Approximating the Domains of Functional and Imperative Programs

Jürgen Brauburger, Jürgen Giesl [1]

FB Informatik, TU Darmstadt, Alexanderstraße 10, 64283 Darmstadt, Germany

Abstract

This paper deals with automated termination analysis of partial functional programs, that is, of functional programs which do not terminate for some input. We present a method to determine their *domains* (resp. non-trivial subsets of their domains) automatically. More precisely, for each functional program a *termination predicate* algorithm is synthesized that only returns true for inputs where the program is terminating. To ease subsequent reasoning about the generated termination predicates we also present a procedure for their simplification. Finally, we show that our method can also be used for automated termination analysis of imperative programs.

1. Introduction

Termination of algorithms is a central problem in software development and formal methods for termination analysis are essential for program verification. While most work on the automation of termination proofs has been done in the areas of *term rewriting systems* (for surveys see e.g. [11,27]) and of *logic programs* (e.g. [24,25,28]), in this paper we focus on *functional programs*.

Up to now all methods for automated termination analysis of functional programs (e.g. [1,3,13,14,23,26,29,32]) aim to prove that a program terminates for *each* input. However, if the termination proof fails then these methods provide no means to find a (sub-)domain where termination is provable. Therefore these methods cannot be used to analyze the termination behavior of *partial* functional programs, i.e., of programs which do not terminate for some input.

Partial functions are often used in practice and therefore tools for automated reasoning about such functions are of vital interest in program analysis [4,22]. Moreover, techniques for handling partial functions are also important

[1] Preliminary version of a paper which appeared in *Science of Computer Programming* 35:113-136, 1999.

for termination analysis of *imperative* programs. The reason is that a direct termination proof for imperative programs is hard to perform automatically. Therefore one attempt to verify their termination is to transform imperative programs into functional ones and to prove termination of these corresponding functional programs instead. In this translation, every *while*-loop is transformed into a separate function (see [19] for example). But in general these functions are *partial*, because termination of *while*-loops often depends on their contexts, i.e., on the preconditions that hold before entering the *while*-loop. So to prove termination of imperative programs in this way, one needs a method for termination analysis of *partial functions* to determine the (sub-)domains where the partial "loop-functions" are terminating.

In this paper we automate Manna's approach for termination analysis of "partial programs" [22]: For every algorithm defining a function f there has to be a *termination predicate*[2] θ_f which specifies the "admissible input" of f (thus, evaluation of f must terminate for each input admitted by the termination predicate). But while in [22] termination predicates have to be provided by the user, in this paper we present a technique to synthesize them *automatically*.

In Section 2 we introduce our functional programming language and sketch the basic approach for proving termination of algorithms. Then in Section 3 we show the requirements termination predicates have to satisfy and based on these requirements we present a procedure for the automated synthesis of termination predicates[3] in Section 4. The generated termination predicates can be used both for further automated and interactive program analysis. To ease the handling of these termination predicates we have developed a procedure for their simplification which is introduced in Section 5. In Section 6 we show how our method can be applied for automated termination analysis of imperative programs. Extensions of our technique are discussed in Section 7. Finally, we give a summary of our method (Section 8) and illustrate its power with a collection of examples.

2. Termination of Algorithms

In this paper we regard an eager first-order functional language with free algebraic data types. To simplify the presentation we restrict ourselves to non-parameterized types and to functions without mutual recursion (see Section 7 for a discussion of possible extensions of our method).

In our language, a data type s is introduced by defining *constructors* $c_1, \ldots, c_k$ that are used to build the data objects of s. Furthermore, for each argu-

[2] Instead of "termination predicates" Manna uses the notion of "input predicates".

[3] Strictly speaking, we synthesize *algorithms* which compute termination predicates. For the sake of brevity sometimes we also refer to these algorithms as "termination predicates".

ment position j of a constructor c_i, a (total) selector d_{ij} is defined such that $d_{ij}(c_i(x_1, \ldots, x_n)) = x_j$. As an example consider the algebraic data type nat for natural numbers. Its objects are built with the constructors $\mathsf{0}$ and succ and we use a selector pred as an inverse function to succ (with $\mathsf{pred}(\mathsf{succ}(x)) = x$ and $\mathsf{pred}(\mathsf{0}) = \mathsf{0}$, i.e., pred is indeed a total function). To ease readability we often write "$\mathsf{1}$" instead of "$\mathsf{succ}(\mathsf{0})$", etc.

For each type s there is a pre-defined equality function "$=$" : $s \times s \to \mathsf{bool}$. Then the following algorithm computes the arithmetical mean of two naturals.

$$
\begin{array}{l}
\mathit{function}\ \mathsf{mean}(x, y : \mathsf{nat}) : \mathsf{nat} \Leftarrow \\
\quad \mathit{if}\ \ x = y\ \ \mathit{then}\ x \\
\qquad\qquad \mathit{else}\ \ \mathsf{mean}(\mathsf{pred}(x), \mathsf{succ}(y))
\end{array}
$$

In general, the body b of an algorithm "$\mathit{function}\ f(x_1 : s_1, \ldots, x_n : s_n) : s \Leftarrow b$" is a term built from the variables $x_1, \ldots, x_n$, constructors, selectors, equality function symbols, function symbols defined by algorithms, and conditionals (where we write "$\mathit{if}\ t_1\ \mathit{then}\ t_2\ \mathit{else}\ t_3$" instead of "$\mathsf{if}(t_1, t_2, t_3)$"). These conditionals are the only functions with non-eager semantics. Thus, when evaluating "$\mathit{if}\ t_1\ \mathit{then}\ t_2\ \mathit{else}\ t_3$", the (boolean) term t_1 is evaluated first and depending on the result of its evaluation either t_2 or t_3 is evaluated afterwards.

To prove *termination* of an algorithm one has to show that in each recursive call a certain *measure* is decreased. For that purpose a *measure function* $|.|$ is used that maps a tuple of data objects $q_1, \ldots, q_n$ to a natural number $|q_1, \ldots, q_n|$. In the following we often abbreviate tuples $q_1, \ldots, q_n$ by q^*.

For example, one might attempt to prove termination of mean with the *size* measure $|.|_{\#}$, where the size of an object of type nat is the number it represents (i.e., the number of succ's it contains). So we have $|\mathsf{0}|_{\#} = 0$, $|\mathsf{succ}(\mathsf{0})|_{\#} = 1$, etc. In general, the *size* $|.|_{\#}$ of an object $c(q_1, \ldots, q_n)$ of type s is defined by

- $|c(q_1, \ldots, q_n)|_{\#} = 1 + |q_{i_1}|_{\#} + \ldots + |q_{i_k}|_{\#}, \quad$ if $i_k > 0$
  (where $i_1, \ldots, i_k$ are all argument positions of c that have type s)
- $|c(q_1, \ldots, q_n)|_{\#} = 0, \quad$ if $i_k = 0$.

As mean is a binary function, for its termination proof we need a measure function on pairs of data objects. Therefore we extend the size measure function to pairs by measuring a pair by the size of the first object, that is, $|q_1, q_2|_{\#} = |q_1|_{\#}$. Hence, to prove termination of mean we now have to verify the following implication. [4]

$$x \neq y \to |\mathsf{pred}(x), \mathsf{succ}(y)|_{\#} < |x, y|_{\#} \tag{1}$$

For instance, C. Walther presented a method to verify implications of the form $\psi \to |t^*|_{\#} < |x^*|_{\#}$ automatically [32]. While in this approach the *size* is

[4] We often use "$t \neq r$" as an abbreviation for $\neg(t = r)$, where the boolean function $\neg$ is defined by an (obvious) algorithm.

used as a fixed measure function, J. Giesl generalized it for *arbitrary* measure functions $|.|$ [14]. Furthermore, he incorporated techniques for the automated synthesis of appropriate measures based on polynomial norms [12,13].

However, these methods fail in proving the implication (1). The reason is that the algorithm for mean does not terminate for all inputs. In fact, mean is a *partial* function, because $\mathsf{mean}(x, y)$ only terminates if the number x is not smaller than the number y and if the difference of x and y is even. For instance, the call $\mathsf{mean}(0, 1)$ leads to the recursive call $\mathsf{mean}(\mathsf{pred}(0), \mathsf{succ}(1))$. As $\mathsf{pred}(0)$ is evaluated to 0, this results in calling $\mathsf{mean}(0, 2)$ and so on. Hence, evaluation of $\mathsf{mean}(0, 1)$ is not terminating. Consequently, any termination proof for mean must fail. For example, (1) is not satisfied if x is 0 and y is 1.

Instead of proving that algorithms terminate for *all* inputs (*total* termination), in the following we are interested in finding subsets of inputs where the algorithms are terminating. Hence, for each algorithm defining a function f we want to generate a *termination predicate* algorithm θ_f where evaluation of θ_f always terminates and if θ_f returns true for some input q^* then evaluation of $f(q^*)$ terminates, too.

Definition 1 Let $f : s_1 \times \ldots \times s_n \to s$ be defined by a (possibly non-terminating) algorithm. A total function $\theta_f : s_1 \times \ldots \times s_n \to \mathsf{bool}$ is a *termination predicate* for f iff for all tuples q^* of data objects, $\theta_f(q^*) = \mathsf{true}$ implies that the evaluation of $f(q^*)$ is terminating.

Of course the problem of determining the *exact* domains of functions is undecidable. As we want to generate termination predicates automatically we therefore only demand that a termination predicate θ_f represents a *sufficient* criterion for f's termination. So in general, a function f may have an infinite number of termination predicates and false is a termination predicate for each function. But of course our aim is to synthesize weaker termination predicates, i.e., termination predicates that return true as often as possible.

3. Requirements for Termination Predicates

In this section we introduce two requirements that are sufficient for termination predicates. In other words, if a (terminating) algorithm satisfies these requirements then it defines a termination predicate for the function under consideration. A procedure for the automated synthesis of such algorithms will be presented in Section 4.

First, we consider simple partial functions like mean (Section 3.1) and afterwards we examine algorithms that call other partial functions (Section 3.2).

3.1. Termination Predicates for Simple Partial Functions

We resume our example and generate a termination predicate θ_{mean} such that evaluation of $\mathsf{mean}(x, y)$ terminates if $\theta_{\mathsf{mean}}(x, y)$ is true. Recall that for proving

total termination one has to show that a certain measure is decreased in each recursive call. But as we illustrated, the algorithm for mean is not always terminating and therefore implication (1) does not hold for all instantiations of x and y. Hence, the central idea for the construction of a termination predicate θ_{mean} is to let $\theta_{\mathsf{mean}}(x, y)$ return true only for those inputs x and y where the measure of x and y is greater than the measure of the corresponding recursive call and to return false for all other inputs. So if evaluation of $\mathsf{mean}(x, y)$ leads to a recursive call (i.e., if $x \neq y$ holds), then $\theta_{\mathsf{mean}}(x, y)$ may only return true if the measure $|\mathsf{pred}(x), \mathsf{succ}(y)|_{\#}$ is smaller than $|x, y|_{\#}$. This yields the following requirement for a termination predicate θ_{mean}.

$$\theta_{\mathsf{mean}}(x, y) \wedge x \neq y \rightarrow |\mathsf{pred}(x), \mathsf{succ}(y)|_{\#} < |x, y|_{\#} \tag{2}$$

For example, the function defined by the following algorithm satisfies (2).

$$\begin{array}{l} \mathit{function}\ \theta_{\mathsf{mean}}(x, y : \mathsf{nat}) : \mathsf{bool} \Leftarrow \\ \quad \mathit{if}\ \ x = y\ \ \mathit{then}\ \mathsf{true} \\ \qquad\qquad \mathit{else}\ \ |\mathsf{pred}(x), \mathsf{succ}(y)|_{\#} < |x, y|_{\#} \end{array}$$

This algorithm for θ_{mean} uses the same case analysis as mean. Since mean terminates in its non-recursive case (if $x = y$), the corresponding result of θ_{mean} is true. For the recursive case (if $x \neq y$), θ_{mean} returns true iff $|\mathsf{pred}(x), \mathsf{succ}(y)|_{\#} < |x, y|_{\#}$ is true. We assume that each measure function $|.|$ is defined by a (terminating) algorithm. Hence, in the result of the second case θ_{mean} calls the algorithm for the computation of the size measure $|.|_{\#}$ and it also calls a (terminating) algorithm to compute the less-than relation "$<$" on natural numbers.

So in general, given an algorithm for f we demand the following requirement for termination predicates θ_f (where $|.|$ is an arbitrary measure function).

$$\begin{array}{l} \text{If evaluation of } f(q^*) \text{ leads to a recursive call } f(p^*), \\ \text{then } \theta_f(q^*) \text{ may only return true if } |p^*| < |q^*| \text{ holds.} \end{array} \tag{Req1}$$

However, (Req1) is not a *sufficient* requirement for termination predicates. For instance, the function θ_{mean} defined above is not a termination predicate for mean although it satisfies requirement (Req1). The reason is that $\theta_{\mathsf{mean}}(1, 0)$ returns true (as $|\mathsf{pred}(1), \mathsf{succ}(0)|_{\#} < |1, 0|_{\#}$ holds). But evaluation of $\mathsf{mean}(1, 0)$ is not terminating because its evaluation leads to the (non-terminating) recursive call $\mathsf{mean}(0, 1)$.

This non-termination is not recognized by θ_{mean} because $\theta_{\mathsf{mean}}(1, 0)$ only checks if the arguments $(0, 1)$ of the *next* recursive call of mean are smaller than the input $(1, 0)$. But it is not guaranteed that *subsequent* recursive calls are also measure decreasing. For example, the next recursive call with the arguments $(0, 1)$ will lead to a subsequent recursive call of mean with the same first argument. So in the subsequent recursive call the measure of the arguments remains the same. Therefore $\theta_{\mathsf{mean}}(1, 0)$ evaluates to true, but application of θ_{mean} to the arguments $(0, 1)$ of the following recursive call yields false.

Hence, in addition to (Req1) we must demand that a termination predicate θ_f remains valid for each recursive call in f's algorithm. This ensures that subsequent recursive calls are also measure decreasing.

$$\begin{array}{l}\text{If evaluation of } f(q^*) \text{ leads to a recursive call } f(p^*),\\ \text{then } \theta_f(q^*) \text{ may only return } \mathsf{true} \text{ if } \theta_f(p^*) \text{ is also } \mathsf{true}.\end{array} \tag{3}$$

In our example, to satisfy the requirements (Req1) and (3) we modify the result of θ_{mean}'s second case by demanding that θ_{mean} also holds for the following recursive call of mean.

$$\begin{array}{l} \mathit{function}\ \theta_{\mathsf{mean}}(x, y : \mathsf{nat}) : \mathsf{bool} \Leftarrow \\ \quad \mathit{if}\ \ x = y\ \ \mathit{then}\ \mathsf{true} \\ \qquad\qquad \mathit{else}\ \ |\mathsf{pred}(x), \mathsf{succ}(y)|_{\#} < |x, y|_{\#}\ \wedge\ \theta_{\mathsf{mean}}(\mathsf{pred}(x), \mathsf{succ}(y)) \end{array}$$

In this algorithm we use the boolean function symbol $\wedge$ to ease readability, where $\varphi_1 \wedge \varphi_2$ abbreviates "*if* φ_1 *then* φ_2 *else* false". Hence, the function $\wedge$ does *not* have eager semantics, because terms in a conjunction are evaluated from left to right. In other words, given a conjunction $\varphi_1 \wedge \varphi_2$ of boolean terms (which we also refer to as "formulas"), φ_1 is evaluated first. If the value of φ_1 is false, then false is returned, otherwise φ_2 is evaluated and its value is returned. Note that we need a *lazy* conjunction function $\wedge$ to ensure termination of θ_{mean}. It guarantees that evaluation of $\theta_{\mathsf{mean}}(x, y)$ can only lead to a recursive call $\theta_{\mathsf{mean}}(\mathsf{pred}(x), \mathsf{succ}(y))$ if the measure of the recursive arguments $|\mathsf{pred}(x), \mathsf{succ}(y)|_{\#}$ is smaller than the measure of the inputs $|x, y|_{\#}$.

The above algorithm really defines a termination predicate for mean, that is, θ_{mean} is a total function and the truth of θ_{mean} is sufficient for the termination of mean. This algorithm for θ_{mean} was constructed in order to obtain an algorithm satisfying the requirements (Req1) and (3). In Section 4 we will show that this construction can easily be automated. A closer look at θ_{mean} reveals that θ_{mean} returns true iff x is greater than or equal to y and the difference of x and y is even. As $\mathsf{mean}(x, y)$ is *only* terminating for those inputs, in this example we have even generated the weakest possible termination predicate. Thus, θ_{mean} returns true not only for a subset but for *all* elements of mean's domain.

3.2. Algorithms Calling Other Partial Functions

In general (Req1) and (3) are no sufficient criteria for termination predicates. These requirements can only be used for algorithms like mean which (apart from recursive calls) only call other *total* functions (like $=$, succ, and pred).

In this section we will examine algorithms that call other *partial* functions. As an example consider the algorithm for $\mathsf{list\_half}(l)$ that halves each element of a list l by application of mean. Objects of the data type list are built with the constructors nil and cons, where $\mathsf{cons}(x, k)$ represents the insertion of the number x into the list k. We also use the selectors head and tail, where head

returns the first element of a list and tail returns a list without its first element (i.e., $\mathsf{head}(\mathsf{cons}(x,k)) = x$, $\mathsf{head}(\mathsf{nil}) = 0$, $\mathsf{tail}(\mathsf{cons}(x,k)) = k$, $\mathsf{tail}(\mathsf{nil}) = \mathsf{nil}$).

$$
\begin{array}{l}
\mathit{function}\ \mathsf{list\_half}(l : \mathsf{list}) : \mathsf{list} \Leftarrow \\
\quad \mathit{if}\ \ l = \mathsf{nil}\ \ \mathit{then}\ \mathsf{nil} \\
\qquad\qquad\ \ \mathit{else}\ \ \mathsf{cons}(\mathsf{mean}(\mathsf{head}(l), 0), \mathsf{list\_half}(\mathsf{tail}(l)))
\end{array}
$$

We construct the following algorithm for $\theta_{\mathsf{list\_half}}$ by measuring lists by their size (or length), that is, $|\mathsf{nil}|_{\#} = 0$ and $|\mathsf{cons}(x,l)|_{\#} = 1 + |l|_{\#}$.

$$
\begin{array}{l}
\mathit{function}\ \theta_{\mathsf{list\_half}}(l : \mathsf{list}) : \mathsf{bool} \Leftarrow \\
\quad \mathit{if}\ \ l = \mathsf{nil}\ \ \mathit{then}\ \mathsf{true} \\
\qquad\qquad\ \ \mathit{else}\ \ |\mathsf{tail}(l)|_{\#} < |l|_{\#} \wedge \theta_{\mathsf{list\_half}}(\mathsf{tail}(l))
\end{array}
$$

Although this algorithm defines a function satisfying (Req1) and (3), it is not a termination predicate for list_half. For example, $\theta_{\mathsf{list\_half}}(\mathsf{cons}(1,\mathsf{nil}))$ evaluates to true because the size of the empty list nil is smaller than the size of $\mathsf{cons}(1,\mathsf{nil})$. But evaluation of $\mathsf{list\_half}(\mathsf{cons}(1,\mathsf{nil}))$ is not terminating as it leads to the (non-terminating) evaluation of $\mathsf{mean}(1,0)$.

The problem is that $\theta_{\mathsf{list\_half}}$ only checks if *recursive* calls of list_half are measure decreasing but it does not guarantee the termination of *other* algorithms called. Therefore we have to demand that $\theta_{\mathsf{list\_half}}$ ensures termination of the subsequent call of mean, that is, in the second case $\theta_{\mathsf{list\_half}}(l)$ must imply $\theta_{\mathsf{mean}}(\mathsf{head}(l), 0)$.

So we replace (3) by a requirement that guarantees the truth of $\theta_g(p^*)$ for *all* function calls $g(p^*)$ in f's algorithm (i.e., also for functions g different from f):

$$
\begin{array}{l}
\text{If evaluation of } f(q^*) \text{ leads to a function call } g(p^*), \\
\text{then } \theta_f(q^*) \text{ may only return true if } \theta_g(p^*) \text{ is also true.}
\end{array}
\qquad \text{(Req2)}
$$

Note that (Req2) must also be demanded for non-recursive cases. The function $\theta_{\mathsf{list\_half}}$ defined by the following algorithm satisfies (Req1) and the extended requirement (Req2).

$$
\begin{array}{l}
\mathit{function}\ \theta_{\mathsf{list\_half}}(l : \mathsf{list}) : \mathsf{bool} \Leftarrow \\
\quad \mathit{if}\ \ l = \mathsf{nil}\ \ \mathit{then}\ \mathsf{true} \\
\qquad\qquad\ \ \mathit{else}\ \ \theta_{\mathsf{mean}}(\mathsf{head}(l), 0) \wedge |\mathsf{tail}(l)|_{\#} < |l|_{\#} \wedge \theta_{\mathsf{list\_half}}(\mathsf{tail}(l))
\end{array}
$$

The above algorithm in fact defines a termination predicate for list_half. Analyzing the algorithm one notices that $\theta_{\mathsf{list\_half}}(l)$ returns true iff all elements of l are even numbers. As evaluation of $\mathsf{list\_half}(l)$ only terminates for such inputs, we have synthesized the weakest possible termination predicate again.

Note that algorithms may also call partial functions in their *conditions*. For example consider the following algorithm for computing the dual logarithm that calls mean in its condition.

$$
\begin{array}{l}
\mathit{function}\ \mathsf{dual\_log}(x : \mathsf{nat}) : \mathsf{nat} \Leftarrow \\
\quad \mathit{if}\ \ \mathsf{mean}(x, \mathsf{0}) = \mathsf{1}\ \ \mathit{then}\ \mathsf{1} \\
\qquad\qquad\qquad\quad \mathit{else}\ \ \mathsf{succ}(\mathsf{dual\_log}(\mathsf{mean}(x, \mathsf{0})))
\end{array}
$$

This algorithm does not terminate for odd inputs, since in the condition the term $\mathsf{mean}(x, \mathsf{0})$ must be evaluated. Therefore due to (Req2), $\theta_{\mathsf{dual\_log}}$ must ensure that all resulting calls of the partial function mean are terminating. Thus, $\theta_{\mathsf{dual\_log}}(x)$ must imply $\theta_{\mathsf{mean}}(x, \mathsf{0})$. The following algorithm for $\theta_{\mathsf{dual\_log}}$ satisfies both requirements (Req1) and (Req2).

$$
\begin{array}{l}
\mathit{function}\ \theta_{\mathsf{dual\_log}}(x : \mathsf{nat}) : \mathsf{bool} \Leftarrow \\
\theta_{\mathsf{mean}}(x, \mathsf{0})\ \ \wedge\ (\ \mathit{if}\ \ \mathsf{mean}(x, \mathsf{0}) = \mathsf{1} \\
\qquad\qquad\qquad\quad \mathit{then}\ \ \mathsf{true} \\
\qquad\qquad\qquad\quad \mathit{else}\ \ \theta_{\mathsf{mean}}(x, \mathsf{0}) \wedge |\mathsf{mean}(x, \mathsf{0})|_{\#} < |x|_{\#} \wedge \theta_{\mathsf{dual\_log}}(\mathsf{mean}(x, \mathsf{0}))\ )
\end{array}
$$

The above algorithm first checks if the call of the algorithm mean in the condition of dual_log is terminating. If the corresponding termination predicate $\theta_{\mathsf{mean}}(x, \mathsf{0})$ is false, then $\theta_{\mathsf{dual\_log}}$ also returns false. Otherwise, evaluation of $\theta_{\mathsf{dual\_log}}$ continues as usual.

This algorithm really defines a termination predicate for dual_log. Analysis of $\theta_{\mathsf{dual\_log}}$ reveals that it returns true iff the input is a power of 2 different from 1. This is the weakest possible termination predicate for dual_log.

The following lemma states that the two requirements we have derived are in fact sufficient for termination predicates. In other words, if a total function θ_f satisfies these two requirements then it is a termination predicate for f.

Lemma 2 *Let θ_f be a function satisfying (Req1) and (Req2). If $\theta_f(q^*)$ evaluates to* true *for some data objects q^*, then evaluation of $f(q^*)$ is terminating.*

Proof. Suppose that there exist data objects q^* such that $\theta_f(q^*)$ returns true but evaluation of $f(q^*)$ does not terminate. Then let q^* be the smallest such data objects, i.e., for all objects p^* with a measure $|p^*|$ smaller than $|q^*|$ the truth of $\theta_f(p^*)$ implies termination of $f(p^*)$.

As we have excluded mutual recursion we may assume that for all other functions g (that are called by f), the truth of θ_g really implies termination of g. Hence, requirement (Req2) ensures that evaluation of $f(q^*)$ can only lead to terminating calls of *other* functions g. Therefore the non-termination of $f(q^*)$ cannot be caused by another function g.

So evaluation of $f(q^*)$ must lead to a recursive call $f(p^*)$. But because of requirement (Req1), p^* has a smaller measure than q^*. Hence, due to the minimality of q^*, $f(p^*)$ must be terminating (as (Req2) ensures that $\theta_f(p^*)$ also returns true). So the recursive calls of f cannot cause non-termination either. Therefore evaluation of $f(q^*)$ must also be terminating. □

4. Automated Generation of Termination Predicates

In this section we show how termination predicates can be synthesized automatically. Given a functional program f, we present a technique to generate a (terminating) algorithm for θ_f satisfying the requirements (Req1) and (Req2). Then due to Lemma 2 this algorithm computes a termination predicate for f.

Requirement (Req2) demands that θ_f may only return true if evaluation of all terms in the conditions and results of f is terminating. Therefore we extend the idea of termination predicates from *algorithms* to arbitrary *terms*.

So for each term t we construct a boolean term $\Theta(t)$ (a *termination formula*) such that for each substitution σ of t's variables by data objects we have:

- evaluation of $\sigma(\,\Theta(t)\,)$ is terminating and
- if $\sigma(\,\Theta(t)\,) = \mathsf{true}$, then evaluation of $\sigma(t)$ is also terminating.

For example, a termination formula for $\mathsf{dual\_log}(\mathsf{mean}(x, 0))$ is $\theta_{\mathsf{mean}}(x, 0) \wedge \theta_{\mathsf{dual\_log}}(\mathsf{mean}(x, 0))$, because due to the eager nature of our functional language in this term mean is evaluated before evaluating $\mathsf{dual\_log}$. So termination formulas have to guarantee that a subterm $g(t^*)$ is only evaluated if $\theta_g(t^*)$ holds. In general, termination formulas are constructed by the following rules.

$$
\begin{array}{lll}
\Theta(x) & :\equiv \mathsf{true}, & \text{for variables } x, \quad \text{(i)} \\
\Theta(g(t_1,\ldots,t_n)) & :\equiv \Theta(t_1) \wedge \ldots \wedge \Theta(t_n) \wedge\ \theta_g(t_1,\ldots,t_n), & \text{for functions } g, \text{ (ii)} \\
\Theta(\mathit{if}\ t_1\ \mathit{then}\ t_2\ \mathit{else}\ t_3) & :\equiv \Theta(t_1)\ \wedge\ \mathit{if}\ t_1\ \mathit{then}\ \Theta(t_2)\ \mathit{else}\ \Theta(t_3). & \text{(iii)}
\end{array}
$$

In Rule (ii), if g is a constructor, a selector, or an equality function, then we define $\theta_g(x^*) = \mathsf{true}$, because those functions are total.

To satisfy requirement (Req2), θ_f must ensure that evaluation of all terms in the body of an algorithm f terminates. So if f is defined by the algorithm "*function* $f(x_1 : s_1, \ldots, x_n : s_n) : s \Leftarrow b$", then θ_f has to check whether the termination formula $\Theta(b)$ of f's body is true.

But the body of f can also contain recursive calls $f(t^*)$. To satisfy requirement (Req1) we must additionally ensure that the measure $|t^*|$ of recursive calls is smaller than the measure of the inputs $|x^*|$. Therefore for recursive calls $f(t^*)$ we have to change the definition of termination formulas as follows.

$$\Theta(f(t_1,\ldots,t_n)) :\equiv \Theta(t_1) \wedge \ldots \wedge \Theta(t_n) \wedge |t_1,\ldots,t_n| < |x_1,\ldots,x_n| \wedge \theta_f(t_1,\ldots,t_n) \qquad \text{(iv)}$$

In this way we obtain the following procedure for the generation of termination predicates.

Theorem 3 *Given an algorithm "function* $f(x_1 : s_1, \ldots, x_n : s_n) : s \Leftarrow b$*", we define the algorithm "function* $\theta_f(x_1 : s_1, \ldots, x_n : s_n) : \mathsf{bool} \Leftarrow \Theta(b)$*", where the termination formula* $\Theta(b)$ *is constructed by the rules (i)–(iv). Then this algorithm defines a termination predicate* θ_f *for* f*, i.e., this algorithm is terminating and if* $\theta_f(q^*)$ *returns* true*, then evaluation of* $f(q^*)$ *is also terminating.*

Proof. For all terms t and all substitutions σ of t's variables by data objects, $\sigma(\Theta(t)) = \mathsf{true}$ implies $\sigma(\Theta(t|_\pi)) = \mathsf{true}$, whenever evaluation of $\sigma(t)$ leads to evaluation of $\sigma(t|_\pi)$. (This is easily proved by induction on the position π).

Thus, due to the construction principles (ii) and (iv), θ_f satisfies the requirements (Req1) and (Req2). Hence by Lemma 2, θ_f is "partially correct". So if $\theta_f(q^*) = \mathsf{true}$ for some data objects q^*, then evaluation of $f(q^*)$ is terminating.

It remains to show that θ_f is total. We first prove the following lemma.

If $\sigma(\Theta(t)) = \mathsf{true}$ for some term t, then evaluation of $\sigma(t)$ is terminating. (4)

We use structural induction on t. If t has the form $g(t^*)$, then the truth of $\sigma(\Theta(t))$ implies the truth of all $\sigma(\Theta(t_i))$ and of $\sigma(\theta_g(t^*))$. By the induction hypothesis, all $\sigma(t_i)$ can be evaluated to data objects q_i. As we excluded mutual recursion we may assume that the termination predicates for all functions $g \neq f$ are partially correct. If $g = f$, then the partial correctness of θ_f follows from Lemma 2 as remarked above. Thus, the truth of $\theta_g(q^*)$ implies termination of $g(q^*)$. The remaining cases of the induction proof are straightforward.

Now we can prove the totality of θ_f. Suppose that there exist data objects q^* such that evaluation of $\theta_f(q^*)$ does not halt. Let q^* be the smallest such data objects, i.e., for all objects p^* with $|p^*| < |q^*|$ evaluation of $\theta_f(p^*)$ is terminating. Let σ denote the substitution $\{x_1/q_1, \ldots, x_n/q_n\}$. To refute our assumption, we show that $\sigma(\Theta(t))$ is terminating for any subterm t of the body b, provided that evaluation of $\sigma(b)$ leads to evaluation of $\sigma(t)$. Then, in the case $t := b$ we obtain the desired contradiction.

The proof is by structural induction on t. If $t = g(t^*)$, then the induction hypothesis implies termination of all $\sigma(\Theta(t_i))$. If one of the $\sigma(\Theta(t_i))$ is false, then termination of $\sigma(\Theta(t))$ is obvious. Otherwise, $\sigma(t^*)$ can be evaluated to data objects p^* by (4). In the case $g \neq f$, termination of $\theta_g(p^*)$ follows from the assumption that the termination predicates for all other functions are total (due to the exclusion of mutual recursion). If $g = f$, then we first have to compute $|p^*| < |q^*|$. If $|p^*| < |q^*|$ yields false, then $\sigma(\Theta(t))$ is trivially terminating. Otherwise, by the minimality of q^* evaluation of $\theta_f(p^*)$ halts and thus, $\sigma(\Theta(t))$ is also terminating. The remaining proof cases are analogous. □

The construction of algorithms for termination predicates according to Theorem 3 can be automated directly. So by this theorem we have developed a procedure for the automated generation of termination predicates. For instance, the termination predicate algorithms for mean, list_half, and dual_log in the last section were built according to Theorem 3 (where for the sake of brevity we omitted termination predicates for *total* functions because such predicates always return true). As demonstrated, the generated termination predicates often are as weak as possible, that is, they often describe the *whole* domain of the partial function under consideration (instead of just a sub-domain).

5. Simplification of Termination Predicates

In the previous section we presented a method for the automated generation of algorithms that define termination predicates. But sometimes the synthesized algorithms are unnecessarily complex. To ease subsequent reasoning about termination predicates now we introduce a procedure to *simplify* the generated termination predicate algorithms.

5.1. Application of Induction Lemmata

First, the well-known induction lemma method by Boyer and Moore [3] is used to eliminate (some of) the inequalities $|t^*| < |x^*|$ (which ensure that recursive calls are measure decreasing) from the termination predicate algorithms. Elimination of these inequalities simplifies the algorithms considerably and often enables the execution of subsequent simplification steps.

An *induction lemma* points out that under a certain hypothesis Δ some operation drives some measure down. So induction lemmata have the form

$$\Delta \rightarrow |t^*| < |x^*|.$$

In the system of Boyer and Moore induction lemmata have to be provided by the *user*. However, C. Walther presented a method to generate a certain class of induction lemmata for the *size* measure function $|.|_{\#}$ *automatically* [32] and J. Giesl generalized this approach towards arbitrary measure functions [14].

Both methods have been implemented in the induction theorem prover INKA [20,31]. Walther's technique verifies termination of many examples automatically (a collection of 60 such algorithms can be found in [30]) and it also proved successful for almost all examples from the data base of [3]. However, three algorithms of this data base terminate with a measure different from *size* and therefore, his approach fails for these examples. The method by Giesl overcomes this drawback and performs successfully on an even larger collection of benchmarks (including all 82 algorithms from [3] and all 60 examples from [15]). For instance, the induction lemmata needed in the following examples can be synthesized by Walther's and Giesl's method.

While Boyer and Moore, Walther, and Giesl use induction lemmata for total termination proofs, we will now illustrate their use for the simplification of termination predicate algorithms. Furthermore, we sketch the main ideas for the automated generation of induction lemmata according to [14,17,32].

Consider again the termination predicate θ_{mean} from Section 3.[5]

$$\begin{array}{l} \mathit{function}\ \theta_{\mathsf{mean}}(x, y : \mathsf{nat}) : \mathsf{bool} \Leftarrow \\ \quad \mathit{if}\ \ x = y\ \ \mathit{then}\ \mathsf{true} \\ \qquad\qquad \mathit{else}\ \ (\ \mathit{if}\ |\mathsf{pred}(x), \mathsf{succ}(y)|_{\#} < |x, y|_{\#}\ \ \mathit{then}\ \theta_{\mathsf{mean}}(\mathsf{pred}(x), \mathsf{succ}(y)) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{else}\ \ \mathsf{false}\ ) \end{array}$$

[5] Recall that $\varphi_1 \wedge \varphi_2$ is an abbreviation for "*if* φ_1 *then* φ_2 *else* false".

In order to eliminate the inequality $|\mathsf{pred}(x), \mathsf{succ}(y)|_{\#} < |x, y|_{\#}$, we search for an induction lemma of the form $\Delta \rightarrow |\mathsf{pred}(x), \mathsf{succ}(y)|_{\#} < |x, y|_{\#}$. The size measure function on pairs was defined by measuring a pair by the size of the first object, i.e., $|q_1, q_2|_{\#} = |q_1|_{\#}$. Hence, we only need a hypothesis Δ satisfying $\Delta \rightarrow |\mathsf{pred}(x)|_{\#} < |x|_{\#}$. For instance, an appropriate hypothesis may be generated by Walther's method.

His technique tries to prove that the size of a function's value is *bounded* by the size of one of its arguments. So *non-strict* inequalities of the form $|g(\ldots x_i \ldots)|_{\#} \leq |x_i|_{\#}$ are verified. If the automated verification of such an inequality succeeds, then based on that proof a (totally terminating) *difference predicate* algorithm δ_g is generated such that $\delta_g(x^*) \rightarrow |g(\ldots x_i \ldots)|_{\#} < |x_i|_{\#}$ holds. In other words, δ_g implies that x_i is a *strict* upper bound for g.

For instance, Walther's system verifies the inequality $|\mathsf{pred}(x)|_{\#} \leq |x|_{\#}$ which establishes that pred is bounded by its only argument. Based on this (trivial) verification the following algorithm for δ_{pred} is generated.

$$
\begin{array}{l}
\mathit{function}\ \delta_{\mathsf{pred}}(x : \mathsf{nat}) : \mathsf{bool} \Leftarrow \\
\quad \mathit{if}\ \ x = \mathsf{0}\ \ \mathit{then}\ \mathsf{false} \\
\qquad\qquad\ \ \mathit{else}\ \ \mathsf{true}
\end{array}
$$

So for each selector d_{ij} that is associated with a constructor c_i, a difference predicate algorithm $\delta_{d_{ij}}(x)$ is synthesized that returns true iff $x = c_i(q^*)$ for some q^*.

Difference predicates like δ_{pred} are used to generate induction lemmata. For instance, since δ_{pred} is sufficient for $|\mathsf{pred}(x)|_{\#} < |x|_{\#}$ by construction, the following induction lemma is generated by Walther's method.

$$\delta_{\mathsf{pred}}(x) \rightarrow |\mathsf{pred}(x)|_{\#} < |x|_{\#} \tag{5}$$

Thus, in mean's termination predicate algorithm we can now replace the inequality $|\mathsf{pred}(x)|_{\#} < |x|_{\#}$ by $\delta_{\mathsf{pred}}(x)$ which yields the following simplified algorithm.

$$
\begin{array}{l}
\mathit{function}\ \theta_{\mathsf{mean}}(x, y : \mathsf{nat}) : \mathsf{bool} \Leftarrow \\
\quad \mathit{if}\ \ x = y\ \ \mathit{then}\ \mathsf{true} \\
\qquad\qquad\ \ \mathit{else}\ \ (\ \mathit{if}\ \delta_{\mathsf{pred}}(x)\ \mathit{then}\ \theta_{\mathsf{mean}}(\mathsf{pred}(x), \mathsf{succ}(y)) \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathit{else}\ \ \mathsf{false}\ )
\end{array}
$$

As another example consider the following multiplication algorithm.

$$
\begin{array}{l}
\mathit{function}\ \mathsf{times}(x, y : \mathsf{nat}) : \mathsf{nat} \Leftarrow \\
\quad \mathit{if}\ \ x = \mathsf{0}\ \ \mathit{then}\ \ \mathsf{0} \\
\qquad\qquad\ \ \mathit{else}\ \ \ (\ \mathit{if}\ \ \mathsf{even}(x)\ \ \mathit{then}\ \ \mathsf{times}(\mathsf{mean}(x, \mathsf{0}), \mathsf{double}(y)) \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathit{else}\ \ \ \mathsf{plus}(y, \mathsf{times}(\mathsf{pred}(x), y))\ )
\end{array}
$$

The value of $\mathsf{times}(x, y)$ is computed as follows. If x is even, then $x * y$ equals

$\frac{x}{2} * \mathsf{double}(y)$. Hence, in this case the algorithm is called recursively where the value of x is halved and the value of y is doubled. If x is odd, then $x * y$ equals $y + \mathsf{pred}(x) * y$. For that purpose, the algorithm uses the total auxiliary functions even, double, plus and the partial function mean.

The algorithm for times terminates for each input. However, as termination of times depends on the termination behavior of the partial function mean, to prove termination of times one needs a method for termination analysis of *partial* functions. Therefore all existing techniques for total termination proofs fail for functions like times.

Using the procedure of Theorem 3 the following termination predicate algorithm is generated. In this algorithm we neglect the calls of the termination predicates θ_{even}, θ_{double}, and θ_{plus} as even, double, and plus are defined by totally terminating algorithms and therefore θ_{even}, θ_{double}, and θ_{plus} always return true.

$$
\begin{array}{l}
\mathit{function}\ \theta_{\mathsf{times}}(x, y : \mathsf{nat}) : \mathsf{bool} \Leftarrow \\
\quad \mathit{if}\ x = \mathsf{0}\ \ \mathit{then}\ \ \mathsf{true} \\
\qquad\qquad \mathit{else}\ \ (\ \mathit{if}\ \mathsf{even}(x)\ \ \mathit{then}\ \ \theta_{\mathsf{mean}}(x, \mathsf{0}) \wedge |\mathsf{mean}(x, \mathsf{0}), \mathsf{double}(y)|_{\#} < |x, y|_{\#} \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\, \theta_{\mathsf{times}}(\mathsf{mean}(x, \mathsf{0}), \mathsf{double}(y)) \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathit{else}\ \ |\mathsf{pred}(x), y|_{\#} < |x, y|_{\#} \wedge \theta_{\mathsf{times}}(\mathsf{pred}(x), y)\ )
\end{array}
$$

The inequality $|\mathsf{pred}(x), y|_{\#} < |x, y|_{\#}$ of θ_{times}'s third result may be replaced by $\delta_{\mathsf{pred}}(x)$ according to the induction lemma (5). The inequality in the second result of θ_{times} is only evaluated if this evaluation is terminating, that is, if $\theta_{\mathsf{mean}}(x, \mathsf{0})$ holds. So in order to eliminate this inequality, we look for an induction lemma of the form

$$\theta_{\mathsf{mean}}(x, \mathsf{0}) \wedge \Delta \rightarrow |\mathsf{mean}(x, \mathsf{0}), \mathsf{double}(y)|_{\#} < |x, y|_{\#}.$$

For that purpose we again use Walther's technique. First, the non-strict inequality $|\mathsf{mean}(x, y)|_{\#} \leq |x|_{\#}$ is verified, i.e., mean is bounded by its first argument. Then the algorithm for the difference predicate δ_{mean} is generated, where $\delta_{\mathsf{mean}}(x, y)$ must ensure that the size of $\mathsf{mean}(x, y)$ is *strictly* smaller than the size of the first argument x whenever evaluation of $\mathsf{mean}(x, y)$ halts.

$$\theta_{\mathsf{mean}}(x, y) \wedge \delta_{\mathsf{mean}}(x, y) \rightarrow |\mathsf{mean}(x, y)|_{\#} < |x|_{\#} \tag{6}$$

The following algorithm for δ_{mean} is constructed *inductively* such that it satisfies implication (6), where "$\varphi_1 \vee \varphi_2$" abbreviates "*if* φ_1 *then* true *else* φ_2".

$$
\begin{array}{l}
\mathit{function}\ \delta_{\mathsf{mean}}(x, y : \mathsf{nat}) : \mathsf{bool} \Leftarrow \\
\quad \mathit{if}\ \ x = y\ \ \mathit{then}\ \mathsf{false} \\
\qquad\qquad \mathit{else}\ \ (\ \mathit{if}\ \delta_{\mathsf{pred}}(x)\ \mathit{then}\ \delta_{\mathsf{mean}}(\mathsf{pred}(x), \mathsf{succ}(y)) \vee \delta_{\mathsf{pred}}(x) \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathit{else}\ \ \mathsf{true}\ )
\end{array}
$$

This algorithm for δ_{mean} uses the same case analysis as θ_{mean}. Under the condition $x = y$, the result of $\mathsf{mean}(x, y)$ is x. Thus, $|\mathsf{mean}(x, y)|_{\#} < |x|_{\#}$ evaluates to $|x|_{\#} < |x|_{\#}$. Since this inequality is false for each x, the algorithm

δ_{mean} has the result false in that case. Under the condition $x \neq y$, the result of $\mathsf{mean}(x, y)$ is $\mathsf{mean}(\mathsf{pred}(x), \mathsf{succ}(y))$. Hence, $|\mathsf{mean}(x, y)|_{\#} < |x|_{\#}$ holds iff $|\mathsf{mean}(\mathsf{pred}(x), \mathsf{succ}(y))|_{\#} < |x|_{\#}$. As mean and pred are bounded by their first arguments, we have

$$|\mathsf{mean}(\mathsf{pred}(x), \mathsf{succ}(y))|_{\#} \leq |\mathsf{pred}(x)|_{\#} \leq |x|_{\#}. \tag{7}$$

If $\delta_{\mathsf{mean}}(\mathsf{pred}(x), \mathsf{succ}(y)) \vee \delta_{\mathsf{pred}}(x)$ is satisfied, then the first or the second inequality in (7) is strict. Hence, the second result of δ_{mean} is indeed sufficient for $|\mathsf{mean}(\mathsf{pred}(x), \mathsf{succ}(y))|_{\#} < |x|_{\#}$. For the condition $x \neq y \wedge \neg\delta_{\mathsf{pred}}(x)$ the algorithm for δ_{mean} trivially satisfies (6). The above algorithm terminates by construction as it is called recursively under the same condition and with the same arguments as the totally terminating algorithm for θ_{mean}.[6] For more details on the automated synthesis of difference predicates see [17,32].

So (6) is a valid induction lemma, because the result of $\mathsf{mean}(x, \mathsf{0})$ is smaller than x, provided that $\mathsf{mean}(x, \mathsf{0})$ terminates and that $\delta_{\mathsf{mean}}(x, \mathsf{0})$ evaluates to true.

Since in the result of θ_{times} the truth of $\theta_{\mathsf{mean}}(x, \mathsf{0})$ is guaranteed before evaluating the inequality $|\mathsf{mean}(x, \mathsf{0}), \mathsf{double}(y)|_{\#} < |x, y|_{\#}$, we can now replace this inequality by $\delta_{\mathsf{mean}}(x, \mathsf{0})$ which yields the following simplified algorithm.

$$\begin{array}{l} \mathit{function}\ \theta_{\mathsf{times}}(x, y : \mathsf{nat}) : \mathsf{bool} \Leftarrow \\ \quad \mathit{if}\ \ x = \mathsf{0}\ \ \mathit{then}\ \ \mathsf{true} \\ \qquad\qquad \mathit{else}\ \ (\ \mathit{if}\ \ \mathsf{even}(x)\ \ \mathit{then}\ \ \theta_{\mathsf{mean}}(x, \mathsf{0}) \wedge \delta_{\mathsf{mean}}(x, \mathsf{0}) \\ \qquad\qquad\qquad\qquad\qquad\quad \wedge \theta_{\mathsf{times}}(\mathsf{mean}(x, \mathsf{0}), \mathsf{double}(y)) \\ \qquad\qquad\qquad\qquad\quad \mathit{else}\ \ \delta_{\mathsf{pred}}(x) \wedge \theta_{\mathsf{times}}(\mathsf{pred}(x), y)\ ) \end{array}$$

So in general, if the body of an algorithm contains an inequality $|t^*| < |x^*|$ that will only be evaluated under the condition ψ, then our simplification procedure looks for an induction lemma of the form

$$\psi \wedge \Delta \rightarrow |t^*| < |x^*|.$$

If such an induction lemma is known (or can be synthesized) then the inequality $|t^*| < |x^*|$ is replaced by Δ.

We have sketched how appropriate induction lemmata are generated automatically following the approach of Walther [32]. However, this technique is restricted to one single fixed measure function, viz. the *size* measure. An extension of Walther's method to arbitrary measures is presented in [14] and an adaptation of this refined method to partial functions is described in [7]. In all these approaches the generation of measure functions and induction lemmata $\psi \wedge \Delta \rightarrow |t^*| < |x^*|$ is based on the analysis of the auxiliary functions in the *arguments* t^* of the recursive calls. We recently developed a new tech-

[6] In fact, the body of δ_{mean}'s algorithm can be subsequently simplified to "*if* $x = y$ *then* false *else* true " using the simplification techniques of the following sections.

nique that also examines auxiliary functions in the *conditions* ψ during that synthesis [8].

5.2. Subsumption Elimination

In the next simplification step *redundant terms* are *eliminated* from the termination predicate algorithms. Recall that $\theta_{\mathsf{mean}}(x, y)$ returns true iff x is greater than or equal to y and the difference of x and y is even. Hence the term $\theta_{\mathsf{mean}}(x, \mathsf{0})$ in the result of θ_{times}'s second case evaluates to true iff x is even. So the condition of the second case implies the truth of $\theta_{\mathsf{mean}}(x, \mathsf{0})$. In other words we can verify

$$x \neq \mathsf{0} \wedge \mathsf{even}(x) \;\rightarrow\; \theta_{\mathsf{mean}}(x, \mathsf{0}). \tag{8}$$

For that reason the *subsumed term* $\theta_{\mathsf{mean}}(x, \mathsf{0})$ may be eliminated from the second case of θ_{times} which yields

$$\begin{array}{llll} \mathit{if}\ x = \mathsf{0} & \mathit{then} & \mathsf{true} & \\ & \mathit{else} & (\ \mathit{if}\ \ \mathsf{even}(x) & \mathit{then}\ \ \delta_{\mathsf{mean}}(x, \mathsf{0}) \wedge \theta_{\mathsf{times}}(\mathsf{mean}(x, \mathsf{0}), \mathsf{double}(y)) \\ & & & \mathit{else}\ \ \delta_{\mathsf{pred}}(x) \wedge \theta_{\mathsf{times}}(\mathsf{pred}(x), y)\ ). \end{array}$$

In a similar way we may eliminate the terms $\delta_{\mathsf{mean}}(x, \mathsf{0})$ and $\delta_{\mathsf{pred}}(x)$ from the algorithm θ_{times}. As $\delta_{\mathsf{mean}}(x, y)$ returns true iff $x \neq y$, the term $\delta_{\mathsf{mean}}(x, \mathsf{0})$ (as well as $\delta_{\mathsf{pred}}(x)$) is true for each x greater than $\mathsf{0}$. Hence we can easily verify

$$x \neq \mathsf{0} \wedge \ \ \mathsf{even}(x) \;\rightarrow\; \delta_{\mathsf{mean}}(x, \mathsf{0}), \tag{9}$$

$$x \neq \mathsf{0} \wedge \neg\mathsf{even}(x) \;\rightarrow\; \delta_{\mathsf{pred}}(x). \tag{10}$$

So the *subsumed terms* $\delta_{\mathsf{mean}}(x, \mathsf{0})$ and $\delta_{\mathsf{pred}}(x)$ can also be eliminated which results in the following algorithm for θ_{times}.

$$\begin{array}{l} \mathit{function}\ \theta_{\mathsf{times}}(x, y : \mathsf{nat}) : \mathsf{bool} \Leftarrow \\ \quad \begin{array}{llll} \mathit{if}\ x = \mathsf{0} & \mathit{then} & \mathsf{true} & \\ & \mathit{else} & (\ \mathit{if}\ \ \mathsf{even}(x) & \mathit{then}\ \ \theta_{\mathsf{times}}(\mathsf{mean}(x, \mathsf{0}), \mathsf{double}(y)) \\ & & & \mathit{else}\ \ \theta_{\mathsf{times}}(\mathsf{pred}(x), y)\ ) \end{array} \end{array}$$

According to [32] we call formulas like (8), (9), and (10) *subsumption formulas*. So in general, if a boolean term ψ_2 is evaluated under the condition ψ_1 and if the subsumption formula

$$\psi_1 \rightarrow \psi_2$$

can be verified, then our simplification procedure replaces the term ψ_2 by true. (Subsequently of course, in a conjunction the term true may be eliminated.)

For the automated verification of subsumption formulas an *induction theorem proving system* is used (e.g. one of those described in [2,3,10,20,21,31]). For instance, the subsumption formula (8) can be verified by an induction

proof and subsumption formulas (9) and (10) can already be proved by case analysis and propositional reasoning only.

5.3. Recursion Elimination

Now we *eliminate* the *recursive calls* of θ_{times} according to the *recursion elimination* technique of Walther [32]. If we can verify that evaluation of a recursive call $\theta_f(t^*)$ always yields the same result (i.e., it always yields true or it always yields false) then we can replace the recursive call $\theta_f(t^*)$ by this result. In this way it is possible to replace both recursive calls of θ_{times} by the value true.

The reason is that the arguments of θ_{times}'s recursive calls always satisfy the condition of the first, second, or third case. So due to the termination of θ_{times} after a finite number of recursive calls θ_{times} will be called with arguments that satisfy the condition of the first (non-recursive) case. Hence, the result of the evaluation is always true. Therefore the recursive calls of θ_{times} can in fact be replaced by true which yields the following non-recursive version of θ_{times}.

$$
\begin{array}{l}
\mathit{function}\ \theta_{\mathsf{times}}(x, y : \mathsf{nat}) : \mathsf{bool} \Leftarrow \\
\quad \mathit{if}\ \ x = \mathsf{0}\ \ \mathit{then}\ \ \mathsf{true} \\
\qquad\qquad \mathit{else}\ \ (\ \mathit{if}\ \ \mathsf{even}(x)\ \ \mathit{then}\ \ \mathsf{true} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{else}\ \ \mathsf{true}\ )
\end{array}
$$

In general, let R be a set of recursive θ_f-cases with *results* of the form $\theta_f(t^*)$ and let ω be a boolean value (either true or false). Our simplification procedure replaces the recursive calls in the R-cases by the boolean value ω, if for each case in R, evaluation of the result $\theta_f(t^*)$ either leads to a non-recursive case with the result ω or to a recursive case from R.

Let Ψ be the set of all *conditions* from non-recursive cases with the result ω and of all conditions from R-cases. Then one has to show that the arguments t^* satisfy one of the conditions $\varphi \in \Psi$. In other words $\varphi[x^*/t^*]$ must be valid (where $[x^*/t^*]$ denotes the substitution of the formal parameters x^* by the terms t^*). Hence, for each case in R with the *condition* ψ the following *recursion elimination formula* has to be verified.

$$\psi \rightarrow \bigvee_{\varphi \in \Psi} \varphi[x^*/t^*]$$

In our example, the set R contains both recursive cases. So for the first recursive call one has to prove that under its condition $x \neq \mathsf{0} \wedge \mathsf{even}(x)$, the recursive arguments $\mathsf{mean}(x, \mathsf{0})$ and $\mathsf{double}(y)$ either satisfy the conditions of an R-case or of the first non-recursive case.

$$
\begin{array}{lll}
x \neq \mathsf{0} \wedge \mathsf{even}(x) \ \rightarrow & \mathsf{mean}(x, \mathsf{0}) = \mathsf{0} \vee & \qquad (11) \\
& (\mathsf{mean}(x, \mathsf{0}) \neq \mathsf{0} \wedge \ \ \mathsf{even}(\mathsf{mean}(x, \mathsf{0}))) \vee & \\
& (\mathsf{mean}(x, \mathsf{0}) \neq \mathsf{0} \wedge \neg\mathsf{even}(\mathsf{mean}(x, \mathsf{0}))) &
\end{array}
$$

A similar recursion elimination formula is also obtained for θ_{times}'s second recursive call.

Again, for the automated verification of such formulas an (induction) theorem prover is used. In fact, the recursion elimination formulas in our example are tautologies that can already be verified by propositional reasoning only.

5.4. Case Elimination

In the last simplification step one tries to replace conditionals by their results. More precisely, regard a conditional of the form "*if* φ_1 *then* φ_2 *else* φ_3" that will only be evaluated under a condition ψ. Now the simplification procedure tries to replace this conditional by the result φ_2. For that purpose the procedure has to check whether under the appropriate premises, φ_2 is equal to the result in the *else*-case of the conditional. Hence, it tries to verify the implication

$$\psi \wedge \neg\varphi_1 \;\rightarrow\; \varphi_2 = \varphi_3.$$

Furthermore, it has to be checked whether the condition φ_1 is necessary to ensure termination of φ_2's evaluation. Hence, the simplification procedure also tries to prove the formula

$$\psi \rightarrow \Theta(\varphi_2).$$

If verification of both *case elimination formulas* succeeds, then the conditional is replaced by φ_2. Otherwise, simplification of the conditional into φ_3 is tried. For that purpose the case elimination formulas $\psi \wedge \varphi_1 \rightarrow \varphi_2 = \varphi_3$ and $\psi \rightarrow \Theta(\varphi_3)$ have to be proved.

In our example, first the conditional "*if* $\mathsf{even}(x)$ *then* true *else* true" is replaced by true after verification of the case elimination formulas $x \neq 0 \wedge \neg\mathsf{even}(x) \rightarrow \mathsf{true} = \mathsf{true}$ and $x \neq 0 \rightarrow \mathsf{true}$. Second, the resulting conditional "*if* $x \neq 0$ *then* true *else* true" is replaced by true since $x = 0 \rightarrow \mathsf{true} = \mathsf{true}$ and true can easily be proved. In this way we obtain the final version of θ_{times}.

$$\mathit{function}\ \theta_{\mathsf{times}}(x, y : \mathsf{nat}) : \mathsf{bool} \Leftarrow \quad \mathsf{true}$$

Using the above techniques this trivial algorithm for θ_{times} has been constructed which states that times is indeed total. In general, our simplification procedure eases further automated reasoning about termination predicates significantly and it also enhances the readability of the termination predicate algorithms.

Summing up, the procedure for simplification of termination predicate algorithms performs the following steps.

(S1) Application of induction lemmata
(S2) Subsumption elimination
(S3) Recursion elimination
(S4) Case elimination

The simplification procedure does not affect the soundness of the transformed termination predicates. So if an algorithm θ'_f is obtained from a termination predicate θ_f by simplification, then θ'_f is also a termination predicate for f.

Theorem 4 *Let the algorithm θ'_f be obtained from an algorithm θ_f by applying the simplification steps (S1)–(S4). Then for all data objects q^*, $\theta'_f(q^*)$ terminates iff $\theta_f(q^*)$ terminates and if $\theta'_f(q^*) = \mathsf{true}$, then $\theta_f(q^*) = \mathsf{true}$, too.*

Proof. For (S1), (S2), and (S4), the soundness follows from the truth of the applied induction lemmata, subsumption formulas, and case elimination formulas. The soundness of recursion elimination is shown in [30]. □

The simplification procedure for termination predicates works *automatically* and it proved successful on numerous examples [6]. It is based on methods for the synthesis of induction lemmata [7,13,14,17,32] and it uses an induction theorem prover to verify the subsumption, recursion elimination, and case elimination formulas (which often is a simple task).

6. Termination Analysis for Imperative Programs

Although *imperative* languages are extensively used in practice, up to now there have been very few attempts to automate termination analysis for imperative programs. However, methods for the automatic translation of imperative programs into functional ones are well known and can be found in several textbooks on functional programming. Therefore, a straightforward approach for automated termination proofs of imperative programs is to transform them into corresponding functional programs. If termination of the resulting functions can be proved, then termination of the original imperative program is verified. However, it turns out that in general the existing approaches for termination analysis of functional programs cannot be used for that purpose, because the functions obtained from the translation of imperative programs are often *partial*.

We regard a simple PASCAL-like language with the atomic statements "... := ...", "*if* ... *then* ... *else* ... *fi*", "*while* ... *do* ... *od*" and the compound statement "... ; ..." which all have the usual semantics.

As an example consider the following imperative program for the multiplication of natural numbers. After execution of the program, the value of the variable r is the result of multiplying the initial values of x and z, i.e., $r = x*z$. Therefore the values of x, z, and r are repeatedly changed similar as in the algorithm times, cf. Section 5.

$r := 0;$
while $\neg x \neq 0$
  do *if* $\mathsf{even}(x)$ *then* $y := 0;$
                while $x \neq y$
                  do $x := \mathsf{pred}(x);$
                       $y := \mathsf{succ}(y)$ *od*;
                $z := \mathsf{double}(z)$
              else $x := \mathsf{pred}(x);$
                $r := \mathsf{plus}(z, r)$ *fi* *od*

} sets x to $\frac{x}{2}$

To translate this imperative program into a functional one, every *while*-loop is transformed into a separate "loop-function". For instance, for the *inner* *while*-loop we obtain the function mean from Section 2 that takes the input values of the variables x and y as arguments and returns the output value of x. (Of course, a similar function returning the output value of y could also be constructed.) If the loop-condition $x \neq y$ is satisfied, then mean is called recursively with the new values of x and y. If the loop-condition is not satisfied, then mean returns the value of x. Using the auxiliary function mean, the outer *while*-loop and the whole imperative program are translated into the functions while and multiply, respectively.

function $\mathsf{while}(x, z, r : \mathsf{nat}) : \mathsf{nat} \Leftarrow$
  if $x \neq 0$ *then* (*if* $\mathsf{even}(x)$ *then* $\mathsf{while}(\mathsf{mean}(x, 0), \mathsf{double}(z), r)$
                                else $\mathsf{while}(\mathsf{pred}(x), z, \mathsf{plus}(z, r))$)
       else r

function $\mathsf{multiply}(x, z : \mathsf{nat}) : \mathsf{nat} \Leftarrow \mathsf{while}(x, z, 0)$

In general, each program written in our imperative programming language translates into a program of our first-order functional language. See e.g. [19] for an automation of this translation.

The resulting function multiply is in fact "equivalent" to the original imperative program, as multiply computes the value of r after execution of the program. In particular, for the termination proof of the imperative program it suffices to show termination of the function multiply.

Note that although the original imperative program is terminating, in general the auxiliary functions resulting from this translation are *partial*. The reason is that in imperative programs, termination of *while*-loops often depends on their contexts. For instance, in our example the inner *while*-loop is only entered with an *even* input x. However, this restriction on the value

of x is no longer present in the function mean. Therefore multiply is totally terminating, but the auxiliary function $\mathsf{mean}(x, y)$ is only terminating if x is greater than or equal to y and if $x - y$ is even.

With our method, termination of multiply can easily be verified. The synthesis of a termination predicate for mean has already been illustrated in Section 3. For the function while, our method generates a termination predicate similar to θ_{times}, cf. Section 5, and the simplification procedure performs exactly the same steps. Hence we finally obtain the following termination predicate algorithm.

$$\mathit{function}\ \theta_{\mathsf{while}}(x, z, r : \mathsf{nat}) : \mathsf{bool} \Leftarrow \quad \mathsf{true}$$

In this way, total termination of the outer *while*-loop is proved. Hence, termination of multiply and thereby, termination of the original imperative program is also verified.

7. Extensions

The synthesis of termination predicates can be directly used for *polymorphic types*, too, where type constants may be parameterized with type variables α. For instance, consider a polymorphic type list_α with the constructors $\mathsf{nil}_\alpha : \mathsf{list}_\alpha$ and $\mathsf{cons}_\alpha : \alpha \times \mathsf{list}_\alpha \to \mathsf{list}_\alpha$ and the selectors $\mathsf{head}_\alpha : \mathsf{list}_\alpha \to \alpha$ and $\mathsf{tail}_\alpha : \mathsf{list}_\alpha \to \mathsf{list}_\alpha$. Then the following algorithm computes the last element of a list containing data objects of type α.

$$\begin{array}{l} \mathit{function}\ \mathsf{last}(l : \mathsf{list}_\alpha) : \alpha \Leftarrow \\ \quad \mathit{if}\ \ l = \mathsf{cons}_\alpha(\mathsf{head}_\alpha(l), \mathsf{nil}_\alpha)\ \ \mathit{then}\ \ \mathsf{head}_\alpha(l) \\ \qquad\qquad\qquad\qquad\qquad\ \ \mathit{else}\ \ \ \mathsf{last}(\mathsf{tail}_\alpha(l)) \end{array}$$

Our method synthesizes the following termination predicate for last where we use the size measure to compare the objects of type list_α, i.e., we have $|\mathsf{nil}_\alpha|_\# = 0$ and $|\mathsf{cons}_\alpha(x, k)|_\# = 1 + |k|_\#$.

$$\begin{array}{l} \mathit{function}\ \theta_{\mathsf{last}}(l : \mathsf{list}_\alpha) : \mathsf{bool} \Leftarrow \\ \quad \mathit{if}\ \ l = \mathsf{cons}_\alpha(\mathsf{head}_\alpha(l), \mathsf{nil}_\alpha)\ \ \mathit{then}\ \ \mathsf{true} \\ \qquad\qquad\qquad\qquad\qquad\ \ \mathit{else}\ \ \ |\mathsf{tail}_\alpha(l)|_\# < |l|_\# \ \wedge\ \theta_{\mathsf{last}}(\mathsf{tail}_\alpha(l)) \end{array}$$

The above algorithm returns true for each non-empty list and thus, it defines exactly the domain of last. In general, no modification of our method is needed to enable the synthesis of termination predicates if polymorphic types are considered as well.

Moreover, our method may also be extended to *mutual recursion* in the same way as suggested in [16] for total termination proofs.

Our technique can be directly generalized to a certain class of *higher-order functions*, viz. functions that may have higher-order arguments but that have first-order results. As an example, consider the following algorithm that applies

a function f to each element of a list l.

$$\begin{array}{l} \mathit{function}\ \mathsf{map}(f : \mathsf{nat} \to \mathsf{nat}, l : \mathsf{list}) : \mathsf{list} \Leftarrow \\ \quad \mathit{if}\ \ l = \mathsf{nil}\ \ \mathit{then}\ \mathsf{nil} \\ \qquad\qquad \mathit{else}\ \ \mathsf{cons}(f(\mathsf{head}(l)), \mathsf{map}(f, \mathsf{tail}(l))) \end{array}$$

A termination predicate algorithm for map also has to check whether the term $f(\mathsf{head}(l))$ terminates if evaluated. For that purpose, the associated termination predicate θ_f is used to compute $\theta_f(\mathsf{head}(l))$.[7] Thus, the higher-order variable f is treated like an auxiliary function when the termination predicate algorithm for map is synthesized and the synthesis rule (ii) is applied to analyze the term $f(\mathsf{head}(l))$. So we obtain the following termination predicate algorithm where instead of $|f, l|_{\#}$ we use $|l|_{\#}$. Thus, first-order termination analysis can be extended to higher-order algorithms by inspecting the decrease of their first-order arguments, cf. also [23].

$$\begin{array}{l} \mathit{function}\ \theta_{\mathsf{map}}(f : \mathsf{nat} \to \mathsf{nat}, l : \mathsf{list}) : \mathsf{bool} \Leftarrow \\ \quad \mathit{if}\ \ l = \mathsf{nil}\ \ \mathit{then}\ \mathsf{true} \\ \qquad\qquad \mathit{else}\ \ \theta_f(\mathsf{head}(l)) \wedge |\mathsf{tail}(l)|_{\#} < |l|_{\#} \wedge \theta_{\mathsf{map}}(f, \mathsf{tail}(l)) \end{array}$$

Again we have obtained an algorithm that defines the exact domain of a partial function (provided that θ_f describes the exact domain of f). The algorithm $\theta_{\mathsf{map}}(f, l)$ returns true iff each element of the list l satisfies θ_f.

In this higher-order extension of our language, we do not allow the use of "λ". Thus, the only higher-order terms are function variables (like f) and function constants (like mean or map). Function variables are handled like auxiliary functions during the computation of termination formulas. Thus, for each term t of non-function type one can compute a termination formula $\Theta(t)$, where Rule (ii) is changed to

$$\Theta(g(t_1,\ldots,t_i,t_{i+1},\ldots,t_n)) :\equiv \Theta(t_{i+1}) \wedge \ldots \wedge \Theta(t_n) \wedge \theta_g(t_1,\ldots,t_n),$$

for functions and function variables g. Here $1,\ldots,i$ denote the higher-order arguments of g, whereas $i+1,\ldots,n$ are the arguments of basic types. Of course, Rule (iv) has to be changed analogously, where instead of $|t_1,\ldots,t_n| < |x_1,\ldots,x_n|$ one only obtains $|t_{i+1},\ldots,t_n| < |x_{i+1},\ldots,x_n|$. Thus, one only inspects the decrease of the first-order arguments.

An extension of our method to a language with "λ" and to functions with higher-order *results* is not as straightforward, because now one would have to extend the concept of termination formulas $\Theta(t)$ to terms t of higher type. Moreover, one does not only need a termination predicate for each function f but one also has to generate termination predicates for the (higher-order) *results* of each function. For example, if f has the type $\mathsf{nat} \to (\mathsf{nat} \to \mathsf{nat})$ then one needs a termination predicate $\theta_f : \mathsf{nat} \to \mathsf{bool}$ for f and a functional

[7] Strictly speaking, "θ_f" is a (higher-order) function that maps f (after its instantiation) to the actual corresponding termination predicate.

$\theta_f^{\text{result}} : \mathsf{nat} \to (\mathsf{nat} \to \mathsf{bool})$ where $\theta_f^{\text{result}}(n)$ is the termination predicate for $f(n)$. An extension of our approach to such higher-order functions is a subject of future work.

8. Conclusion

We have presented a method to determine the domains (resp. non-trivial sub-domains) of partial functions automatically. For that purpose we have *automated* the approach for termination analysis suggested by Manna [22]. Our analysis uses *termination predicates* which represent conditions that imply the termination of the algorithm under consideration. Based on sufficient requirements for termination predicates we have developed a procedure for the automated synthesis of termination predicate algorithms. Subsequently we introduced a procedure for the simplification of these generated termination predicate algorithms which also works automatically. Furthermore, by computing termination predicates for the partial "loop-functions", with our approach it is also possible to perform termination analysis for *imperative programs*. Finally, we have extended our method for polymorphic types and (a certain class of) higher-order functions.

Our method proved successful on numerous algorithms (see Table 1 for some examples to illustrate its power). For each function f in this table the corresponding termination predicate θ_f could be synthesized automatically. In all these examples the synthesized termination predicate is not only sufficient for termination, but it even describes the *exact* domain of the functions.

These examples demonstrate that the procedure of Theorem 3 is able to synthesize sophisticated termination predicate algorithms (e.g. for a quotient algorithm it synthesizes the termination predicate "divides", for a logarithm algorithm it synthesizes a termination predicate that checks if one number is a power of another number, for an algorithm that deletes an element from a list a termination predicate for list membership is synthesized, etc.). By subsequent application of our simplification procedure one usually obtains very simple formulations of the synthesized termination predicate algorithms.

Up to now, the termination behavior of the algorithms in Table 1 could not be analyzed with any other automatic method. Those functions in the table that have the termination predicate true are total, but their algorithms call other non-terminating algorithms. Therefore the existing methods for total termination proofs failed in proving their totality. A detailed description of our experiments can be found in [6].

The presented procedure for the generation of termination predicates works for any given measure function $|.|$. Therefore the procedure can also be combined with methods for the *automated* generation of suitable measure functions (e.g. the one presented in [12,14]), cf. [7,8,17]. In this way we obtained an extremely powerful approach for automated termination analysis of partial functions that performed successfully on a large collection of benchmarks

Table 1
Termination predicates synthesized by our method

Function f	**Term. Pred.** θ_f	**Function** f	**Term. Pred.** θ_f
$\mathsf{minus}(x, y)$	$x \geq y$	$\mathsf{list\_half}(l)$	$\bigwedge_i \mathsf{even}(l_i)$
$\mathsf{half1}(x)$	$\mathsf{even}(x)$	$\mathsf{last}(l)$	$l \neq \mathsf{nil}$
$\mathsf{half2}(x)$	$\mathsf{even}(x) \wedge x \neq 0$	$\mathsf{but\_last}(l)$	$l \neq \mathsf{nil}$
$\mathsf{times}(x, z)$	true	$\mathsf{reverse}(l)$	true
$\mathsf{exp}(x, y)$	true	$\mathsf{list\_min}(l)$	$l \neq \mathsf{nil}$
$\mathsf{quotient1}(x, y)$	$y \neq 0$	$\mathsf{last\_x}(l, x)$	$\mathsf{length}(l) \geq x$
$\mathsf{quotient2}(x, y)$	$y \mid x$	$\mathsf{index}(x, l)$	$x = 0 \vee \mathsf{member}(x, l)$
$\mathsf{mod}(x, y)$	$y \neq 0$	$\mathsf{delete}(x, l)$	$x = 0 \vee \mathsf{member}(x, l)$
$\mathsf{lcm}(x, y)$	$x \neq 0 \wedge y \neq 0$	$\mathsf{sum\_lists}(l, k)$	$\mathsf{length}(l) = \mathsf{length}(k)$
$\mathsf{dual\_log1}(x)$	$x \neq 0$	$\mathsf{nat\_to\_bin}(x, y)$	$y = 2^n$
$\mathsf{dual\_log2}(x)$	$x = 2^n$	$\mathsf{bin\_vec}(x)$	$x \neq 0$
$\mathsf{log1}(x, y)$	$x = 1 \vee$ $x \neq 0 \wedge y \neq 0 \wedge y \neq 1$	$\mathsf{gcd}(x, y)$	$x = 0 \wedge y = 0 \vee$ $x \neq 0 \wedge y \neq 0$
$\mathsf{log2}(x, y)$	$x = 1 \vee$ $x = y^n \wedge x \neq 0 \wedge y \neq 1$	$\mathsf{mean}(x, y)$	$x \geq y \wedge \mathsf{even}(x - y)$
		$\mathsf{list\_minus}(l, y)$	$\bigwedge_i l_i \geq y$

(including all 68 examples from [6,9]). Our method also proved successful for termination analysis of imperative programs. For instance, in 33 of 45 examples from [18] the exact domain could be determined automatically.

Acknowledgments

We thank Christoph Walther and an anonymous referee for helpful comments. This work was supported by the Deutsche Forschungsgemeinschaft under grant no. Wa 652/7-2 as part of the focus program "Deduktion".

References

[1] P. H. Andersen and C. K. Holst, Termination Analysis for Offline Partial Evaluation of a Higher Order Functional Language, in: *Proc. SAS '96*, Lecture Notes in Computer Science, Vol. **1145** (Springer, Berlin, 1996) 67-82.

[2] A. Bouhoula and M. Rusinowitch, Implicit Induction in Conditional Theories, *Journal of Automated Reasoning* **14** (1995) 189-235.

[3] R. S. Boyer and J S. Moore, *A Computational Logic* (Academic Press, 1979).

[4] R. S. Boyer and J S. Moore, The Addition of Bounded Quantification and Partial Functions to A Computational Logic and Its Theorem Prover, *Journal of Automated Reasoning* **4** (1988) 117-172.

[5] J. Brauburger and J. Giesl, Termination Analysis for Partial Functions, in: *Proc. SAS '96*, Lecture Notes in Computer Science, Vol. **1145** (Springer, Berlin, 1996) 113-127.

[6] J. Brauburger and J. Giesl, Termination Analysis for Partial Functions, Technical Report IBN–96–33, TH Darmstadt, Germany, 1996. http://www.inferenzsysteme.informatik.tu-darmstadt.de/~reports/notes/ibn-96-33.ps.gz

[7] J. Brauburger. Automatic Termination Analysis for Partial Functions Using Polynomial Orderings, in: *Proc. SAS '97*, Lecture Notes in Computer Science, Vol. **1302** (Springer, Berlin, 1997) 330-344.

[8] J. Brauburger and J. Giesl. Termination Analysis by Inductive Evaluation, in: *Proc. 15th CADE*, Lecture Notes in Artificial Intelligence, Vol. **1421** (Springer, Berlin, 1998) 254-269.

[9] J. Brauburger and J. Giesl. Termination Analysis by Inductive Evaluation, Technical Report IBN–98–47, TU Darmstadt, Germany, 1998. http://www.inferenzsysteme.informatik.tu-darmstadt.de/~reports/notes/ibn-98-47.ps.gz

[10] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill, The OYSTER-CLAM System, in: *Proc. 10th CADE*, Lecture Notes in Artificial Intelligence, Vol. **449** (Springer, Berlin, 1990) 647-648.

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

[12] J. Giesl, Generating Polynomial Orderings for Termination Proofs, in: *Proc. RTA-95*, Lecture Notes in Computer Science, Vol. **914** (Springer, Berlin, 1995) 426-431.

[13] J. Giesl, Automated Termination Proofs with Measure Functions, in: *Proc. KI-95*, Lecture Notes in Artificial Intelligence, Vol. **981** (Springer, Berlin, 1995) 149-160.

[14] J. Giesl, Termination Analysis for Functional Programs using Term Orderings, in: *Proc. SAS '95*, Lecture Notes in Computer Science, Vol. **983** (Springer, Berlin, 1995) 154-171.

[15] J. Giesl, *Automatisierung von Terminierungsbeweisen für rekursiv definierte Algorithmen*. PhD thesis, TH Darmstadt, DISKI 96 (Infix-Verlag, St. Augustin, 1995).

[16] J. Giesl, Termination of Nested and Mutually Recursive Algorithms, *Journal of Automated Reasoning* **19** (1997) 1-29.

[17] J. Giesl, C. Walther, and J. Brauburger, Termination Analysis for Functional Programs, in: W. Bibel and P. Schmitt, eds., *Automated Deduction – A Basis for Applications*, Vol. 3 (Kluwer Academic Publishers, 1998) 135-164.

[18] D. Gries, *The Science of Programming* (Springer, New York, 1981).

[19] P. Henderson, *Functional Programming* (Prentice-Hall, London, 1980).

[20] D. Hutter and C. Sengler, INKA: The Next Generation, in: *Proc. 13th CADE*, Lecture Notes in Artificial Intelligence, Vol. **1104** (Springer, Berlin, 1996) 288-292.

[21] D. Kapur and H. Zhang, An Overview of Rewrite Rule Laboratory (RRL), *J. Computer Math. Appl.* **29** (1995) 91-114.

[22] Z. Manna, *Mathematical Theory of Computation* (McGraw-Hill, 1974).

[23] F. Nielson and H. R. Nielson, Operational Semantics of Termination Types, *Nordic Journal of Computing* **3** (1996) 144-187.

[24] L. Plümer, *Termination Proofs for Logic Programs*, Lecture Notes in Artificial Intelligence, Vol. **446** (Springer, Berlin, 1990).

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

[26] C. Sengler, Termination of Algorithms over Non-Freely Generated Data Types, in: *Proc. 13th CADE*, Lecture Notes in Artificial Intelligence, Vol. **1104** (Springer, Berlin, 1996) 121-135.

[27] J. Steinbach, Simplification Orderings: History of Results, *Fundamenta Informaticae* **24** (1995) 47-87.

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

[29] C. Walther, Argument-Bounded Algorithms as a Basis for Automated Termination Proofs, in: *Proc. 9th CADE*, Lecture Notes in Computer Science, Vol. **310** (Springer, Berlin, 1988) 602-621.

[30] C. Walther, *Automatisierung von Terminierungsbeweisen* (Vieweg, Braunschweig, 1991).

[31] C. Walther, Mathematical Induction, in: D. M. Gabbay, C. J. Hogger, and J. A. Robinson, eds., *Handbook of Logic in Artificial Intelligence and Logic Programming*, Vol. 2 (Oxford University Press, 1994) 127-231.

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