha(\mathcal{R})$ are the dependency pairs of $\mathcal{R}$ and there is an arrow from $s \to t$ to $u \to v$ if and only if both $\Sigma(t) \cap (\to_{\mathcal{R}_\alpha}^*)[\Sigma(\mathsf{REN}(u))] \neq \varnothing$ and $\Sigma(u) \cap (\to_{(\mathcal{R}^{-1})_\alpha}^*)[\Sigma(\mathsf{REN}(t))] \neq \varnothing$.*

Lemma 1. *Let $\mathcal{R}$ be a TRS and α an approximation mapping.*

1. *If α is regularity preserving then $\mathsf{DG}_\alpha(\mathcal{R})$ is computable.*
2. $\mathsf{DG}(\mathcal{R}) \subseteq \mathsf{DG}_\alpha(\mathcal{R})$.

3 Comparison

In this section we compare our nv-approximated dependency graph with the estimated dependency graph of Arts and Giesl and the approximation of the dependency graph defined by Kusakari and Toyama [5].

Theorem 1. $\mathsf{DG}_{\mathrm{nv}}(\mathcal{R}) \subseteq \mathsf{EDG}(\mathcal{R})$ *for every TRS* $\mathcal{R}$.

The reverse does not hold. Consider the TRS $\mathcal{R}$ of Example 1. The TRS $\mathcal{R}_{\mathrm{nv}}$ consists of the two rules

$$\begin{aligned} \mathsf{f}(\mathsf{a},\mathsf{b},x) &\to \mathsf{f}(x_1,x_2,x_3) \\ \mathsf{a} &\to \mathsf{c} \end{aligned}$$

and we clearly have $\Sigma(\mathsf{REN}(\mathsf{F}(\mathsf{a},\mathsf{b},x))) = \{\mathsf{F}(\mathsf{a},\mathsf{b},t) \mid t \in \mathcal{T}(\mathcal{F})\}$. The only term that rewrites in $\mathcal{R}_{\mathrm{nv}}$ to a (b) is a (b) itself and hence $(\to_{\mathcal{R}_{\mathrm{nv}}}^*)[\Sigma(\mathsf{REN}(\mathsf{F}(\mathsf{a},\mathsf{b},x)))]$ $= \{\mathsf{F}(\mathsf{a},\mathsf{b},t) \mid t \in \mathcal{T}(\mathcal{F})\}$. Because no instance of $\mathsf{F}(x,x,x)$ belongs to this set, $\mathsf{DG}_{nv}(\mathcal{R})$ contains no arrow. Therefore $\mathcal{R}$ is trivially terminating.

The TRS $\mathcal{R}$ is interesting because it is not *DP quasi-simply terminating*. The class of DP quasi-simply terminating TRSs was introduced by Giesl and Ohlebusch [4] and supposed to "capture all TRSs where an automated termination proof using dependency pairs is potentially feasible". We note that the various refinements of the dependency pair method (narrowing, rewriting, instantiation; see Giesl and Arts [3]) are not applicable and moreover that proving innermost termination (which is easy with the standard dependency pair technique) is insufficient for termination as $\mathcal{R}$ does not belong to a known class for which termination and innermost termination coincide.

Kusakari and Toyama [5] defined a rather complicated approximation of the dependency graph based on the concepts of ω-reduction and Ω-reduction. It can be shown that the approximated dependency graph of Kusakari and Toyama and the estimated dependency graph of Arts and Giesl are incomparable in general (contradicting the remark in [5] that their algorithm is more powerful than the one of Arts and Giesl). It can also be shown that our nv-approximated dependency graph is always a subgraph and often a proper subgraph of Kusakari and Toyama's approximated dependency graph. In particular, their approximation is of no help for proving termination of the TRS of Example 1.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
2. I. Durand and A. Middeldorp. Decidable call by need computations in term rewriting. In *Proc. 14th CADE*, volume 1249 of *LNAI*, pages 4–18, 1997.
3. J. Giesl and T. Arts. Verification of Erlang processes by dependency pairs. *Applicable Algebra in Engineering, Communication and Computing*, 2000. To appear.
4. J. Giesl and E. Ohlebusch. Pushing the frontiers of combining rewrite systems farther outwards. volume 7 of *Studies in Logic and Computation*, pages 141–160. Wiley, 2000.
5. K. Kusakari and Y. Toyama. On proving AC-termination by AC-dependency pairs. Research Report IS-RR-98-0026F, School of Information Science, JAIST, 1998.

Semantic labeling meets dependency pairs

Enno Ohlebusch

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

Zantema [Zan95] devised the method of semantic labeling in which a given term rewriting system $(\mathcal{F}, \mathcal{R})$ is transformed into a labeled TRS $(\mathcal{F}_{lab}, \mathcal{R}_{lab})$ by labeling the function symbols. The labeling is always based on a quasi-model of $\mathcal{R}$ and it turns out that $\mathcal{R}$ is terminating if and only if $\mathcal{R}_{lab}$ is terminating. Semantic labeling is an interesting technique for proving termination because termination of $\mathcal{R}_{lab}$ is often much easier to prove than termination of $\mathcal{R}$; for example, a recursive path ordering might be applicable to $\mathcal{R}_{lab}$ but not to $\mathcal{R}$. The main result in [Zan95] reads as follows.

Theorem 1.1 A TRS $(\mathcal{F}, \mathcal{R})$ is terminating if and only if there exists a non-empty quasi-model $(\mathcal{M}, \succsim)$ of $(\mathcal{F}, \mathcal{R})$ and a labeling (L, lab) for $\mathcal{M}$ such that the TRS $(\mathcal{F}_{lab}, \mathcal{R}_{lab} \cup dec(\mathcal{F}_{lab}))$ is terminating, where

$$dec(\mathcal{F}_{lab}) = \bigcup_{f \in \mathcal{F}^{(n)}, n \in \mathbb{N}} \{f_a(x_1, \ldots, x_n) \to f_b(x_1, \ldots, x_n) \mid a >_f b\}.$$

However, Zantema [Zan95] remarks that "the technique of semantic labelling is hard to automate". In contrast to that, the dependency pair method of Arts and Giesl [AG00, AG98] can be automated. This fact constitutes the main advantage of the dependency pair method over semantic labeling. In the dependency pair approach a set IN of inequalities, each of the form $s \geq t$ or $s > t$, is generated and the existence of a quasi-ordering $\succsim$ satisfying these inequalities is sufficient for showing termination. More precisely, Arts and Giesl [AG98] showed the following theorem (note that here we use the stable-strict ordering corresponding to a quasi-ordering instead of its strict part; see [GO00]).

Theorem 1.2 A TRS $\mathcal{R}$ is terminating if and only if for each cycle $\mathcal{P}$ in the estimated dependency graph EDG($\mathcal{R}$) of $\mathcal{R}$, there is an argument filtering system (AFS) $\mathcal{A}$ and a quasi-reduction ordering $\succsim$ on $\mathcal{T}(\mathcal{F}', \mathcal{V})$ such that

- $l\downarrow_{\mathcal{A}} \succsim r\downarrow_{\mathcal{A}}$ for all rules $l \to r$ from $\mathcal{R}$,
- $s\downarrow_{\mathcal{A}} \succsim t\downarrow_{\mathcal{A}}$ for all dependency pairs $\langle s, t\rangle$ from $\mathcal{P}$,

- $s\downarrow_{\mathcal{A}} \succ t\downarrow_{\mathcal{A}}$ for at least one dependency pair $\langle s,t\rangle$ from $\mathcal{P}$,

where $\mathcal{F}'$ consists of all function symbols occurring in the inequalities.

The dependency pair method and semantic labeling can quite naturally be combined as we will show. We consider only finite TRSs over finite signatures and assume that every signature as well as every algebra is non-empty. Given a TRS $\mathcal{R}$, we apply the dependency pair method in order to show termination. Suppose for a cycle $\mathcal{P}$ in $\mathrm{EDG}(\mathcal{R})$ we have found an AFS $\mathcal{A}$ and a well-founded weakly monotone $\mathcal{F}$-algebra $(\mathcal{M},\succsim)$ such that

- $l\downarrow_{\mathcal{A}} \succsim_{\mathcal{M}} r\downarrow_{\mathcal{A}}$ for all rules $l\to r$ from $\mathcal{R}$,
- $s\downarrow_{\mathcal{A}} \succsim_{\mathcal{M}} t\downarrow_{\mathcal{A}}$ for all dependency pairs $\langle s,t\rangle$ from $\mathcal{P}$,

but there is no dependency pair $\langle s,t\rangle$ in $\mathcal{P}$ with $s\downarrow_{\mathcal{A}} \succ_{\mathcal{M}} t\downarrow_{\mathcal{A}}$ (if there were a dependency pair $\langle s,t\rangle\in\mathcal{P}$ with $s\downarrow_{\mathcal{A}} \succ_{\mathcal{M}} t\downarrow_{\mathcal{A}}$, then $\succsim_{\mathcal{M}}$ would be a quasi-reduction ordering satisfying the inequalities and we are done). Since we work with the stable-strict relation corresponding to $\succsim_{\mathcal{M}}$, this means that for every dependency pair $\langle s,t\rangle\in\mathcal{P}$ there is a ground substitution $\sigma\colon\mathcal{V}\to\mathcal{T}(\mathcal{F})$ with $[t\downarrow_{\mathcal{A}}\sigma]_{\mathcal{M}} \succsim [s\downarrow_{\mathcal{A}}\sigma]_{\mathcal{M}}$. Therefore, we have to take care of all assignments $\mu\colon\mathcal{V}\to M$ with $[\mu]_{\mathcal{M}}(s\downarrow_{\mathcal{A}})\approx[\mu]_{\mathcal{M}}(t\downarrow_{\mathcal{A}})$, where $\approx$ denotes the equivalence relation on M induced by $\succsim$. In this situation we try to use semantic labeling w.r.t. the labeling induced by the quasi-model $(\mathcal{M},\succsim)$. So in contrast to Theorem 1.1, $(\mathcal{M},\succsim)$ is assumed to be well-founded. Moreover, since the quasi-model is fixed, the method is amenable to automation.

Definition 1.3 A weakly monotone $\mathcal{F}$-algebra $(\mathcal{M},\succsim)$ is a *quasi-model* of a set of inequalities IN if $s\succsim_{\mathcal{M}} t$ for every $s>t$ and every $s\geq t$ in IN.

Given a quasi-model $(\mathcal{M},\succsim)$, we label the function symbols as follows. For every $f\in\mathcal{F}^{(n)}$, choose either $L_f=M/{\approx}$ or $L_f=\emptyset$. That is, function symbols are labeled by equivalence classes from $M/{\approx}$. More precisely, in case $L_f\neq\emptyset$, define $lab_f(m_1,\ldots,m_n)=\mathbf{m}$, where $\mathbf{m}$ denotes the equivalence class of $f_{\mathcal{M}}(m_1,\ldots,m_n)$. Thus, for every assignment $\mu\colon\mathcal{V}\to M$, the mapping $lab_\mu:\mathcal{T}(\mathcal{F},\mathcal{V})\to\mathcal{T}(\mathcal{F}_{lab},\mathcal{V})$ is defined as

$$lab_\mu(t)=\begin{cases} t & \text{if } t\in\mathcal{V},\\ f(lab_\mu(t_1),\ldots,lab_\mu(t_n)) & \text{if } t=f(t_1,\ldots,t_n) \text{ and } L_f=\emptyset,\\ f_{\mathbf{m}}(lab_\mu(t_1),\ldots,lab_\mu(t_n)) & \text{if } t=f(t_1,\ldots,t_n) \text{ and } L_f\neq\emptyset,\end{cases}$$

where $\mathbf{m}$ denotes the equivalence class of $[\mu]_{\mathcal{M}}(t)$.

Our main result is the following theorem.

Theorem 1.4 Let IN be a set of inequalities of the form $s>t$ or $s\geq t$, where $s,t\in\mathcal{T}(\mathcal{F},\mathcal{V})$. There exists a quasi-reduction ordering on $\mathcal{T}(\mathcal{F},\mathcal{V})$ satisfying the inequalities in IN if and only if there is a well-founded quasi-model $(\mathcal{M},\succsim)$ of IN, and a quasi-reduction ordering $\succsim_{lab}$ on $\mathcal{T}(\mathcal{F}_{lab},\mathcal{V})$ satisfying the inequalities in IN_{lab}, where

$$\begin{aligned}\mathrm{IN}_{lab} &= \{lab_\mu(s)>lab_\mu(t)\mid s>t\in\mathrm{IN},\mu\colon\mathcal{V}\to M,[\mu]_{\mathcal{M}}(s)\approx[\mu]_{\mathcal{M}}(t)\}\\ &\cup \{lab_\mu(s)\geq lab_\mu(t)\mid s\geq t\in\mathrm{IN},\mu\colon\mathcal{V}\to M,[\mu]_{\mathcal{M}}(s)\approx[\mu]_{\mathcal{M}}(t)\}\end{aligned}$$

Corollary 1.5 A TRS $\mathcal{R}$ is terminating if and only if for each cycle $\mathcal{P}$ in the estimated dependency graph of $\mathcal{R}$ there is an AFS $\mathcal{A}$ and a well-founded weakly monotone $\mathcal{F}'$-algebra $(\mathcal{M}, \succsim)$ satisfying

- $l\downarrow_{\mathcal{A}} \succsim_{\mathcal{M}} r\downarrow_{\mathcal{A}}$ for all rules $l \to r$ from $\mathcal{R}$,
- $s\downarrow_{\mathcal{A}} \succsim_{\mathcal{M}} t\downarrow_{\mathcal{A}}$ for all dependency pairs $\langle s, t\rangle$ from $\mathcal{P}$,

where $\mathcal{F}'$ consists of all function symbols occurring in the inequalities, and a quasi-reduction ordering $\succsim_{lab}$ on $\mathcal{T}(\mathcal{F}'_{lab}, \mathcal{V})$ satisfying

- $lab_\mu(l\downarrow_{\mathcal{A}}) \succsim_{lab} lab_\mu(r\downarrow_{\mathcal{A}})$ for every rule $l \to r \in \mathcal{R}$ and every assignment $\mu\colon \mathcal{V} \to M$ with $[\mu]_{\mathcal{M}}(l\downarrow_{\mathcal{A}}) \approx [\mu]_{\mathcal{M}}(r\downarrow_{\mathcal{A}})$,
- $lab_\mu(s\downarrow_{\mathcal{A}}) \succsim_{lab} lab_\mu(t\downarrow_{\mathcal{A}})$ for every dependency pair $\langle s, t\rangle$ from $\mathcal{P}$ and every assignment $\mu\colon \mathcal{V} \to M$ with $[\mu]_{\mathcal{M}}(s\downarrow_{\mathcal{A}}) \approx [\mu]_{\mathcal{M}}(t\downarrow_{\mathcal{A}})$,
- $lab_\mu(s\downarrow_{\mathcal{A}}) \succ_{lab} lab_\mu(t\downarrow_{\mathcal{A}})$ for at least one dependency pair $\langle s, t\rangle$ from $\mathcal{P}$ and every assignment $\mu\colon \mathcal{V} \to M$ with $[\mu]_{\mathcal{M}}(s\downarrow_{\mathcal{A}}) \approx [\mu]_{\mathcal{M}}(t\downarrow_{\mathcal{A}})$.

Proof: Direct consequence of Theorems 1.2 and 1.4.

By means of the preceding corollary, we are able to show termination of the TRS

$$\begin{array}{lcl} f(\lambda(x), y) & \to & \lambda(f(x, g(1, f(y, 2)))) \\ f(g(x, y), z) & \to & g(f(x, z), f(y, z)) \\ f(\lambda(x), y) & \to & f(x, \lambda(c(y))) \\ f(x, c(y)) & \to & f(c(x), y) \end{array}$$

which contains the first two rules of the system σ_0 and two new rules. The system σ_0 describes the process of substitution in combinatory categorical logic; cf. [Zan95].

References

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

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

[GO00] J. Giesl and E. Ohlebusch. Pushing the frontiers of combining rewrite systems farther outwards. In *Frontiers of Combining Systems 2 (Proceedings of the 2nd International Workshop on Frontiers of Combining Systems)*, pages 141–160. Research Studies Press Ltd., 2000.

[Zan95] H. Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24:89–105, 1995.

Busy Beaver PCPs

Mario Schmidt, Heiko Stamer, and Johannes Waldmann *

Institut für Informatik, Universität Leipzig
Augustusplatz 10-11, D-04109 Leipzig, Germany
pcp@informatik.uni-leipzig.de

1 Introduction

One way to investigate a problem is to look for small instances that are hard. The goal is to bound the hardness of the instance by a function of its size. (The existence of) such a function gives information on the problem's (decidability and) complexity. We apply this idea to the Post Correspondence Problem (PCP) [Pos46]. This resembles the hunt for Busy Beaver Turing machines [Mar00].

A *PCP instance* is a finite list $[(u_1, v_1), \ldots (u_n, v_n)]$ of pairs of words $u_i, v_i \in \Sigma^*$ that determines two morphisms $\phi, \psi : \Delta^* \to \Sigma^*$ with $\Delta = \{1, \ldots, n\}$ by $\phi(i) = u_i$ resp. $\psi(i) = v_i$. A *solution* of an instance is a member of the equality set $E(\phi, \psi) = \{w \mid w \in \Delta^+, \phi(w) = \psi(w)\}$. It is well known that it is undecidable whether a PCP instance has a solution.

The *size* of an instance is the number of its pairs (i. e. $|\Delta|$). We call PCP(n) the set of all instances of size n. It is known that solvability of PCP(7) is undecidable [MS96], while PCP(2) is decidable [EKR82]. We focus on PCP(3).

The *width* of an instance is the maximal length of a word $\max_{i \in \Delta}\{|u_i|, |v_i|\}$. We call PCP($n, w$) the set of all instances of size n and width w. We want to know, for fixed $n = 3$, how the maximal length of a minimal solution grows with w; with the aim of collecting information on the decidability of PCP(3).

In this note, we show some of the current record holding instances, then we describe a family of instances that seems to contain a lot of hard ones, and finally we discuss growth rates.

While we could not determine the status of PCP(3) so far, we think that our work nicely demonstrates techniques from combinatorics of words, (string) rewriting, and termination, applied to an interesting problem that is accessible to students, and thus can be used in teaching. In fact Stamer's implementation of the search algorithms started from a programming assignment.

Details will be elaborated in Schmidt's forthcoming diploma thesis, and an accompanying technical report. This abstract just collects results and conjectures.

* corresponding author

2 PCP(3) record holders

When looking for solutions of (randomly generated) instances, it is important to prune unsuccessful branches of the search tree, and to reject unsolvable instances early, without wasting too much time.

Moreover, note that it is not enough to find (or guess) and then verify a solution, since we also have to check that the solution has minimal length.

Some applicable techniques are explained (independently) in [Lor00], and we implemented some extensions. We found these PCP(3) instances of seemingly maximal hardness (the first one already is in [Lor00]):

instance	$\begin{pmatrix} 0 & 1 & 001 \\ 001 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 10 & 1101 \\ 0 & 1011 & 1 \end{pmatrix}$	$\begin{pmatrix} 01000 & 0 & 100 \\ 0 & 01001 & 0010 \end{pmatrix}$
length of min. sol.	75	132	182

The current list of record holders is maintained at our web site `http://www.informatik.uni-leipzig.de/~pcp/`. We welcome any addition, correction, or independent verification of entries of this list. Moreover, if you visit our web site, you can play an online PCP puzzle game. Each day, a fresh (semi-hard) PCP instance is generated automatically.

3 Morphic PCPs

We noticed that a lot of the hard instances seem to follow the simple pattern $M(u,v) = \begin{pmatrix} 0 & 1 & u \\ v & 0 & 1 \end{pmatrix}$, where v starts with 0, and u ends with 1. (For the record PCP(3,3) (see above), we even have $u = v$.)

We call these *morphic* instances, because if we only use the first two pairs, we obtain the infinite word that is the limit of the morphism $0 \mapsto v, 1 \mapsto 0$. We get a (different) infinite word by using the last two pairs only, from the morphism $0 \mapsto 1, 1 \mapsto \tilde{u}$. The question is how these words "meet".

A special case are *Fibonacci* instances $F(u) = M(u, 01)$, so named because the first two pairs give the infinite Fibonacci word $f = 01001010010\ldots$.

It is clear that for $F(u)$ to be solvable, u must be a factor of f that ends with 1. Is this condition also sufficient for solvability? We could verify this for $|u| \leq 6$, and some larger ones, and found rather hard instances among them. The table lists the lengths of the minimal solutions:

u	1	01	001	101	0101	1001	00101	01001	001001	100101	101001	0100101	0101001
l	1	3	30	44	6	78	8	80	120	36	108	55	22

Solvability of $F(1001001)$ is open.

4 Growth rates

As noted in [Lor00], the instance family $\begin{pmatrix} 0 & 10^n \\ 0^n1 & 0 \end{pmatrix}$ with minimal solutions A^nB^n achieves the fastest known growth (of the length of a minimal solution, seen as a function of the instance width) among PCP(2). Still there does not seem to be a simple proof for this — such a proof would imply a simple proof of decidability of PCP(2).

For PCP(3) families, we can achieve quadratic growth, by $\begin{pmatrix} 0 & 0^n & 1^n0 \\ 00 & 1 & 0 \end{pmatrix}$, with minimal solution $A^{n^2}B^nC$. A more non–obvious, but still only quadratic behaviour is exhibited by $\begin{pmatrix} 01 & 1^{2n}0 & 1 \\ 1 & 10 & 1010 \end{pmatrix}$, for suitable values of n:

n	1	2	3	4	5	6	7	8	9	10	11	12	13
l	4	10	32	22	84	170	240	46	208	522	128	280	760

Interestingly, the proof requires some number theory: the length of the solution depends on the number of incongruent values of 2^k modulo $2n-1$.

So far, we did not find larger growth for PCP(3) families (neither polynomials of higher degree, nor exponentials). This gives hope for decidability ...

References

[EKR82] A. Ehrenfeucht, J. Karhumaki, and G. Rozenberg. The (generalized) post correspondence problem with lists consisting of two words is decidable. *Theoretical Computer Science*, 21(2):119–144, November 1982.

[Lor00] Richard J Lorentz. Creating difficult instances of the post correspondence problem. 2nd International Conference on Computers and Games, Hamamatsu, 2000.

[Mar00] Heiner Marxen. Busy beaver. `http://www.drb.insel.de/~heiner/BB/index.html`, 2000.

[MS96] Yuri Matiyasevich and Géraud Sénizergues. Decision problems for semi-Thue systems with a few rules. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 523–531, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[Pos46] Emil Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52:264–268, 1946.

Inference of termination conditions for numerical loops

Alexander Serebrenik

Department of Computer Science, K.U. Leuven
Celestijnenlaan 200A, B-3001, Heverlee, Belgium
Email: Alexander.Serebrenik@cs.kuleuven.ac.be

Numerical computations form an essential part of almost any real-world program. Clearly, in order for a termination analyser to be of practical use it should contain a mechanism for inferring termination of such computations. However, this topic attracted less attention of the research community. In works of Dershowitz et al. [5] and Ruggieri [9] such termination analysis techniques were presented. In [8] a termination inference technique for constraints logic programming was suggested.

The study of termination for constraint logic programs [9] provide the necessary theoretical results and imply the equivalence of a variant of acceptability [1] and termination. Unfortunately, this work does not provide a methodology for actual proving termination or inferring conditions that imply it. Alternatively, Mesnard [8] provides techniques for inferring termination conditions, but does not consider inherently non well-founded CLP-domains.

In this paper we present a termination inference technique for numerical loops based on the well-known constraints based approach [4], further extending [3], and on the adornments technique [6, 7]. We restrict our interest only to integer loops, since termination of real number computations is often implementation dependent (see [5]).

Example 1.

$$p(X) \leftarrow X > 0, X \leq 5, X1 \textit{ is } X + 1, p(X)$$
$$p(X) \leftarrow X > 5$$

This example illustrates two possible sources of termination. First, if no rule is applicable for some query (e.g. ?–$p(-1)$), and second, if some rules are applicable, but the execution terminates(e.g. ?– $p(3)$ or ?– $p(7)$). We distinguish between these cases and infer termination conditions separately.

A condition ensuring termination in the first case is defined syntactically, based on the constraints appearing in rules bodies.

One of the major difficulties while analysing termination in the second case is defining correct level-mappings. On the one hand, level-mappings should map atoms to natural numbers. On the other hand, level-mappings should reflect the possibly negative numeric values of integer arguments. This contradiction is solved by step-by-step inference of bounded integer arguments and refining the definition of the level-mapping.

Each time a new definition of the level-mapping is suggested acceptability decreases are constructed as in [4]. Then, the system of constraints is solved and it either provides some constraints on integer variables, that are assumed to hold, and, thus, extends a set of bounded variables on which level-mapping can be based, or defines a level-mapping completely, thus, proving termination. The termination condition in the latter case is constructed from all the constraints on integer arguments assumed so far.

The presented technique is robust enough to analyse correctly examples such as *sqrt* [9], *between* and *range* [10], *towers of Hanoi* [2], *Hilbert* and *Sierpinski curves* [11], *bad loops* [5] or belonging to programming classics *factorial*, *gcd* and *fibonacci*.

Finally, as future work we consider implementing the methodology and estimating its power by benchmarking.

References

1. K. R. Apt and D. Pedreschi. Studies in Pure Prolog: Termination. In J. W. Lloyd, editor, *Proc. Esprit Symp. on Comp. Logic*, pages 150–176. Springer Verlag, 1990.
2. W. F. Clocksin and S. Mellish, Christopher. *Programming in Prolog.* Springer Verlag, 1981.
3. S. Decorte and D. De Schreye. Termination analysis: some practical properties of the norm and level mapping space. In J. Jaffar, editor, *Proc. of the 1998 Joint Int. Conf. and Symp. on Logic Programming*, pages 235–249. MIT Press, June 1998.
4. S. Decorte, D. De Schreye, and H. Vandecasteele. Constraint-based termination analysis of logic programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(6):1137–1195, November 1999.
5. N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Appl. Algebr. Eng. Commun. Comput.*, 2000. accepted.
6. A. Y. Levy and Y. Sagiv. Constraints and redundancy in datalog. In *Proc. of the Eleventh ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 67–80. ACM Press, 1992.
7. N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. Unfolding mystery of the *mergesort*. In N. Fuchs, editor, *Proc. of the Seventh Int. Workshop on Logic Program Synthesis and Transformation*. Springer Verlag, 1998. LNCS 1463.
8. F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs. In M. Maher, editor, *Proc. JICSLP'96*, pages 7–21. The MIT Press, 1996.
9. S. Ruggieri. Termination of constraint logic programs. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium, ICALP'97*, pages 838–848. Springer Verlag, 1997. LNCS 1256.
10. L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 1994.
11. N. Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, 1976.

On termination of meta-programs

Alexander Serebrenik

Department of Computer Science, K.U. Leuven
Celestijnenlaan 200A, B-3001, Heverlee, Belgium
Email: Alexander.Serebrenik@cs.kuleuven.ac.be

The term *meta-programming* refers to the ability of writing programs that have other programs as data and exploit their semantics [1]. The choice of logic programming as a basis for meta-programming offers a number of practical and theoretical advantages. One of them is the possibility of tackling critical foundation problems of meta-programming within a framework with a strong theoretical basis. Another is the surprising ease of programming. These reasons motivated an intensive research on meta-programming inside the logic programming community [1, 8, 10, 11]. See also [6] for a survey.

On the other hand, termination analysis is one of the most intensive research areas in logic programming as well. See [3] for the survey. More recent work on this topic can be found among others in [4, 5, 7, 9, 12, 14–16].

Traditionally, termination analysis of logic programs have been done either by the "transformational" approach or by the "direct" one. A transformational approach first transforms the logic program into an "equivalent" term-rewrite system (or, in some cases, into an equivalent functional program). Here, equivalence means that, at the very least, the termination of the term-rewrite system should imply the termination of the logic program, for some predefined collection of queries[1]. Direct approaches do not include such a transformation, but prove the termination directly on the basis of the logic program. In [13] we have developed an approach that provides the best of both worlds: a means to incorporate into "direct" approaches the generality of general term-orderings.

As we show in this paper, the suggested technique is very useful for proving termination of different meta-interpreters together with different classes of object programs. Our research has been motivated by the famous "vanilla" meta-interpreter, undoubtly belonging to logic programming classics.

Example 1.

$$solve(true).$$
$$solve((Atom, Atoms)) \leftarrow solve(Atom), solve(Atoms).$$
$$solve(Head) \leftarrow clause(Head, Body), solve(Body).$$

Termination of this meta-interpreter, presented in Example 1, have been studied by Pedreschi and Ruggieri. They proved, that it improves termination

[1] The approach of Arts [2] is exceptional in the sense that the termination of the logic program is concluded from a weaker property of *single-redex normalisation* of the term-rewrite system.

(Corollary 40, [11]). However, we can claim more—"vanilla" not just improves termination, but preserves it.

In order for meta-interpreters to be useful in applications they should be able to cope with a reacher language than the "vanilla" meta-interpreter, including, for example, negation. Moreover, typical applications of meta-interpreters, such as debuggers, will also require producing some additional output or performing some additional tasks during the execution, such as constructing proof trees or cutting "unlikely" branches for uncertainty reasoner with cutoff. These extensions can and usually will influence termination properties of the meta-interpreter.

By extending the suggested technique [13] to normal programs, we are able to perform the correct analysis of a number of possible extended meta-interpreters, performing tasks as described above. We identify most popular classes of meta-interpreters, such as *extended meta-interpreters* [10], and using this extended technique prove that termination is usually improved. We also state some more generic conditions implying preservation of termination. Moreover, the same extension of the technique allows us to perform termination analysis of Situation Calculus applications.

Finally, we state yet other meta-interpreters for which performing termination analysis would be an interesting issue for further research.

References

1. K. R. Apt and F. Turini, editors. *Meta-Logics and Logic Programming.* Logic Programming. The MIT Press, 1995.
2. T. Arts. *Automatically proving termination and innermost normalisation of term rewriting systems.* PhD thesis, Universiteit Utrecht, 1997.
3. D. De Schreye and S. Decorte. Termination of logic programs: The never-ending story. *J. Logic Programming*, 19/20:199–260, May/July 1994.
4. S. Decorte and D. De Schreye. Termination analysis: some practical properties of the norm and level mapping space. In J. Jaffar, editor, *Proc. of the 1998 Joint Int. Conf. and Symp. on Logic Programming*, pages 235–249. MIT Press, June 1998.
5. S. Decorte, D. De Schreye, and H. Vandecasteele. Constraint-based termination analysis of logic programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(6):1137–1195, November 1999.
6. P. Hill and J. Gallagher. Meta-programming in logic programming. In D. M. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of logic in Artificial Intelligence and Logic Programming*, pages 421–498. Clarendon press, 1998. volume 5. Logic Programming.
7. S. Hoarau. *Inférer et compiler la terminaison des programmes logiques avec contraintes.* PhD thesis, Université de La Réunion, 1999.
8. G. Levi and D. Ramundo. A formalization of metaprogramming for real. In D. S. Warren, editor, *Logic Programming, Proceedings of the Tenth International Conference on Logic Programming*, pages 354–373. MIT Press, 1993.
9. N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In L. Naish, editor, *Proc. of the Fourteenth Int. Conf. on Logic Programming*, pages 63–77. MIT Press, July 1997.

10. B. Martens and D. De Schreye. Why untyped nonground metaprogramming is not (much of) a problem. *Journal of Logic Programming*, 22(1):47–99, January 1995.
11. D. Pedreschi and S. Ruggieri. Verification of meta-interpreters. *Journal of Logic and Computation*, 7(2):267–303, November 1997.
12. S. Ruggieri. *Verification and validation of logic programs.* PhD thesis, Universitá di Pisa, 1999.
13. A. Serebrenik and D. De Schreye. Non-transformational termination analysis of logic programs, based on general term-orderings. In K.-K. Lau, editor, *Pre-Proceedings of Tenth International Workshop on Logic-based Program Synthesis and Transformation, 2000*, pages 45–54. University of Manchester, 2000. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-00-6-1, URL : http://www.cs.man.ac.uk/cstechrep/titles00.html.
14. J.-G. Smaus. *Modes and Types in Logic Programming.* PhD thesis, University of Kent, 1999.
15. C. Taboch. A semantic basis for termination analysis of logic programs. Master's thesis, Ben-Gurion University of the Negev, 1998.
16. S. Verbaeten. *Static verification of compositionality and termination for logic programming languages.* PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, June 2000. v+265+xxvii.

On a Control-Theoretic Approach for Proving Termination

Oliver Theel

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

Phone: [49] (6151) 16-5306
Fax: [49] (6151) 16-5410
Email: theel@informatik.tu-darmstadt.de

Keywords: Termination, Self-Stabilization, Convergence, Control Theory, Ljapunov Equation, Ljapunov Function

Self-stabilization is a powerful property of a distributed algorithm: a self-stabilizing distributed algorithm is guaranteed to reach a pre-defined set of so-called *legitimate states* in finite time, regardless it its initial system state. Once in a legitimate state, a self-stabilizing distributed algorithm does not voluntarily leave the set of legitimate states. Only in case of faults, leading to a perturbation of the system state, the algorithm may switch to a so-called *illegitimate state*.[1] Being in an illegitimate state, the algorithm re-enters a legitimate state as soon as new faults cease to appear and errors in the system state as well as failed components have been corrected or repaired. Thus, self-stabilizing distributed algorithms tolerate – although in a non-masking manner – any number and distribution of *transient* faults [3]. Furthermore, self-stabilizing distributed algorithms do not need to be correctly initialized, since an arbitrary, improper initialization corresponds to an illegitimate state. From there, as described above, the algorithm will autonomously converge to a legitimate state. Due to these advantages, self-stabilizing distributed algorithms are subject to excessive research focusing on design as well as verification techniques.

The verification of a self-stabilizing distributed algorithm is generally done in two steps. In one step, *convergence* into the set of legitimate state in finite time must be shown. In another step, the *closure* of the set of legitimate states must be proven. Whereas proving closure is easy, proving convergence is much more challenging. In fact, proving convergence is an instance of proving termination, since for any self-stabilizing algorithm $\mathcal{A}$ an alternative algorithm $\mathcal{B}$ can be given that repeatedly executes $\mathcal{A}$'s program actions in a loop while a certain state predicate P is false. This state predicate evaluates to true if and only if the current state of the system is identical to a legitimate state. Consequently, through $\mathcal{B}$'s termination proof follows $\mathcal{A}$'s convergence proof and vice versa. Thus, all methods for proving termination and convergence that exist in self-stabilization and termination literature can be applied. However, the existing methods all boil down to a well-foundedness argument: One has to devise a system state function mapping the system state into a domain for which a total order among the entities can be given and where a smallest domain element with respect to the total ordering exists. Building on this function, the proof must show that while the system performs subsequent state transitions, the function value decreases with respect to the total ordering. This guarantees that after a finite number of state transitions the function value is minimal. If the corresponding system state is a legitimate state then convergence has been proven. Although it is obvious how to perform a convergence proof using such a termination function (which is called *convergence function* in self-stabilization literature), it is be no means easy to devise such a function since

[1] A system state is either a legitimate state or an illegitimate state.

it somehow captures "the very essence of convergence" of the algorithm. Techniques to systematically identify those functions are therefore highly looked for.

Self-stabilizing algorithms exhibit certain analogies with stable feedback systems used in electrical and mechanical engineering. In contrast to self-stabilizing algorithms which present a relatively young area of research in computer science, feedback systems research dates back for more than a century. Not surprisingly, feedback system theory offers a variety of methods for reasoning about system stability. The basic idea of our research is to investigate to what extend feedback system theory can be used for reasoning about the self-stabilization property of algorithms.

As a first result, we could identify a system model for so-called *discrete-time non-linear dynamic variable structure feedback systems* [5] that allows the modeling of self-stabilizing algorithms in terms of feedback system terminology. This system model can be seen as connector of two research areas: it bridges the gap between self-stabilizing algorithms and feedback systems. As a consequence, based on the model, a powerful criterion of control theory, known as *Ljapunov's "Second Method"* [1, 2] can directly be adopted for reasoning about system stability and, thus, for the reasoning about the self-stabilization property of a modeled algorithm. If a so-called *Ljapunov equation* for a given system can be solved then a convergence function (called *Ljapunov function*) is guaranteed to exists and is identified in the course of the calculation. Unfortunately, the Ljapunov equation cannot always straightforwardly be solved,[2] but it does simplify convergence function identification for many self-stabilizing algorithms. For instance, for self-stabilizing algorithms that can be modeled as *linear* time-discrete dynamic dynamic feedback systems (with and without variable structure), a standard starting point always solves the Ljapunov equation leading to an identification of a suitable Ljapunov function. But even in the non-linear case, Ljapunov theory offers many good starting points for solving the Ljapunov equation or for direct identification of a Ljapunov function.

Interestingly, it turned out that the transfer function approach (presented at WST'99, see [4] for details) seamlessly fits into our new approach: through the system model and Ljapunov theory we are now able to identify a termination function solely based on a given transfer function.

In the talk, we would like to present the control-theoretic approach for identifying termination functions as sketched in this abstract. We plan to illustrate the usefulness of the approach through a sample algorithm whose convergence property is to be proven. We will show how to model the algorithm as an instance of the system model and how to systematically identify a suitable Ljapunov function. Finally, we would like to elaborate on the potentials and limitations of the presented technique.

References

[1] R. E. Kalman and J. E. Bertram. Control System Analysis and Design Via the "Second Method" of Lyapunov - Part II: Discrete-Time Systems. *Transactions of the ASME, Journal of Basic Engineering*, 82:394–400, June 1960.

[2] M. A. Ljapunov. Problème général de la stabilité du movement. *Ann. Fac. Sci. Toulouse*, 9:203–474, 1907.

[3] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, 1993.

[4] O. Theel and F. C. Gärtner. On Proving Convergence through Transfer Functions (Extended Abstract). In Jürgen Giesl, editor, *Proc. of the 4th International Workshop on Termination (WST'99), Dagstuhl, Germany*, pages 46–47, May 1999.

[5] V I. Utkin. Variable Structure Systems with Sliding Modes. *IEEE Transactions on Automatic Control*, 22:212–222, 1977.

[2] E.g., for instable systems the equation can, of course, never be solved, since the existence of a Ljapunov function implies system stability.

Proving Termination Automatically and Incrementally

Xavier Urbain

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

In order to provide a structural and hierarchical approach of TRS and to run termination proofs incrementally, we introduce notions of *rewriting modules* and *relative dependency pairs* built upon these modules.

Definition 1. *Let R_1 be a TRS over $\mathcal{F}_1$. A* module extending R_1 *is a couple $(\mathcal{F}_2 \mid R_2)$ such that: $\mathcal{F}_1 \cap \mathcal{F}_2 = \emptyset$; R_2 is a TRS over $\mathcal{F}_1 \cup \mathcal{F}_2$; for each $l \to r \in R_2$, $\Lambda(l) \in \mathcal{F}_2$.*

System $R_1 \cup R_2$ over $\mathcal{F}_1 \cup \mathcal{F}_2$ is then a hierarchical extension of system $R_1(\mathcal{F}_1)$ by module $(\mathcal{F}_2 \mid R_2)$.

Definition 2. *Let $(\mathcal{F}_2 \mid R_2)$ be a module hierarchically extending a TRS R_1. For each rule $l \to r \in R_2$, a pair $\langle l, r' \rangle$ where r' is a subterm of r such that $\Lambda(r') \in \mathcal{F}_2$ is called a* dependency pair *of module $(\mathcal{F}_2 \mid R_2)$.*

The set of dependency pairs of all rules of a module M is denoted $DP(M)$.

Dependency pairs of $(\mathcal{F} \mid R)$ are then given relatively to $\mathcal{F}$ only.

Important results concerning Cε termination (that is termination stable under addition of $\pi = \{G(x,y) \to x;\ G(x,y) \to y\}$) [2–4] can be expressed in that framework.

Theorem 1. *Let $R_1(\mathcal{F}_1)$ be a TRS and let $(\mathcal{F}_2 \mid R_2)$ be a module extending R_1. If R_1 is Cε terminating and if there is no infinite dependency chain of $(\mathcal{F}_2 \mid R_2)$ over $R_1 \cup R_2$, then $R_1 \cup R_2$ is strongly normalizing.*

Theorem 2. *Let $R_1(\mathcal{F}_1)$ be a TRS, let $(\mathcal{F}_2 \mid R_2)$ and $(\mathcal{F}_3 \mid R_3)$ be two modules extending R_1 with $\mathcal{F}_2 \cap \mathcal{F}_3 = \emptyset$.*
If $R_1 \cup R_2$ Cε terminates and if there is no infinite dependency chain of $(\mathcal{F}_3 \mid R_3)$ over $R_1 \cup R_3 \cup \pi$, then $R_1 \cup R_2 \cup R_3$ Cε terminates.

Note that R_2 **does not interfere** with the premise over $(\mathcal{F}_3 \mid R_3)$.

Both theorems can be turn into effective new incremental methods for termination proof. Since they are well suited for automation, we propose an implementation of modular criteria inside system C*i*ME2 [1]. Termination proofs can now be performed automatically (for both ordering search and modular strategy) in an incremental fashion as illustrated by example below.

Example Let us build a system supposed to compute logarithms of numbers in binary notation. What we need is firstly a basic binary representation of numbers that is:

$$\begin{array}{l}\mathcal{F}_{\#}\ \{\# : \textit{constant}; 1, 0 : \textit{postfix unary}\}\\ R_{\#}\ \{\#0 \to \#\end{array}$$

We then need some arithmetic functions over those numbers, at least addition in $(\mathcal{F}_+ \mid R_+)$:

$$\begin{array}{l}\mathcal{F}_+\ \{+ : \textit{binary}\}\\ R_+ \left\{\begin{array}{ll} x + \# \to x, & \# + x \to x,\\ x0 + y0 \to (x + y)0, & x0 + y1 \to (x + y)1,\\ x1 + y0 \to (x + y)1, & x1 + y1 \to (x + (y + \#1))0.\end{array}\right.\end{array}$$

Termination of $R_{\#} \cup R_+$ is proven using relative dependency pairs and polynomial interpretation.

$$\mathrm{DP}((\mathcal{F}_+ \mid R_+)) : \left\{\begin{array}{l}\langle x0 + y0, x + y\rangle\\ \langle x0 + y1, x + y\rangle\\ \langle x1 + y0, x + y\rangle\\ \langle x1 + y1, y + \#1\rangle\\ \langle x1 + y1, x + (y + \#1)\rangle\end{array}\right\} \quad \begin{array}{l}[[\#]] = 0\\ [[0]](x) = [\![x]\!] + 1\\ [[1]](x) = [\![x]\!] + 1\\ [[+]](x, y) = [\![x]\!] + [\![y]\!]]\end{array}$$

Since $\mathrm{DP}((\mathcal{F}_+ \mid R_+))$ strictly decreases and $R_{\#} \cup R_+ \cup \pi$ weakly decreases, there is no infinite chain of $(\mathcal{F}_+ \mid R_+)$ over $R_{\#} \cup R_+ \cup \pi$.

We need also substraction, providing thus another module $(\mathcal{F}_- \mid R_-)$:

$$\begin{array}{l}\mathcal{F}_-\ \{- : \textit{binary}\}\\ R_- \left\{\begin{array}{ll} x - \# \to x, & \# - x \to \#,\\ x0 - y0 \to (x - y)0, & x0 - y1 \to ((x - y) - \#1)1,\\ x1 - y0 \to (x - y)1, & x1 - y1 \to (x - y)0.\end{array}\right.\end{array}$$

Termination proof of $(\mathcal{F}_- \mid R_-)$ extending $R_{\#}$ is performed using same interpretation for $\#$, 0 and 1, and taking $[[-]](x, y) = [\![x]\!]]$ From theorem 2, $R_{\#} \cup R_+ \cup R_-$ Cε terminates.

We shall use Boolean operations ($R_\neg$), and then comparisons over integers (presented by $R_{\#}$) thus extending both $R_\neg$ and $R_{\#}$:

$$\begin{array}{l}\mathcal{F}_\neg\ \{\textit{true}, \textit{false} : \textit{constants}\ ;\ \neg : \textit{unary}\ ;\ \textit{if} : \textit{ternary}\}\\ R_\neg \left\{\begin{array}{ll} \neg(\textit{true}) & \to \textit{false}\\ \neg(\textit{false}) & \to \textit{true}\\ \textit{if}(\textit{true}, x, y) & \to x\\ \textit{if}(\textit{false}, x, y) & \to y\end{array}\right.\end{array}$$

$$\begin{array}{l}\mathcal{F}_{ge}\ \{ge : \textit{binary}\}\\ R_{ge} \left\{\begin{array}{ll} ge(x0, y0) & \to ge(x, y)\\ ge(x0, y1) & \to \neg ge(y, x)\\ ge(x1, y0) & \to ge(x, y)\\ ge(x1, y1) & \to ge(x, y)\\ ge(x, \#) & \to \textit{true}\\ ge(\#, x0) & \to ge(\#, x)\\ ge(\#, x1) & \to \textit{false}\end{array}\right.\end{array}$$

$(\mathcal{F}_\neg \mid R_\neg)$ has no DP. For polynomial interpretation

$$[\![\begin{array}{lll} [\![\#]\!] = 0, &](\,[\![0]\!]) = [\![x]\!] + 1, &](\,[\![1]\!]) = [\![x]\!] + 1, \\ [\![true]\!] = 0, & [\![false]\!] = 0, & [\![\neg]\!](x) = 0, \\ [\![if]\!](x,y,z\,) = [\![y]\!] + [\![z]\!], & [\![ge]\!](x,y\,) = [\![x]\!] + [\![y]\!], & \end{array}$$

dependency pairs of $(\mathcal{F}_{ge} \mid R_{ge})$ strictly decrease while rules of $R_\# \cup R_\neg \cup R_{ge}$ weakly decrease. We may then apply Theorem 2 and conclude that $R_\# \cup R_+ \cup R_- \cup R_\neg \cup R_{ge}$ is Cε terminating.

Now we can consider firstly $(\mathcal{F}_{Log'} \mid R_{Log'})$ and then $(\mathcal{F}_{Log} \mid R_{Log})$ extending it and computing logarithms.

$$\mathcal{F}_{Log'}\ \{Log' : unary\}$$
$$R_{Log'} \begin{cases} Log'(\#) \to \# \\ Log'(x1) \to Log'(x) + \#\ 1 \\ Log'(x0) \to if(ge(x, \#1), Log'(x) + \#\ 1, \#) \end{cases}$$

$$\mathcal{F}_{Log}\ \{Log : unary\}$$
$$R_{Log}\ \{\, Log(x) \to Log'(x) - \#1$$

$$\mathrm{DP}((\mathcal{F}_{Log'} \mid R_{Log'}))\ \left\{ \begin{array}{l} \langle Log'(x1), Log'(x) \rangle \\ \langle Log'(x0), Log'(x) \rangle \end{array} \right\}$$

For the polynomial interpretation used for $R_\# \cup R_+$ extended with

$$\begin{array}{lll} [\![\neg]\!](x) = 0, & [\![true]\!] = 0, & [\![false]\!] = 0, \\ [\![ge]\!](x) = 0, & [\![if]\!](x,y,z\,) = [\![y]\!] + [\![z]\!], & [\![Log']\!](x) = [\![x]\!], \end{array}$$

dependency pairs of $(\mathcal{F}_{Log'} \mid R_{Log'})$ stricly decrease while rules of $R = R_\# \cup R_+ \cup R_\neg \cup R_{ge} \cup R_{Log'}$ weakly decrease, thus R Cε terminates. Finally, Module $(\mathcal{F}_{Log} \mid R_{Log})$ extending $(\mathcal{F}_{Log'} \mid R_{Log'})$ and $(\mathcal{F}_- \mid R_-)$ has no dependency pair so no chain exists.

We can then conclude that $R_\# \cup R_+ \cup R_- \cup R_\neg \cup R_{ge} \cup R_{Log'} \cup R_{Log}$ is Cε strongly normalizing.

References

1. E. Contejean, C. Marché, B. Monate, and X. Urbain. Cime version 2, 2000. Prerelease available at `http://www.lri.fr/~demons/cime.html`.
2. B. Gramlich. A structural analysis of modular termination of term rewriting systems. Research Report SR-91-15, University Kaiserslautern, 1991.
3. B. Gramlich. Generalized sufficient conditions for modular termination of rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 5:131–158, 1994.
4. E. Ohlebusch. Modular properties of composable term rewriting systems. *Journal of Symbolic Computation*, 20:1–41, 1995.

Binding-time Annotations without Binding-time Analysis
(Extended Abstract)

Wim Vanhoof and Maurice Bruynooghe

Department of Computer Science,
K.U.Leuven, Belgium.
e-mail: {wimvh,maurice}@cs.kuleuven.ac.be

Abstract. In this work, we argue that the use of termination conditions as unfolding criteria is insufficient to obtain adequate specialisation of logic programs. We describe how a binding-time analysis for logic programs that allows more liberal unfoldings can be obtained by altering termination analysis in such a way that it proves termination *at specialisation-time* rather than *at run-time*.

1 Introduction and Motivation

Partial evaluation is a well-studied source-to-source transformation, capable of specialising a program P with respect to a part s of its input. The result is a program P_s that computes, when provided with the remaining part d of the input, the same result as the original program P on the complete input $s + d$. The general effect of partial evaluation is that the computations performed by a program are *staged*: some (ideally all) operations in P that depend only on s are performed by the specialiser; the remaining computations (those depending on d) by the residual program P_s. Partial evaluation can be used to speed up the computation of a program, in particular when the program must be run a number of times while part of its input (the part denoted by s) remains constant. Indeed, using partial evaluation, the latter computations must be performed only once to construct P_s, which can then be run any number of times with different inputs $d_1, \ldots, d_n$.

The heart of any partial evaluator is an evaluation mechanism for the language under consideration. In a logic programming setting, "evaluation" of a program corresponds to building an SLD-tree for a program/query pair $\langle P, Q \rangle$. If the program terminates, the corresponding SLD-tree is finite. In this setting, *partially available* input corresponds to a query Q' that is less instantiated than Q. Due to the nature of logic programming, the program could, in principle, simply be evaluated with respect to Q'. Most likely, however, the SLD-tree built for $\langle P, Q' \rangle$ will be infinite. Indeed, if the control flow is determined by a value that is unknown in $\langle P, Q' \rangle$, SLD-derivations of infinite length may be created resulting in a non-terminating specialisation process. Instead of building such a possibly infinite SLD-tree, a partial evaluator for logic programs builds a finite number of *finite* SLD-trees that together cover the complete computation for $\langle P, Q' \rangle$ [5]. The resulting SLD-trees are *partial*, in the sense that, while building the SLD-tree, the partial evaluator unfolds some predicate calls whereas it does not unfold others. The predicate calls that are not unfolded are said to be *residualised* – they will appear as code in the residual program.

Most work on partial evaluation in logic programming concentrates on the so-called *on-line* approach to partial evaluation [4]: during the construction of a partial SLD-tree, the partial evaluator selects each call occurring in an SLD-derivation and decides whether or not to unfold it; usually basing its decision on the structure of the SLD-tree built so far. In the *off-line* approach on the other hand, the program is first analysed by a so-called binding-time analysis (BTA). Binding-time analysis is a global analysis that takes a program and (an abstraction of) the query and generates an *annotated* version of the original program, in which every predicate call is accompanied by an instruction stating whether or not instances of this call must be unfolded. The actual specialiser builds the partial SLD-trees simply by following the instructions generated by BTA.

In general, on-line systems tend to achieve better specialisation results, since they consider each call occurring in an SLD-derivation in isolation. This differs from the off-line approach, in

which a control decision is associated to a syntactic occurrence of a call in the program, based on an abstraction of the concrete input. Off-line systems, however, also offer a number of advantages over on-line systems. First, the separation of the process in a binding-time analysis followed by a specialisation phase makes the process conceptually easier to reason about, and results in a fairly simple (and efficient!) specialiser from which the burden of continuously monitoring the evaluation process has been removed. Also, the analysis output can be represented by annotations on the original source program, and provide as such excellent feedback to the user providing clues to why an optimisation was (not) performed. In spite of these advantages and the extensive work on off-line partial evaluation in other paradigms, only few efforts have been made to construct an off-line partial evaluation system for logic programming [1].

The basic task of binding-time analysis is to mark as much predicate calls as possible *unfoldable* in the program while guaranteeing that building a partial SLD-tree by following the annotations is a terminating process. Termination of the process is guaranteed if only calls that are guaranteed to terminate are marked *unfoldable*. This is basically the approach taken in [1]: termination analysis (either by hand, or by an automatic system) is used to derive conditions – expressed in terms of availability of a predicate's arguments – under which a call to the particular predicate is guaranteed to terminate. Next, the program is analysed by a classical groundness analysis [6], the results of which (expressing what arguments in a predicate call are available) are combined with the termination conditions to settle the unfolding conditions during analysis. However, appealing as the approach may seem, the use of *run-time* termination of a call as an unfold condition imposes considerable restrictions on the unfolding possibilities. The fact that a call is marked unfoldable only in case it terminates under normal evaluation implies that only calls that can *completely* be unfolded to *true* or *fail* are unfolded. In other words, a specialiser relying on the result of such a binding-time analysis is restricted to building *complete* SLD-trees, rather than *partial* SLD-trees. We illustrate this limitation with the following example.

Example 1. Consider the meta interpreter depicted in the left-hand side of Fig. 1. The interpreter has two predicates implementing the classical `member/2` and `append/3` predicates as object program. Suppose we want to annotate this program with respect to the query `solve([mem(X,Xs)])`.

Meta interpreter	Specialised meta interpreter
`solve([]).`	`solve([mem(X,Xs)]):- solve_atom(mem(X,Xs)).`
`solve([A|Gs]):- solve_atom(A), solve(Gs).`	
	`solve_atom(mem(X,Xs)):-`
`solve_atom(A):- cl(A,Body), solve(Body).`	`  solve_atom(app(A,[X|_], Xs)).`
`cl(mem(X,Xs), [app(_,[X|_],_Xs)]).`	`solve_atom(app([],[X|B],[X|B])).`
`cl(app([],L,L), []).`	`solve_atom(app([E|Es],[X|B],[Z|Zs])):-`
`cl(app([X|Xs],Y,[Z|Zs]),[app(Xs,Y,Zs)]).`	`  solve_atom(app(Es, [X|B], Zs)).`

Fig. 1. Vanilla meta interpreter

Using termination as a criterium for unfolding results in no calls to `solve/1` at all being marked unfoldable, and practically no specialisation would be obtained. Indeed, all calls to `solve/1` arising during unfolding would be of the form `solve([mem(X,Xs)])` or `solve([app(A,[X|_],Xs)])`, none of which is terminating.

2 From Termination Analysis to Binding-time Analysis

In this work, we devise a binding-time analysis that allows more liberal unfoldings than those based on termination conditions *tout-court*. The general idea is to construct step-wise an annotated version of the program and to use termination analysis to prove that building a *partial* SLD-tree for an initial goal with respect to the *annotated program under construction* (as opposed to the

original program) terminates. Hence, the process of building a partial SLD-tree according to the annotations terminates.

Example 2. Reconsider the vanilla interpreter from Example 1. Intuitively, from a specialisation point of view, we can see that it is perfectly safe to unfold all calls to the `solve/1` predicate as long as the intermediate calls to `solve_atom/1` are residualised. The idea is that the `solve/1` predicate in a sense only performs the parsing of an object goal (deconstructing a list of atoms), which is terminating in Example 1. Thus, residualising the calls to `solve_atom/1` and unfolding the others results in the specialised program depicted in the right-hand side of Fig. 1 which corresponds, modulo a standard structure filtering transformation [3] with the standard definitions for the `append/3` and `member/2` predicates. Hence, all meta interpretation overhead has been removed.

In order to construct such a binding-time analysis, the key observation is the following: the termination behaviour of building a partial SLD-tree for a query Q with respect to an *annotated* program P_{ann} is equivalent with the termination behaviour of building a complete SLD-tree for a program that is derived from P by *removing* those calls that are annotated to residualise in P_{ann}, keeping only those that are annotated unfoldable in P_{ann}. Our binding-time analysis then proceeds as follows: Assume we have to annotate a program P with respect to an initial query Q. If termination of Q with respect to P can be proven by an automatic termination analysis, all calls in P can safely be annotated unfoldable. As a result, every call occurring during specialisation of Q will be unfolded, and specialisation of Q boils down to plain evaluation of Q in P (constructing a complete SLD-tree for $\langle P, Q \rangle$). If, on the other hand, termination of Q can not be proven, due to the presense of a "suspect" call in P, a new program is derived from P by *deleting* the particular call from P. Once more, termination analysis is performed on the transformed program, and the process is repeated until enough calls are removed from the program such that it terminates. Deleting a call from the program corresponds with marking the particular call to residualise in the annotated program: the call under consideration is not unfolded, and no bindings are created by it.

Such a binding-time analysis can easily be implemented starting from a termination analysis that is capable of identifying "problematic" calls, i.e. those calls due to which the analysis was unable to prove termination of a goal. An example of such a termination analysis is [2]. The analysis constructs a finite approximation of the *binary unfoldings semantics* of the program under consideration. The binary unfoldings semantics of a program consists of a set of binary clauses that relates the head atom with a particular body atom of one of the program's clauses. To prove termination of a query Q with respect to a program P, it suffices to show [2] that a particular condition holds on (some of) the binary clauses constructed by the termination analysis. If the condition does not hold on a particular binary clause, its single body atom identifies the problematic call in the original program.

References

1. Maurice Bruynooghe, Michael Leuschel, and Kostis Sagonas. A polyvariant binding-time analysis for off-line partial deduction. In C. Hankin, editor, *Programming Languages and Systems, Proc. of ESOP'98, part of ETAPS'98*, pages 27–41, Lisbon, Portugal, 1998. Springer-Verlag. LNCS 1381.
2. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
3. J. Gallagher and M. Bruynooghe. Some low-level source transformations for logic programs. In M. Bruynooghe, editor, *Proceedings Meta'90*, pages 229–244, Leuven, April 1990.
4. M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1), 1998.
5. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11(3&4):217–242, 1991.
6. Kim Marriott and Harald Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM letters on Programming Languages and Systems*, 2(1-4):181–196, 1993.

Mathematical analysis of some termination principles

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

Abstract

In this survey talk we start with recalling some celebrated combinatorial independence results. These include Friedman's miniaturization of the well-foundedness of ε_0, Friedman's miniaturization of Kruskal's theorem (KT) and Kirby's and Paris' results on Goodstein sequences and Hydra battles. We then give a mathematical classification of these principles in terms of growth rate conditions using recursion theory and methods from analytic combinatorics, a field which is familiar in computer science from the average case analysis of algorithms.

We relate Otter's tree constant 2.95576... to Friedman's miniaturization of KT and obtain the following refinement [to appear in JSL] of a result by Loebl and Matoušek. Let α be Otter's tree constant and $c := \frac{1}{\log_2(\alpha)}$. Then for any rational number $r > c$ the following assertion is true but independent of first order Peano arithmetic: For any natural number K there exists a natural number M so large that for any sequence $T_0, \ldots, T_M$ of finite trees satisfying that the number of nodes in T_i is bounded by $K + r \cdot \log_2(i+1)$ for $i = 0, \ldots, M$ we find indices i and j less than or equal to M such that $i < j$ and T_i is homeomorphically embeddable into T_j.

This result is rather sharp since for $r < c$ the corresponding assertion can be proved within primitive recursive arithmetic.

Friedman's miniaturization of the well-foundedness for ε_0 and iterated ω-powers ω_h is classified similarly with an application of the saddle point method. Using a result from random walk theory we give a classification of Friedman's miniaturization of KT for binary trees. Using related techniques we also obtain complete classification results for Hydra battles and Goodstein sequences. These investigations lead to new Goodstein style sequences for which termination is independent. We further characterize the derivation lengths for the finitary Ackermann functions using Brigham's theorem from analytic number theory (joint work with I. Lepper).