Termination Analysis for Functional Programs using Term Orderings*

Jürgen Giesl

FB Informatik, Technische Hochschule Darmstadt,
Alexanderstr. 10, 64283 Darmstadt, Germany
Email: giesl@inferenzsysteme.informatik.th-darmstadt.de

Abstract. To prove the termination of a *functional program* there has to be a well-founded ordering such that the arguments in each recursive call are *smaller* than the corresponding inputs. In this paper we present a procedure for *automated* termination proofs of *functional programs.* In contrast to previously presented methods a suited well-founded ordering does not have to be *fixed in advance by the user*, but can be synthesized automatically.

For that purpose we use approaches developed in the area of *term rewriting systems* for the automated generation of suited well-founded *term orderings.* But unfortunately term orderings cannot be directly used for termination proofs of *functional programs* which call other algorithms in the arguments of their recursive calls. The reason is that while for the termination of term rewriting systems orderings between *terms* are needed, for functional programs we need orderings between objects of *algebraic data types.* Our method solves this problem and enables *term orderings* to be used for termination proofs of *functional programs.*

1 Introduction

Termination of algorithms is a central problem in software development and formal methods for proving termination are essential for program verification. In this paper we develop a method for *automated* termination proofs of *functional programs.* Of course due to the undecidability of the halting problem no procedure can prove or disprove the termination of *all* algorithms.

Most work on the automation of termination proofs has been done in the areas of *term rewriting systems* and *logic programs* (for surveys on these topics see [Der87] and [SD93] resp.). Methods for termination proofs of *functional programs* have for instance been developed by R. S. Boyer and J S. Moore [BM79], C. Walther [Wal88], [Wal94] and F. and H. R. Nielson [NN95]. The procedure of Boyer and Moore has also been adapted for conditional rewrite systems [BL93].

But both the technique of Boyer and Moore and the methods for *logic programs* (e.g. [UV88], [Plü90], [SV91], [DSF93]) are only semi-automatic, i.e. for every termination proof at least the main characteristics of the suited well-founded ordering have to be *given in advance by the user.* The methods of Walther and of Nielson and Nielson are fully automated, but they are restricted to *one single fixed ordering* (resp. to lexicographic combinations of this ordering).

* Appeared in *Proceedings of the Second International Static Analysis Symposium*, Glasgow, Scotland, Springer–Verlag, LNCS 983, 1995.

To prove termination of *term rewriting systems* several methods for the *automated* synthesis of *term orderings* have been developed [Ste94]. For instance, there exist procedures for the automated generation of Knuth-Bendix orderings [Mar87], of polynomial orderings [Col75], [BCL87], [Ste94], [Gie95a] and of path orderings [Aït85], [DF85], [DH93].

Our aim is to use these synthesis methods for automated termination proofs of *functional programs*. Unfortunately, while term orderings can easily be used for termination proofs of *term rewriting systems*, they cannot be directly applied for termination proofs of *functional programs*, cf. Section 2.

After illustrating the disadvantages of straightforward solutions for this problem (Section 3) we develop a method which enables term orderings to be used for functional programs in Section 4. Then procedures for the automated generation of the right term ordering can also be applied for termination proofs of functional programs. The resulting method is more powerful than Walther's approach and has a higher degree of automation than the technique of Boyer and Moore. It has been implemented and integrated within the induction theorem proving system INKA [BHHW86]. We introduce some refinements of our method in Section 5 and end up with a conclusion and an outlook on future work.

2 Functional Programs and Term Orderings

In this paper we regard an eager functional language with algebraic data types. For the moment we restrict ourselves to non-parameterized types and to first order functions only. In Section 5 we will indicate how to use our method for polymorphic types and higher order functions.

2.1 Termination of Functional Programs

To prove the termination of a functional program we have to show that the arguments in each recursive call are *smaller* than the corresponding inputs. Every well-founded[1] relation can be used for this comparison.

As an example consider the algebraic data type tree for binary trees whose objects are built with the *constructors* nil and cons. The nullary function nil represents a leaf and $\mathsf{cons}(t_1, t_2)$ is the tree whose root has the direct subtrees t_1 and t_2. The following algorithm[2] transforms trees in a linear form such that all left subtrees are leafs, cf. Figures 1 and 2.

function flatten $(x : \mathsf{tree}) : \mathsf{tree} \Leftarrow$
if $x = \mathsf{nil}$ *then* nil
if $x = \mathsf{cons}(\mathsf{nil}, y)$ *then* $\mathsf{cons}(\mathsf{nil}, \mathsf{flatten}(y))$
if $x = \mathsf{cons}(\mathsf{cons}(u, v), w)$ *then* $\mathsf{flatten}(\mathsf{cons}(u, \mathsf{cons}(v, w)))$

[1] A relation $\succ$ is *well-founded* iff there exists no infinite descending chain $t_1 \succ t_2 \succ \ldots$

[2] To ease readability we have used a formulation of the algorithm without *selectors* (or *destructors*), i.e. our language uses *pattern matching* (where the patterns should be exhaustive).

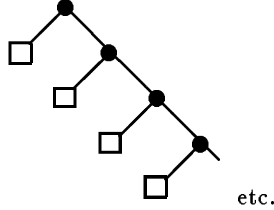

Fig. 1. Results computed by flatten.

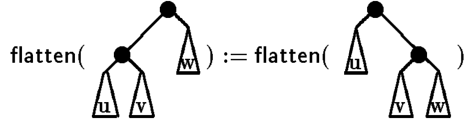

Fig. 2. Rotation of the tree.

To prove the termination of flatten we have to find a well-founded relation $\succ$ such that the inputs are always greater than the objects in the arguments of the recursive calls, i.e. the *termination hypotheses*

$$\mathsf{cons}(\mathsf{nil}, y) \succ y, \tag{1}$$

$$\mathsf{cons}(\mathsf{cons}(u, v), w) \succ \mathsf{cons}(u, \mathsf{cons}(v, w)) \tag{2}$$

must hold for all instantiations of y, u, v, w by terms built with nil and cons.

In other words, we have to synthesize a well-founded ordering on terms (a so-called *term ordering*) which satisfies the constraints (1) and (2). Techniques for the automated generation of well-founded term orderings satisfying such constraints have been developed in the area of term rewriting systems. For instance, termination of flatten can be proved with a *polynomial ordering* [Lan79], where every n-ary function symbol is associated with an n-ary polynomial over the natural numbers. Then a ground term t is *greater* than a ground term s with respect to the polynomial ordering iff the number corresponding to t is greater than the number corresponding to s. As all ground terms are associated with natural numbers, polynomial orderings are well-founded.

Let nil be associated with the nullary polynomial 0 and let $\mathsf{cons}(x, y)$ be associated with the polynomial $1 + 2x + y$. Then $\mathsf{cons}(\mathsf{nil}, y)$ corresponds to $1 + y$. As $1 + y > y$ holds for all instantiations of y with natural numbers, this polynomial ordering satisfies the first constraint (1). Analogously, $\mathsf{cons}(\mathsf{cons}(u, v), w)$ corresponds to $3 + 4u + 2v + w$ and $\mathsf{cons}(u, \mathsf{cons}(v, w))$ corresponds to $2 + 2u + 2v + w$. As $3 + 4u + 2v + w > 2 + 2u + 2v + w$ is true for all natural numbers u, v, w, this polynomial ordering also satisfies the second constraint (2). Therefore by using this polynomial ordering the termination of flatten is proved.

2.2 The Problem with Term Orderings

Unfortunately it is not always possible to use term orderings in this easy way for termination proofs of functional programs. Consider the following algorithm

function f $(x : \mathsf{tree}) : \mathsf{tree} \Leftarrow$
if $x = \mathsf{nil}$ *then* nil
if $x = \mathsf{cons}(u, v)$ *then* $\mathsf{f}(\mathsf{flatten}(\mathsf{cons}(u, v)))$.

Obviously the algorithm f is not terminating. For instance, evaluation of the expression f(cons(nil, nil)) leads to the recursive call f(flatten(cons(nil, nil))). As our functional language has eager semantics, the argument flatten(cons(nil, nil)) is evaluated (which yields cons(nil, nil)) and then f is called again with the argument cons(nil, nil).

Nevertheless there exists a well-founded term ordering $\succ$ such that the input term cons(u, v) is always greater than the term flatten(cons(u, v)) in the recursive call. For instance, let $t \succ s$ hold iff the leading function symbol of t is cons and the leading function symbol of s is flatten. But the function f does not terminate for inputs of the form cons(u, v). So the existence of a well-founded term ordering such that input terms are greater than the terms in the corresponding recursive calls is *not sufficient* for the termination of a functional program!

The reason for this problem is that in the recursive call of f another *algorithm* flatten is called. As flatten is *defined* by an algorithm (i.e. flatten is a so-called *defined function symbol*), terms built with the function symbol flatten are evaluated further. But the above term ordering does not respect the *semantics* of the algorithm flatten as flatten(cons(nil, nil)) and cons(nil, nil) are not equivalent with respect to this ordering. So the direct use of arbitrary well-founded term orderings for termination proofs of functional programs is *unsound*!

To prove the termination of a functional program, the inputs do not have to be compared with the *terms* in the recursive calls, but with the *data objects* which result from evaluating these terms. Hence, for termination proofs of functional programs instead of an ordering on *terms* we need an ordering on the *objects of the algebraic data type* (i.e. on terms like cons(nil, nil) which *consist only of constructors*). So for the termination of f, we would have to find a well-founded ordering on objects of the data type tree which satisfies the following *termination hypothesis* if flatten is *evaluated according to its algorithm.*

$$\mathsf{cons}(u, v) \succ \mathsf{flatten}(\mathsf{cons}(u, v))$$

We cannot directly use methods known from the area of term rewriting systems for the synthesis of such a term ordering, because the generated ordering would not respect the semantics of the defined function symbol flatten.

Termination of functional programs can only be directly proved with term orderings if the arguments of recursive calls contain no functions except *constructors* (as nil and cons), because terms built only with constructors cannot be evaluated further.

3 Two Straightforward Solutions

In this section we comment on two straightforward solutions which would enable the use of term orderings for termination proofs of functional programs. But these solutions lead to termination criteria whose requirements are *too strong*, i.e. with these solutions most termination proofs will fail.

3.1 Term Orderings Respecting the Semantics of Algorithms

A straightforward solution to enable the use of term orderings for functional programs which call other algorithms is the restriction to term orderings which respect the semantics of the called algorithms. Then *different terms* which denote the *same data object* (like $\mathsf{flatten}(\mathsf{cons}(\mathsf{nil},\mathsf{nil}))$ and $\mathsf{cons}(\mathsf{nil},\mathsf{nil})$) must be equivalent with respect to these orderings. This termination criterion is *sound*, i.e. with this restriction there obviously exists no well-founded term ordering such that the input term $\mathsf{cons}(u,v)$ of f is greater than the term $\mathsf{flatten}(\mathsf{cons}(u,v))$ in the recursive call.

But in general this restriction is too strong. Consider the following sorting algorithm on lists of natural numbers. The objects of the data structure list are built with the *constructors* empty and add, where $\mathsf{add}(n,x)$ represents the insertion of n into the list x. The algorithm sort calls two other algorithms min and rm, where $\mathsf{min}(x)$ computes the minimum of a list x and $\mathsf{rm}(n,x)$ removes *all* occurrences of n from the list x (i.e. the algorithm sort also eliminates duplicates in the list).

function sort $(x : \mathsf{list}) : \mathsf{list} \Leftarrow$
if $x = \mathsf{empty}$ *then* empty
if $x \neq \mathsf{empty}$ *then* $\mathsf{add}(\mathsf{min}(x), \mathsf{sort}(\mathsf{rm}(\mathsf{min}(x), x)))$

Our aim is to use term orderings known from term rewriting systems for termination proofs of algorithms. But none of these term orderings both respects the semantics of min and rm *and* makes inputs greater than the corresponding recursive calls of sort:

Termination of sort cannot be proved with a *polynomial ordering* which respects the semantics of min and rm[3]. Virtually all other term orderings $\succ$ used in the area of term rewriting systems are *simplification orderings* [Der79], [Ste94]. As these orderings possess the *subterm property* (i.e. $f(\ldots t \ldots) \succ t$), simplification orderings do not respect the semantics of the algorithm min. The reason is that $\mathsf{min}(\mathsf{add}(0,\mathsf{empty}))$ is evaluated to its *subterm* 0 and therefore these terms cannot be equivalent with respect to a simplification ordering.

3.2 Functional Programs as (Conditional) Term Rewriting Systems

Another straightforward solution for the termination proof of the algorithm sort is to transform sort and the auxiliary algorithms min and rm into a *term rewriting system* and to prove its termination instead.

Functional programs in our language can be regarded as a special type of (conditional) term rewriting systems with an *innermost evaluation* strategy. But due to their special form and due to this evaluation strategy it is possible to use a different approach for termination proofs of functional programs than it is necessary for term rewriting systems. For instance, for functional programs it

[3] A proof for this observation can be found in the appendix.

is sufficient to compare the input *arguments* with the *arguments* in the corresponding recursive calls, while for term rewriting systems left and right hand sides of *all rules* have to be compared (and moreover, the term ordering has to be *monotonic*), cf. [Der87].

Therefore by the transformation of functional programs into a term rewriting system we impose unnecessarily strong requirements for the termination proof. For instance, with most of the commonly used[4] term orderings the termination of the term rewriting system resulting from sort, min and rm cannot be proved. The reason is that these orderings are monotonic and possess at least the non-strict subterm property (i.e. $f(\ldots t \ldots) \succeq t$). But as the obtained term rewriting system contains the rule

$$\mathsf{sort}(\mathsf{add}(n,y)) \;\rightarrow\; \mathsf{add}(\ldots,\mathsf{sort}(\mathsf{rm}(\ldots,\mathsf{add}(n,y)))),$$

it cannot be oriented by any of these term orderings.

We have illustrated that the direct use of *arbitrary* term orderings for termination proofs of algorithms is *unsound* and that the straightforward solutions (i.e. the restriction to term orderings which respect the semantics of the algorithms or the transformation of functional programs into a term rewriting system) impose too strong requirements such that termination proofs *often fail.* In the next section we present a different, powerful method to enable the application of term orderings for termination proofs of functional programs.

4 Elimination of Defined Function Symbols

To prove the termination of sort we have to show that there exists a well-founded ordering $\succ$ on lists which satisfies the following *termination hypothesis* if the defined functions rm and min are *evaluated according to their algorithms.*

$$x \neq \mathsf{empty} \;\rightarrow\; x \succ \mathsf{rm}(\mathsf{min}(x),x) \tag{3}$$

As demonstrated in Section 2 the application of methods for the synthesis of well-founded term orderings is only possible if the termination hypotheses do not contain defined function symbols (like rm and min). Therefore our aim is to transform the termination hypothesis (3) into formulas *without defined function symbols.*

The structure of our termination proof method is illustrated in Figure 3. We have developed a calculus which eliminates defined function symbols. In this way the termination hypotheses TH_0 of an algorithm are transformed into TH_1, TH_2 etc. until we obtain a set of formulas TH_n containing no defined function symbols any more. This transformation is an *abduction* process [Pei31], i.e. $\mathrm{TH}_{i+1} \models \mathrm{TH}_i$

[4] In this paper we only refer to those term orderings that are amenable to automation. There also exist classes of term orderings which can orient *every* terminating term rewriting system (e.g. semantical path orderings [KL80] or transformation orderings [BL90]). But the disadvantage of these powerful approaches is that up to now there are only very few suggestions for their automated generation [Ste95].

$$
\begin{array}{ccccccccc}
\mathrm{TH}_0 & \rightarrow & \mathrm{TH}_1 & \rightarrow & \mathrm{TH}_2 & \rightarrow & \ldots & \rightarrow & \mathrm{TH}_n \\
\Uparrow & & \Uparrow & & \Uparrow & & & & \Uparrow \\
\succ_0 & \leftarrow & \succ_1 & \leftarrow & \succ_2 & \leftarrow & \ldots & \leftarrow & \succ_n
\end{array}
$$

Fig. 3. Elimination of defined function symbols from termination hypotheses.

holds for all i. Therefore if a relation $\succ_{i+1}$ satisfies the constraints TH_{i+1}, then it also satisfies the constraints TH_i (where defined function symbols must be evaluated according to their algorithmic definitions).

The formulas TH_n resulting from the transformation process contain no defined function symbols. Therefore we can now directly apply methods from the area of term rewriting systems to generate a well-founded term ordering $\succ_n$ satisfying the constraints TH_n. As TH_n implies TH_0, this ordering $\succ_n$ also satisfies the termination hypotheses TH_0 of the algorithm. Therefore the existence of a term ordering satisfying the resulting constraints TH_n is sufficient for the termination of the algorithm.

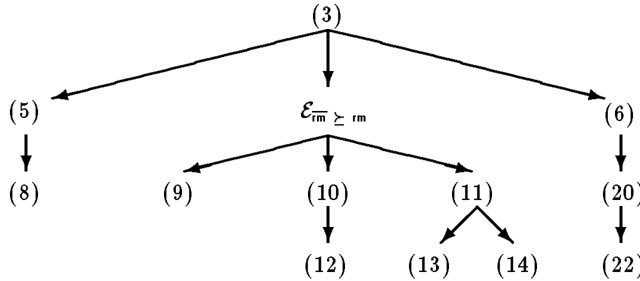

Fig. 4. Termination proof of sort.

The derivation tree in Figure 4 illustrates the transformation of the termination hypothesis (3) of sort. Every node in the tree is transformed into its successors by an application of one transformation rule. Leafs of the tree are formulas that do not contain defined function symbols (and therefore no transformation rule is applicable to them). As each transformation rule is an abduction step, the formulas at the leafs imply the termination hypothesis at the root of the tree. Therefore the existence of a well-founded term ordering satisfying the constraints at the leafs of the tree is sufficient for the termination of sort. In the following our termination proof method is presented in three steps.

In Section 4.1 we define our two main inference rules, viz. the *estimation* and the *generalization* rule. To apply the estimation rule we need so-called *estimation inequalities* and *strictness predicates* and Section 4.2 illustrates how they can be computed automatically. While all other steps of our transformation process are purely *syntactical*, in the last step we consider the *semantics* to eliminate the introduced strictness predicates again, cf. Section 4.3.

4.1 Estimation and Generalization

The termination hypothesis (3) of sort contains two defined function symbols rm and min. The central idea of our termination procedure is the *estimation* of defined function symbols by new *undefined* function symbols. Therefore rm is replaced by a new undefined function symbol $\overline{\mathsf{rm}}$ and we demand that the result of $\overline{\mathsf{rm}}$ is always greater or equal than the result of rm, i.e. we demand

$$\overline{\mathsf{rm}}(n, x) \succeq \mathsf{rm}(n, x). \tag{4}$$

In contrast to rm the undefined function symbol $\overline{\mathsf{rm}}$ has no fixed semantics. In the following we transform the termination hypothesis (3) into inequalities which contain undefined function symbols like $\overline{\mathsf{rm}}$, but no defined function symbols like rm. If these resulting inequalities are satisfied by a well-founded term ordering, then the termination of sort is proved.

Assume for the moment that we know a set of so-called *estimation inequalities* $\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}}$ (without defined function symbols) which imply (4). Then demanding

$$x \succeq \overline{\mathsf{rm}}(\mathsf{min}(x), x) \tag{5}$$

and $\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}}$ is sufficient for the *non-strict*[5] version of the termination hypothesis, i.e.

$$x \succeq \overline{\mathsf{rm}}(\mathsf{min}(x), x) \succeq \mathsf{rm}(\mathsf{min}(x), x).$$

To ensure the original *strict* inequality we have to demand that under the condition $x \neq \mathsf{empty}$ inequality (5) has to be strict or $\overline{\mathsf{rm}}(\mathsf{min}(x), x) \succ \mathsf{rm}(\mathsf{min}(x), x)$ has to be true.

Assume also that we know a so-called *strictness predicate* $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}$ (which is defined by an algorithm) that returns true iff the result of $\overline{\mathsf{rm}}$ is *strictly* greater than the result of rm. Then $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}(\mathsf{min}(x), x)$ indicates whether the estimation of $\mathsf{rm}(\mathsf{min}(x), x)$ by $\overline{\mathsf{rm}}(\mathsf{min}(x), x)$ is strict. Therefore we can replace the termination hypothesis (3) of sort by (5), $\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}}$ and the formula

$$x \neq \mathsf{empty} \;\rightarrow\; x \succ \overline{\mathsf{rm}}(\mathsf{min}(x), x) \vee \delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}(\mathsf{min}(x), x). \tag{6}$$

The transformation of (3) into (5), $\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}}$ and (6) corresponds to the step from the root of the derivation tree in Figure 4 to its direct successors. It is an *abduction* step, because (5), $\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}}$ and (6) are *sufficient* for the termination hypothesis (3). For this transformation we have used the following *estimation rule* to *estimate* a *defined* function symbol g:

[5] In this paper we use the word "strict" to distinguish inequalities built with $\succ$ from those built with $\succeq$. This should not be confused with the use of the word "strict" in semantics.

$$\frac{\varphi \rightarrow t \succ g(q_1, \ldots, q_n)}{\begin{array}{c} t \succeq \bar{g}(q_1, \ldots, q_n) \\ \mathcal{E}_{\bar{g} \succeq g} \\ \varphi \rightarrow t \succ \bar{g}(q_1, \ldots, q_n) \vee \delta_{\bar{g} \succ g}(q_1, \ldots, q_n) \end{array}} \quad (7)$$

This rule[6] embodies a main principle of our transformation: For a termination hypothesis $\varphi \rightarrow t \succ s$ we ensure that the *non-strict unconditional* inequality $t \succeq s$ holds. The advantage of omitting the condition φ (which contains *semantical* information) is that defined function symbols in $t \succeq s$ can be eliminated by repeated application of *purely syntactical* inference rules (like the *estimation* rule). A *semantical* inference rule which considers the condition φ and the semantics of strictness predicates is only necessary to guarantee that one of the estimation steps in the derivation is strict (see Section 4.3 for the handling of formulas like (6)).

The resulting formulas (5) and (6) still contain *defined* function symbols. To eliminate the defined function symbol min from formula (5) we could again use the technique of *estimation* and replace min by a new undefined function symbol $\overline{\mathsf{min}}$. But apart from estimation there exists another (obvious) method for the elimination of the defined function symbol min, viz. replacing the term $\mathsf{min}(x)$ by a new variable y. In this way, (5) is transformed into

$$x \;\succeq\; \overline{\mathsf{rm}}(y, x). \quad (8)$$

This *generalization* is also an abduction step, because if (8) holds for all y, then (5) must hold as well. As (8) contains no defined function symbols, this inequality corresponds to a leaf in the derivation tree. Hence, we have introduced two rules for the elimination of defined function symbols, viz. *estimation* and *generalization.*

4.2 Estimation Inequalities and Strictness Predicates

In this section we show how to compute *estimation inequalities* and *strictness predicates* which are needed for the estimation technique of Section 4.1. The estimation inequalities $\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}}$ have to guarantee that $\overline{\mathsf{rm}}$ is really an upper bound for rm and the strictness predicate $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}$ has to indicate whether this estimation is strict.

4.2.1 Estimation Inequalities

The function rm is defined by the following algorithm which removes all occurrences of a number from a list of natural numbers.

[6] We also need an estimation rule for *non-strict* inequalities $t \succeq g(\ldots)$ which only contains the first two consequences of (7).

function $\mathsf{rm}\ (n : \mathsf{nat},\ x : \mathsf{list}) : \mathsf{list} \Leftarrow$
  if $x = \mathsf{empty}$ *then* empty
  if $x = \mathsf{add}(n, y)$ *then* $\mathsf{rm}(n, y)$
  if $x = \mathsf{add}(m, y) \wedge m \neq n$ *then* $\mathsf{add}(m, \mathsf{rm}(n, y))$

To compute $\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}}$ we consider each case of rm separately. Instead of $\overline{\mathsf{rm}}(n, x) \succeq \mathsf{rm}(n, x)$ we therefore demand (again omitting *conditions* (like $m \neq n$) in *non-strict* inequalities)

$$\overline{\mathsf{rm}}(n, \mathsf{empty}) \succeq \mathsf{empty}, \tag{9}$$

$$\overline{\mathsf{rm}}(n, \mathsf{add}(n, y)) \succeq \mathsf{rm}(n, y), \tag{10}$$

$$\overline{\mathsf{rm}}(n, \mathsf{add}(m, y)) \succeq \mathsf{add}(m, \mathsf{rm}(n, y)). \tag{11}$$

We cannot define $\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}} = \{(9), (10), (11)\}$ as the inequalities (10) and (11) still contain the defined function symbol rm. Defined function symbols occurring in such formulas have to be eliminated by *estimation* or *generalization* again.

But the problem here is that rm *itself* appears in the inequalities (10) and (11). Therefore we cannot use the (non-strict version of the) estimation rule (7) for the estimation of rm, because we do not know the estimation inequalities $\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}}$ yet.

We solve this problem by constructing $\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}}$ *inductively* with respect to the *computation ordering*[7] of rm. The base case of this inductive construction corresponds to the non-recursive case of rm. Inequality (9) ensures that in the base case $\overline{\mathsf{rm}}$ is an upper bound for rm.

In the second case of rm we have to ensure that inequality (10) holds, i.e. for inputs of the form $(n, \mathsf{add}(n, y))$ the result of $\overline{\mathsf{rm}}$ must be greater or equal than the result of rm. As *induction hypothesis* we can now use that this estimation is already correct for the arguments (n, y), i.e. $\overline{\mathsf{rm}}(n, y) \succeq \mathsf{rm}(n, y)$ holds. Then it is sufficient for (10) if

$$\overline{\mathsf{rm}}(n, \mathsf{add}(n, y)) \succeq \overline{\mathsf{rm}}(n, y) \tag{12}$$

is true. Therefore we can replace (10) by inequality (12) which does not contain defined function symbols.

So to eliminate the defined function symbol rm from (10) we use the (non-strict version of the) estimation rule (7) and due to an inductive argument we can omit the second consequence ($\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}}$) of this inference rule.

In the third case of rm we proceed in an analogous way, because for the transformation of (11) we can again use the induction hypothesis $\overline{\mathsf{rm}}(n, y) \succeq \mathsf{rm}(n, y)$. In this way we obtain the inequality

$$\overline{\mathsf{rm}}(n, \mathsf{add}(m, y)) \succeq \mathsf{add}(m, \overline{\mathsf{rm}}(n, y)). \tag{13}$$

[7] The *computation ordering* $>_g$ of an n-ary algorithm g is defined as $(t_1, \ldots, t_n) >_g (s_1, \ldots, s_n)$ iff the evaluation of the expression $g(t_1, \ldots, t_n)$ leads to the recursive call $g(s_1, \ldots, s_n)$.

To imply (11), the right hand side of (13) would have to be greater or equal than the right hand side of (11). But (13) and the induction hypothesis are not sufficient for (11), because the induction hypothesis $\overline{\mathsf{rm}}(n,y) \succeq \mathsf{rm}(n,y)$ does not imply $\mathsf{add}(m,\overline{\mathsf{rm}}(\ldots)) \succeq \mathsf{add}(m,\mathsf{rm}(\ldots))$! Therefore additionally we have to demand that the *result* of add should also be decreasing if the second *argument* of add is decreasing. In other words, add should be *monotonic* in its second argument, i.e. in addition to (13) we demand the constraint

$$u \succ v \;\rightarrow\; \mathsf{add}(m,u) \succ \mathsf{add}(m,v). \tag{14}$$

This problem always appears when estimating a defined function symbol which is not the *leading* function symbol. For the estimation of a function symbol which appears within a term at a position p we therefore have to extend the estimation rule (7) by a consequence which demands that the term is *monotonic* in the position p. Subsequently any defined function symbols in monotonicity formulas like (14) must be eliminated by *generalization*.

Now we have finished our inductive construction of $\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}}$ (as illustrated in Figure 4) and obtain

$$\begin{aligned}
\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}} = \{&\overline{\mathsf{rm}}(n,\mathsf{empty}) \succeq \mathsf{empty}, && (9)\\
&\overline{\mathsf{rm}}(n,\mathsf{add}(n,y)) \succeq \overline{\mathsf{rm}}(n,y), && (12)\\
&\overline{\mathsf{rm}}(n,\mathsf{add}(m,y)) \succeq \mathsf{add}(m,\overline{\mathsf{rm}}(n,y)), && (13)\\
&u \succ v \;\rightarrow\; \mathsf{add}(m,u) \succ \mathsf{add}(m,v)\}. && (14)
\end{aligned}$$

So in general *estimation inequalities* $\mathcal{E}_{\bar{g} \succeq g}$ are computed in the following way:

1. For each case "*if* φ *then* r" of the algorithm g we construct the formula $\bar{g}(\ldots) \succeq r$.
2. Then the defined function symbols in $\bar{g}(\ldots) \succeq r$ are eliminated by estimation or generalization. When g *itself* is estimated we can omit the second consequence ($\mathcal{E}_{\bar{g} \succeq g}$) of the (non-strict version of the) estimation rule (7).

The construction of $\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}}$ by *induction* with respect to the computation ordering of rm is only sound, if this computation ordering is well-founded (i.e. if rm is terminating). So before proving the termination of sort we must have proved the termination of rm. Therefore we always demand that (apart from recursive calls) algorithms only call other algorithms whose termination has been verified *before*, i.e. we exclude mutually recursive algorithms.

4.2.2 Strictness Predicates

The algorithm for the *strictness predicate* $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}$ is also constructed by induction with respect to the computation ordering of rm. The predicate $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}(n,x)$ has to return true iff $\overline{\mathsf{rm}}(n,x)$ is strictly greater than $\mathsf{rm}(n,x)$. By an analysis according to the cases of the algorithm rm one obtains the following cases for the algorithm $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}$:

$$if\ x = \mathsf{empty}\ then\ \overline{\mathsf{rm}}(n, \mathsf{empty}) \succ \mathsf{empty}, \tag{15}$$

$$if\ x = \mathsf{add}(n, y)\ then\ \overline{\mathsf{rm}}(n, \mathsf{add}(n, y)) \succ \mathsf{rm}(n, y), \tag{16}$$

$$if\ x = \mathsf{add}(m, y) \wedge m \neq n\ then\ \overline{\mathsf{rm}}(n, \mathsf{add}(m, y)) \succ \mathsf{add}(m, \mathsf{rm}(n, y)). \tag{17}$$

But the inequalities (16) and (17) still contain the defined function symbol rm. Therefore we eliminate this defined function symbol by estimation again. So inequality (16) from the second case is transformed into (12), $\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}}$ and

$$if\ x = \mathsf{add}(n, y)\ then\ \overline{\mathsf{rm}}(n, \mathsf{add}(n, y)) \succ \overline{\mathsf{rm}}(n, y) \vee \delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}(n, y). \tag{18}$$

We construct $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}$ *inductively*, i.e. when defining $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}(n, \mathsf{add}(n, y))$ we use the induction hypothesis that $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}(n, y)$ is already correctly defined. This results in the *recursive call* $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}(n, y)$.

Note that (12) is already included in $\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}}$. Therefore as long as $\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}}$ holds we only have to consider the third consequence of the estimation rule and replace (16) by (18). We proceed in the same way for the third case (17) and obtain the following strictness predicate algorithm.

predicate $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}\ (n : \mathsf{nat},\ x : \mathsf{list}) \Leftarrow$
$if\ x = \mathsf{empty}\ then\ \overline{\mathsf{rm}}(n, \mathsf{empty}) \succ \mathsf{empty}$
$if\ x = \mathsf{add}(n, y)\ then\ \overline{\mathsf{rm}}(n, \mathsf{add}(n, y)) \succ \overline{\mathsf{rm}}(n, y) \vee \delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}(n, y)$
$if\ x = \mathsf{add}(m, y) \wedge m \neq n\ then\ \overline{\mathsf{rm}}(n, \mathsf{add}(m, y)) \succ \mathsf{add}(m, \overline{\mathsf{rm}}(n, y)) \vee \delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}(n, y)$

In general *strictness predicate algorithms* $\delta_{\bar{g} \succ g}$ are constructed as follows:

1. For each case "*if* φ *then* r" of the algorithm g we construct a case "*if* φ *then* $\bar{g}(\ldots) \succ r$".
2. Then the defined function symbols in $\bar{g}(\ldots) \succ r$ are eliminated by estimation or generalization. When using estimation we can omit all consequences of the estimation rule (7) except the third one.

This construction of strictness predicate algorithms is *sound*, i.e. if the estimation inequalities $\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}}$ hold, then for each number n and each list x the strictness predicate $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}(n, x)$ defined by the above algorithm returns true iff $\overline{\mathsf{rm}}(n, x) \succ \mathsf{rm}(n, x)$ holds.

4.3 Elimination of Strictness Predicates

By estimation and generalization of defined function symbols we have transformed the termination hypothesis (3) of sort into inequality (8), the estimation inequalities $\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}}$ and the formula

$$x \neq \mathsf{empty} \ \rightarrow\ x \succ \overline{\mathsf{rm}}(\mathsf{min}(x), x) \vee \delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}(\mathsf{min}(x), x), \tag{6}$$

cf. Figure 4. While (8) and $\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}}$ contain no defined function symbols, formula (6) contains the strictness predicate $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}$ which is *defined* by an algorithm. In order to complete the transformation of the termination hypothesis (3) into inequalities without defined function symbols we now have to eliminate the strictness predicate $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}$ from (6).

For that purpose we omit one part of the disjunction, i.e. instead of (6) we demand one of the following two constraints:

$$x \neq \mathsf{empty} \rightarrow x \succ \overline{\mathsf{rm}}(\mathsf{min}(x), x) \quad \text{or} \tag{19}$$

$$x \neq \mathsf{empty} \rightarrow \delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}(\mathsf{min}(x), x). \tag{20}$$

Formula (19) does not contain a strictness predicate. So the defined function symbol min can be eliminated by estimation or generalization. But if we transform (6) into (19) the termination proof of sort will fail. The reason is that there exists no well-founded ordering satisfying the constraints $\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}}$ and (19)[8].

So instead of omitting the second part of the disjunction in (6) we should rather omit the first part and replace (6) by (20). As (20) contains the *defined* strictness predicate $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}$ we now have to transform (20) into inequalities without $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}$ that are sufficient for (20). For that purpose we choose some of the *inequalities occurring in the algorithm* $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}$, i.e. inequalities from

$$\overline{\mathsf{rm}}(n, \mathsf{empty}) \succ \mathsf{empty}, \tag{21}$$

$$\overline{\mathsf{rm}}(n, \mathsf{add}(n, y)) \succ \overline{\mathsf{rm}}(n, y), \tag{22}$$

$$\overline{\mathsf{rm}}(n, \mathsf{add}(m, y)) \succ \mathsf{add}(m, \overline{\mathsf{rm}}(n, y)). \tag{23}$$

To minimize the number of resulting constraints we should select a *minimal* subset of these inequalities which is sufficient for (20). For instance, (20) is implied by inequality (22) from the second case of $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}$.

Essentially, the reason is that every non-empty list x contains its minimum. Therefore when evaluating $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}(\mathsf{min}(x), x)$, after a finite number of recursive calls $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}$ will be called with a list which begins with its minimum. Then the condition of the second case will be satisfied and therefore $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}$ returns true if (22) is true.

Different to the syntactical inference rules presented in the preceding sections we now had to consider the *semantics* of the strictness predicate $\delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}$. To eliminate strictness predicates we have to *prove* that certain inequalities (like (22)) are sufficient for formulas like $x \neq \mathsf{empty} \rightarrow \delta_{\overline{\mathsf{rm}} \succ \mathsf{rm}}(\ldots)$. To perform such proofs automatically we make use of an *induction theorem proving system* (e.g. those described in [BM79], [BHHW86], [BHHS90]).

So to *eliminate the defined strictness predicate* $\delta_{\bar{g} \succ g}$ from a formula of the form $\varphi \rightarrow t \succ s \vee \delta_{\bar{g} \succ g}(\ldots)$ we proceed as follows:

---

[8] A proof for this observation can be found in the appendix.

1. Either we replace the formula by $\varphi \rightarrow t \succ s$
2. or we replace it by a minimal set of inequalities from the algorithm $\delta_{\bar{g} \succ g}$ that is sufficient for $\varphi \rightarrow \delta_{\bar{g} \succ g}(\ldots)$.

By replacing (6) with (22) we have finished the transformation of sort's termination hypothesis into inequalities without defined function symbols, i.e. we have constructed the derivation tree in Figure 4. To prove sort's termination we now have to find a well-founded term ordering that satisfies the constraints (8), (9), (12), (13), (14), (22) at the leafs of the tree.

For instance, the above constraints are satisfied by a *polynomial ordering* where empty is associated with 0, $\mathsf{add}(n, x)$ is associated with the polynomial $x+1$ and $\overline{\mathsf{rm}}(n, x)$ is associated with the polynomial x. Therefore the termination of sort is proved.

For the synthesis of such well-founded term orderings we apply procedures which are used in the area of term rewriting systems, cf. Section 1. For instance, the algorithm of *G. E. Collins* [Col75] can decide whether there exists a real polynomial ordering of a given degree which satisfies a set of constraints. A procedure to generate polynomial orderings using an efficient, incomplete modification of Collins' algorithm has been presented in [Gie95a].

5 Comments and Refinements

To enable the use of term orderings for termination proofs of functional programs *defined* function symbols in the recursive calls have to be eliminated. This elimination proceeds in three steps. First, defined function symbols g in the termination hypotheses are *generalized* or *estimated* by new undefined function symbols $\bar{g}$ (Section 4.1). To guarantee that $\bar{g}$ is an upper bound for g we have to demand *estimation inequalities* $\mathcal{E}_{\bar{g} \succeq g}$. By the estimation of a function symbol g we also obtain a formula containing the *strictness predicate* $\delta_{\bar{g} \succ g}$. This predicate indicates whether the result of $\bar{g}$ is strictly greater than the result of g (Section 4.2). Finally the (defined) strictness predicate has to be eliminated (using an induction theorem proving system) (Section 4.3).

Of course the more powerful the used induction theorem proving system is, the more algorithms can be proved terminating by our method. However we tested our method on more than 100 examples and noticed that for about 90% of the algorithms the required proofs can already be accomplished by case analysis and propositional reasoning only (i.e. no *induction* proofs are needed for them).

The transformation of termination hypotheses into formulas without defined function symbols *terminates* as the number of defined function symbols is decreasing. Therefore derivation trees only contain paths of *finite* length. But our termination procedure contains two choice points: First, defined function symbols can be eliminated by *estimation* or by *generalization*. Second, for disjunctions like (6) one has to decide which part of the disjunction to omit. Moreover,

there may be more than one minimal sufficient set of inequalities for the elimination of strictness predicates. So there can be several different derivation trees for one termination hypothesis. But as the number of derivation trees for one termination hypothesis is also *finite* (and *small*), we can backtrack if no well-founded term ordering satisfying the constraints at the leafs can be found. To improve the efficiency of our method, we have also developed heuristics for choosing the "right" derivation tree [Gie95b]. These heuristics have proved successful in practice. An alternative method for termination proofs with *user provided* orderings which avoids such choice points is presented in [Gie95c].

Our method can easily be extended to algorithms with *several* formal parameters. For that purpose we introduce a new undefined function symbol ν and instead of two *tuples* $(t_1, \ldots, t_n)$ and $(s_1, \ldots, s_n)$ we compare the *terms* $\nu(t_1, \ldots, t_n)$ and $\nu(s_1, \ldots, s_n)$. So for instance for the termination proof of the algorithm rm we have to demand $\nu(n, \mathsf{add}(n, y)) \succ \nu(n, y)$ and $m \neq n \rightarrow \nu(n, \mathsf{add}(m, y)) \succ \nu(n, y)$.

The presented technique also works for *polymorphic* types, as type constants (like list) may of course also be parameterized with type variables (e.g. $\alpha\mathsf{list}$ with the constructors $\mathsf{empty} : \alpha\mathsf{list}$ and $\mathsf{add} : \alpha \times \alpha\mathsf{list} \rightarrow \alpha\mathsf{list}$).

For termination analysis of *higher order* functional programs we suggest to integrate our method into the type inference system presented by Nielson and Nielson [NN95]. In their system functional types have an *annotation* to distinguish *total* functions from probably partial ones. To prove termination of a recursive function, they use a rule which infers the annotated type of the function under the assumption that the recursive calls of the function are terminating. For the application of this rule one has to verify that recursive calls are only applied to *smaller* arguments.

For that purpose Nielson and Nielson use (lexicographic combinations of) a *fixed* ordering that *cannot deal with functions which call other algorithms in their recursive calls.* But instead one could use our technique and thereby obtain a powerful, fully automated procedure for termination analysis of *higher order* functional programs.

6 Conclusion and Further Work

We have presented a method for automated termination proofs of functional programs which uses approaches known from the area of term rewriting systems to generate term orderings automatically. As demonstrated in Section 2 and 3 due to the *defined* function symbols in recursive calls a direct use of term orderings for termination proofs of *functional programs* is not possible. Therefore in Section 4 and 5 we have developed a method to eliminate defined function symbols from the termination hypotheses of an algorithm.

Our method has been implemented within the induction theorem proving system INKA [BHHW86] (using a procedure for the automated generation of polynomial orderings [Gie95a]) and proved successful on several examples. For instance, it can *fully automatically* prove the termination of all 60 algorithms

from the database of [Wal88], [Wal94] and of all 82 algorithms from [BM79] (where one algorithm (greatest.factor) must be slightly modified). In all other methods for termination proofs of functional programs (e.g. [BM79], [Wal94], [NN95]) the orderings for proving termination are either *fixed* or *have to be provided in advance by the user* while in our method the right ordering can be synthesized automatically.

For the computation of estimation inequalities we had to exclude *mutually recursive* algorithms, but we plan to extend our method to mutual recursion in the future. Moreover, we intend to examine whether our method can also be used for an analysis of partial functions resp. functions which do not *always* terminate. Further work will also include termination proofs for functional languages with *lazy* semantics.

A Proofs

Observation 1 *Termination of* sort *cannot be proved with a polynomial ordering which respects the semantics of* min *and* rm*.*

Proof:
Evaluation of $\mathsf{min}(\mathsf{add}(0,x))$ yields the result 0 *for all* lists x. Therefore if the polynomial ordering respects the semantics of min, then $\mathsf{min}(\mathsf{add}(0,x))$ must correspond to the same number *for all* x.

If min is associated with a *non-constant* polynomial, then it can only map a *finite* number of different arguments to the same value. So there exists only a *finite* number of lists (say k) that correspond to different numbers. But a call of sort can lead to more than k subsequent recursive calls. Therefore the number corresponding to the list in the recursive call cannot always be smaller than the number corresponding to the input list.

If min is associated with a *constant* and if the polynomial ordering respects the semantics of rm, then it can be shown that *all* lists must be associated with the same number so that input and recursive call of sort are equivalent with respect to this ordering. □

Observation 2 *There exists no well-founded ordering satisfying the constraints* $\mathcal{E}_{\overline{\mathsf{rm}} \succeq \mathsf{rm}}$ *and (19).*

Proof:
For each number n the following inequalities hold:

$$\begin{aligned} \mathsf{add}(n,\mathsf{empty}) &\succ \overline{\mathsf{rm}}(\mathsf{min}(\mathsf{add}(n,\mathsf{empty})),\mathsf{add}(n,\mathsf{empty})), \quad \text{because of (19)} \\ &\succeq \mathsf{add}(n,\overline{\mathsf{rm}}(\mathsf{min}(\mathsf{add}(n,\mathsf{empty})),\mathsf{empty})), \quad \text{because of (13)} \\ &\succeq \mathsf{add}(n,\mathsf{empty}), \quad \text{because of (9) and (14).} \end{aligned}$$

□

Acknowledgements
I would like to thank Jürgen Brauburger, Stefan Gerberding, Thomas Kolbe, Martin Protzen, Christoph Walther and the referees for helpful suggestions.

References

[Aït85] H. Aït-Kaci. An Algorithm for Finding a Minimal Recursive Path Ordering. *RAIRO*, 19(4):359-382, 1985.

[BL90] F. Bellegarde & P. Lescanne. Termination by Completion. *Applicable Algebra in Engineering, Communication and Computing*, 1:79-96, 1990.

[BCL87] A. Ben Cherifa & P. Lescanne. Termination of Rewriting Systems by Polynomial Interpretations and its Implementation. *Science of Computer Programming*, 9(2):137-159, 1987.

[BL93] E. Bevers & J. Lewi. Proving Termination of (Conditional) Rewrite Systems. *Acta Informatica*, 30:537-568, 1993.

[BHHW86] S. Biundo, B. Hummel, D. Hutter & C. Walther. The Karlsruhe Induction Theorem Proving System. In *Proceedings of the 8th International Conference on Automated Deduction*, Oxford, England, 1986.

[BM79] R. S. Boyer & J S. Moore. *A Computational Logic.* Academic Press, 1979.

[BHHS90] A. Bundy, F. van Harmelen, C. Horn & A. Smaill. The OYSTER-CLAM System. In *Proceedings of the 10th International Conference on Automated Deduction*, Kaiserslautern, Germany, 1990.

[Col75] G. E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. In *Proceedings of the Second GI Conference on Automata Theory and Formal Languages*, Kaiserslautern, Germany, 1975.

[DSF93] S. Decorte, D. De Schreye & M. Fabris. Automatic Inference of Norms: A Missing Link in Automatic Termination Analysis. In *Proceedings of the International Logic Programming Symposium*, Vancouver, Canada, 1993.

[DF85] D. Detlefs & R. Forgaard. A Procedure for Automatically Proving the Termination of a Set of Rewrite Rules. In *Proceedings of the First International Conference on Rewriting Techniques and Applications*, Dijon, France, 1985.

[Der79] N. Dershowitz. A Note on Simplification Orderings. *Information Processing Letters*, 9(5):212-215, 1979.

[Der87] N. Dershowitz. Termination of Rewriting. *Journal of Symbolic Computation*, 3(1, 2):69-115, 1987.

[DH93] N. Dershowitz & C. Hoot. Topics in Termination. In *Proceedings of the 5th International Conference on Rewriting Techniques and Applications*, Montreal, Canada, 1993.

[Gie95a] J. Giesl. Generating Polynomial Orderings for Termination Proofs. In *Proceedings of the 6th International Conference on Rewriting Techniques and Applications*, Kaiserslautern, Germany, 1995.

[Gie95b] J. Giesl. *Automatisierung von Terminierungsbeweisen für rekursiv definierte Algorithmen.* Doctoral Dissertation, Technische Hochschule Darmstadt, Germany, 1995.

[Gie95c] J. Giesl. Automated Termination Proofs with Measure Functions. In *Proceedings of the 19th Annual German Conference on Artificial Intelligence*, Bielefeld, Germany, 1995.

[KL80] S. Kamin & J.-J. Levy. Two Generalizations of the Recursive Path Ordering. Unpublished Note, Department of Computer Science, University of Illinois, Urbana, IL, 1980.

[Lan79] D. S. Lankford. On Proving Term Rewriting Systems are Noetherian. Technical Report Memo MTP-3, Mathematics Department, Louisiana Technical University, 1979.

[Mar87] U. Martin. How to choose Weights in the Knuth-Bendix Ordering. In *Proceedings of the Second International Conference on Rewriting Techniques and Applications*, Bordeaux, France, 1987.

[NN95] F. Nielson & H. R. Nielson. Termination Analysis based on Operational Semantics. Technical Report, Aarhus University, Denmark, 1995. Available from `http://www.daimi.aau.dk/~fn/Papers/PB492.ps.Z`.

[Pei31] C. S. Peirce. *Collected Papers of C. Sanders Peirce*, vol. 2. Hartshorne et al. (eds.), Harvard University Press, Cambridge, MA, 1931.

[Plü90] L. Plümer. *Termination Proofs for Logic Programs.* Springer–Verlag, 1990.

[SD93] D. De Schreye & S. Decorte. Termination of Logic Programs: The Never-Ending Story. Technical Report Compulog II, D 8.1.1, K. U. Leuven, Belgium, 1993.

[SV91] K. Sohn & A. van Gelder. Termination Detection in Logic Programs using Argument Sizes. In *Proceedings of the 10th ACM Symposium on Principles of Database Systems*, Denver, Colorado, 1991.

[Ste94] J. Steinbach. *Termination of Rewriting — Extensions, Comparison and Automatic Generation of Simplification Orderings.* Doctoral Dissertation, Universität Kaiserslautern, Germany, 1994.

[Ste95] J. Steinbach. Automatic Termination Proofs with Transformation Orderings. In *Proceedings of the 6th International Conference on Rewriting Techniques and Applications*, Kaiserslautern, Germany, 1995.

[UV88] J. D. Ullman & A. van Gelder. Efficient Tests for Top-Down Termination of Logical Rules. *Journal of the ACM*, 35(2):345-373, 1988.

[Wal88] C. Walther. Argument-Bounded Algorithms as as Basis for Automated Termination Proofs. In *Proceedings of the 9th International Conference on Automated Deduction*, Argonne, IL, 1988.

[Wal94] C. Walther. On Proving the Termination of Algorithms by Machine. *Artificial Intelligence*, 71(1):101-157, 1994.