JÜRGEN GIESL, CHRISTOPH WALTHER, JÜRGEN BRAUBURGER

TERMINATION ANALYSIS FOR FUNCTIONAL PROGRAMS

1. INTRODUCTION

Proving termination is a central problem in software development and formal methods for termination analysis are essential for program verification. However, since the halting problem is undecidable and totality of functions is not even semi-decidable, there is no procedure to prove or disprove the termination of *all* algorithms.

While most work on the automation of termination proofs has been done in the areas of *term rewriting systems* (for surveys see e.g. (Dershowitz, 1987; Steinbach, 1995b)) and of *logic programs* (e.g. (Ullman and van Gelder, 1988; Plümer, 1990; De Schreye and Decorte, 1994)), in this chapter we focus on *functional programs* and we also investigate the application of our methods for termination analysis of *loops* in *imperative programs*.

To prove termination of a functional algorithm, one has to find a well-founded relation such that the arguments in each recursive call are smaller than the corresponding inputs. (A relation $\prec$ is *well founded* iff there is no infinite descending chain $\ldots \prec t_2 \prec t_1$.) A semi-automatic method for termination proofs of LISP functions has been implemented in the NQTHM system of *R. S. Boyer* and *J S. Moore* (1979). For an algorithm $f(x)$ their prover shows that in each recursive call $f(r)$ a given *measure* is decreased. The system uses a *measure function* $|.|$ which maps data objects to natural numbers and it verifies that the number $|r|$ is smaller than the number $|x|$.

For this proof the *system user* has to supply so-called *induction lemmata* of the form $\Delta \rightarrow |r| < |x|$, which assert that certain operations are measure decreasing if some hypotheses Δ are satisfied. For the termination proof of an algorithm $f(x)$ with a recursive call $f(r)$, the system searches among the known induction lemmata for some lemma with the conclusion $|r| < |x|$. Then the prover verifies $b \rightarrow \Delta$, where b is the condition under which the recursive call $f(r)$ is performed.

Of course instead of using induction lemmata, one could also try to prove inequalities like $|r| < |x|$ *directly*. But the advantage of induction lemmata is

Authors' address: Fachbereich Informatik, Technische Universität Darmstadt, Alexanderstr. 10, 64283 Darmstadt, Germany, E-mail: {giesl,walther,brauburger}@informatik.tu-darmstadt.de

that difficult facts are established once and for all (by an *induction* proof). Then by applying induction lemmata, subsequent termination proofs are usually quite simple, i.e. they can often be accomplished without induction.

Boyer and Moore's technique for proving termination is very powerful, as the methods works for *arbitrary measure functions* $|.|$. But its disadvantage is a low degree of automation. Induction lemmata have to be *formulated by the system user* and therefore a human has to find the idea why an algorithm terminates. Moreover, to ensure their soundness, these lemmata have to be verified by the system before use. This verification may be hard, as in general an induction proof is needed.

In this chapter, we present an alternative method for automated termination proofs of algorithms (Section 2). With this method a certain class of induction lemmata can be synthesized *automatically* and the soundness of these induction lemmata is *guaranteed by construction*. So compared to the technique of Boyer and Moore the advantage of our method is a much higher degree of automation.

But a limitation of this method is that it is restricted to *one single fixed* measure function, viz. the so-called *size measure function*, while the system of Boyer and Moore can use arbitrary measure functions for termination proofs. Using the size measure function, data objects t are compared by their *size* $|t|_{\#}$, i.e. lists are compared by their length, trees are compared by the number of their nodes etc. Although this approach is successful for many examples, there are numerous relevant algorithms whose termination proofs require a measure different from *size* and for these algorithms the method of Section 2 must fail.

Therefore, in Section 3 we extend our method in order to handle *arbitrary measure functions*. To determine suitable measures automatically we use approaches from the area of *term rewriting systems* for the generation of well-founded *term orderings*. But unfortunately term orderings cannot be directly used for termination proofs of *functional algorithms* which call other algorithms in the arguments of their recursive calls. The reason is that for termination of term rewriting systems orderings between *terms* are needed, whereas for functional programs orderings between the *evaluated terms* are required. Our method solves this problem and enables *term orderings* to be used for *functional programs*. In this way, we obtain an *automated* method for termination proofs where suitable measure functions and induction lemmata are synthesized by machine. This combines a high degree of automation with the powerful generality of Boyer and Moore's method.

The approaches of Section 2 and 3 aim to prove that an algorithm terminates for *each* input ("*total* termination"). Thus, if the termination proof fails then they cannot find a (sub-)domain where termination is provable. How-

ever, this is necessary for termination proofs of algorithms which call *partial* auxiliary algorithms, i.e. auxiliary algorithms which do not terminate for all inputs. In particular, this problem arises when examining the termination behaviour of loops in imperative programs. Therefore, in Section 4 we extend our techniques to termination analysis of partial functions.

2. Termination Proofs with Argument-Bounded Algorithms

In this section we illustrate a first approach for automated termination proofs using a *fixed* ordering. After an introduction to termination proofs in Section 2.1, Section 2.2 shows how inequalities are proved by the technique of *estimation*. For this technique we need certain knowledge about the algorithms under consideration and in Section 2.3 we discuss how this knowledge can be acquired automatically.

2.1. *Termination Proofs of Functional Programs*

We regard an eager first-order functional language with (free) algebraic data types and pattern matching (where the patterns have to be exhaustive and exclusive)[1]. For example, consider the data type bool with the nullary *constructors* true and false and the data type nat (for naturals) whose objects are built with the constructors 0 and $\mathsf{s} : \mathsf{nat} \to \mathsf{nat}$.

The following algorithms compute predecessor, subtraction, and division (the boolean function lt is the usual "less than" relation on nat).

function $\mathsf{p} : \mathsf{nat} \to \mathsf{nat}$

$$\begin{aligned} \mathsf{p}(0) &= 0 \\ \mathsf{p}(\mathsf{s}(x)) &= x \end{aligned}$$

function $\mathsf{minus} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$

$$\begin{aligned} \mathsf{minus}(x,0) &= x \\ \mathsf{minus}(0,\mathsf{s}(y)) &= 0 \\ \mathsf{minus}(\mathsf{s}(x),\mathsf{s}(y)) &= \mathsf{minus}(x,y) \end{aligned}$$

function $\mathsf{quot} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$

$$\begin{aligned} \mathsf{quot}(x,0) &= 0 \\ \mathsf{quot}(x,\mathsf{s}(y)) &= \mathsf{if}(\mathsf{lt}(x,\mathsf{s}(y)),\ 0,\ \mathsf{s}(\mathsf{quot}(\mathsf{minus}(\mathsf{p}(x),y),\mathsf{s}(y)))) \end{aligned}$$

For each data type s there is a pre-defined conditional $\mathsf{if} : \mathsf{bool} \times s \times s \to s$. These conditionals are the only algorithms with non-eager semantics, i.e. when evaluating $\mathsf{if}(b,t_1,t_2)$, the (boolean) term[2] b is evaluated first and de-

[1] An eager language evaluates the arguments of a function call *before* application (this corresponds to the *call-by-value* parameter passing discipline). The use of pattern matching instead of *selectors* (or *destructors*) has no real impact on the difficulty of the termination proof, but it eases the readability of our presentation.

[2] In the following we often refer to boolean terms as "formulas", where $\neg$, $\wedge$, $\vee$, and $\to$ are pre-defined boolean functions with obvious semantics.

pending on the result of its evaluation either t_1 or t_2 is evaluated afterwards yielding the result of the whole conditional.

To prove *termination* of algorithms one has to show that in each recursive call a certain *measure* is decreased w.r.t. a well-founded relation. For that purpose a *measure function* $|.|$ is used which maps each data object t to a natural number $|t|$. For minus' termination proof we can use the *size* measure $|.|_{\#}$, where the size of an object of type nat is the number it represents (i.e. the number of s-occurrences it contains). So we have $|0|_{\#} = 0$, $|\mathsf{s}(0)|_{\#} = 1$ etc. Termination of the (binary) function minus can now be verified by regarding its first argument only. Hence, we have to show that $|x|_{\#}$ is smaller than $|\mathsf{s}(x)|_{\#}$ for all instantiations of x, i.e. we have to verify the following *termination formula*[3] for minus.

$$|x|_{\#} < |\mathsf{s}(x)|_{\#}$$

So for a function f with several arguments we try to prove termination by comparing the i-th argument of the recursive calls with the i-th input argument (for a fixed i). For every recursive call in a defining equation $f(\ldots t_i \ldots) = \ldots f(\ldots r_i \ldots) \ldots$ we obtain a *termination formula*

$$b \;\rightarrow\; |r_i|_{\#} < |t_i|_{\#} \tag{1.1}$$

where b is the condition under which the recursive call $f(\ldots r_i \ldots)$ is evaluated. For instance, the recursive call of quot is evaluated under the condition $\neg\mathsf{lt}(x, \mathsf{s}(y))$. So, by regarding the first argument of quot, we obtain the termination formula

$$\neg\mathsf{lt}(x, \mathsf{s}(y)) \;\rightarrow\; |\mathsf{minus}(\mathsf{p}(x), y)|_{\#} < |x|_{\#}.$$

The approach of comparing data objects t by their size $|t|_{\#}$ is frequently used in termination proofs. For any data object of type s, we can define an abstract notion of *size* by counting the number of *reflexive*[4] constructors of type s, where substructures are ignored.

For example, let list be a data type with the constructors $\mathsf{empty} : \mathsf{list}$ and $\mathsf{add} : \mathsf{nat} \times \mathsf{list} \rightarrow \mathsf{list}$, where $\mathsf{add}(n, l)$ represents the insertion of the number n in front of the list l. Then we have $|\mathsf{add}(\mathsf{s}(0), \mathsf{empty})|_{\#} = 1$, because this term (of type list) contains only one occurrence of a reflexive list-constructor,

[3] We only regard universally closed formulas of the form $\forall \ldots \varphi$ for verification, where φ is quantifier free and we omit the quantifiers to ease readability.

[4] A function symbol $f : s_1 \times \ldots \times s_n \rightarrow s$ is *reflexive* if its range type s is among its domain types s_i, and otherwise it is *irreflexive*. For instance, 0 is an irreflexive function symbol, whereas s, p, minus, and quot are reflexive.

whereas $|\mathsf{add}(0,\mathsf{add}(0,\mathsf{empty}))|_{\#} = 2$. Now we can define the *size ordering* $\prec_{\#}$ which compares data objects by their size, i.e. $t_1 \prec_{\#} t_2$ iff $|t_1|_{\#} < |t_2|_{\#}$. The *non-strict* size ordering $\preceq_{\#}$ is defined as $t_1 \preceq_{\#} t_2$ iff $|t_1|_{\#} \leq |t_2|_{\#}$. Hence, the termination formulas of minus and quot can be restated as

$$x \prec_{\#} \mathsf{s}(x), \tag{1.2}$$

$$\neg\mathsf{lt}(x,\mathsf{s}(y)) \rightarrow \mathsf{minus}(\mathsf{p}(x),y) \prec_{\#} x. \tag{1.3}$$

2.2. *Proving Inequalities by Estimation*

Termination formula (1.2) of minus holds by definition of the size ordering since $\mathsf{s}(x)$ contains one more occurrence of s than x for all ground instantiations of x. Note that for the termination proof of minus a pair of terms built only with *variables* and *constructors* has to be compared. But for an algorithm like quot which calls other *algorithms* (viz. minus and p), a termination formula like (1.3) is more difficult to verify. The reason is that *size* is defined in terms of the constructors 0 and s only. Hence, we cannot directly compute the size of the data object resulting from $\mathsf{minus}(\mathsf{p}(x),y)$ by evaluation of the algorithms minus and p.

A common verification technique is to use *estimations* for proving such inequalities. Assume that we know the estimations

$$\mathsf{minus}(x,y) \preceq_{\#} x$$
$$\mathsf{p}(x) \preceq_{\#} x$$

which state that minus and p are *1-bounded* operations, i.e. the results of $\mathsf{minus}(x,y)$ and $\mathsf{p}(x)$ are always size-smaller or have the same size as their first argument x.

In general, a function g is *p-bounded* iff its result is always size-smaller than or size-equal to its input x_p on argument position p, i.e. $g(x_1,\ldots,x_n) \preceq_{\#} x_p$ for all ground instantiations of $x_1,\ldots,x_n$, and it is *argument-bounded* iff it is p-bounded for some argument position p. From the 1-boundedness of minus and p we can conclude

$$\mathsf{minus}(\mathsf{p}(x),y) \preceq_{\#} \mathsf{p}(x) \preceq_{\#} x. \tag{1.4}$$

Hence, by transitivity of $\preceq_{\#}$ the *non-strict* version of the termination formula (1.3) is verified. But for the termination proof of quot, $\mathsf{minus}(\mathsf{p}(x),y)$ has to be *strictly* smaller than x under the condition of quot's recursive call.

Assume also that we know *difference predicates* $\Delta^1_{\mathsf{minus}}$ and Δ^1_{p} which indicate whether $\mathsf{minus}(x,y)$ resp. $\mathsf{p}(x)$ are *strictly* smaller than x:

function $\Delta^1_{\mathsf{minus}} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{bool}$		*function* $\Delta^1_{\mathsf{p}} : \mathsf{nat} \to \mathsf{bool}$	
$\Delta^1_{\mathsf{minus}}(x, \mathsf{0})$	$= \mathsf{false}$	$\Delta^1_{\mathsf{p}}(\mathsf{0})$	$= \mathsf{false}$
$\Delta^1_{\mathsf{minus}}(\mathsf{0}, \mathsf{s}(y))$	$= \mathsf{false}$	$\Delta^1_{\mathsf{p}}(\mathsf{s}(x))$	$= \mathsf{true}$
$\Delta^1_{\mathsf{minus}}(\mathsf{s}(x), \mathsf{s}(y))$	$= \mathsf{true}$		

Then the following *induction lemmata* hold for minus and p:

$$\begin{aligned} \Delta^1_{\mathsf{minus}}(x,y) &\leftrightarrow \mathsf{minus}(x,y) \prec_{\#} x, \\ \Delta^1_{\mathsf{p}}(x) &\leftrightarrow \mathsf{p}(x) \prec_{\#} x. \end{aligned}$$

To finish the termination proof of quot we have to establish that under the condition of quot's recursive call at least one of the two inequalities in (1.4) is strict, i.e. that $\mathsf{minus}(\mathsf{p}(x),y) \prec_{\#} x$ holds. Since $\Delta^1_{\mathsf{minus}}(\mathsf{p}(x),y)$ is equivalent to the strictness of the first inequality and $\Delta^1_{\mathsf{p}}(x)$ is equivalent to the strictness of the second inequality,

$$\Delta^1_{\mathsf{minus}}(\mathsf{p}(x),y) \ \vee \ \Delta^1_{\mathsf{p}}(x)$$

is an equivalent requirement for $\mathsf{minus}(\mathsf{p}(x),y) \prec_{\#} x$. Hence, termination of quot is proved by verifying the *termination hypothesis*

$$(1.5) \quad \neg \mathsf{lt}(x, \mathsf{s}(y)) \ \to \ \Delta^1_{\mathsf{minus}}(\mathsf{p}(x),y) \ \vee \ \Delta^1_{\mathsf{p}}(x),$$

i.e. by verifying $x \geq y+1 \ \to \ x-1 \neq 0 \wedge y \neq 0 \ \vee \ x \neq 0$.

In general, each p-bounded function $g : s_1 \times \ldots \times s_n \to s$ is associated with a *difference predicate* $\Delta^p_g : s_1 \times \ldots \times s_n \to \mathsf{bool}$ such that

$$(1.6) \quad \Delta^p_g(x_1, \ldots, x_n) \ \leftrightarrow \ g(x_1, \ldots, x_n) \prec_{\#} x_p$$

holds for all ground instantiations of $x_1, \ldots, x_n$. So if a p-bounded function g is applied to some input $t_1, \ldots, t_n$ and it returns something size-smaller than t_p, then the difference predicate Δ^p_g applied to $t_1, \ldots, t_n$ yields true and it returns false iff g returns something size-equal to t_p.

The notion of argument-bounded functions is the key concept for our formalization of estimation proofs. Suppose that for each $p \in \mathbb{N}$ we know a set Γ_p of p-bounded function symbols and let Γ be the family of all these sets. Then a *decidable* so-called *estimation relation* $\preceq_\Gamma$ on terms can be defined, such that for all terms t_1 and t_n we have

$$(1.7) \quad t_1 \ \preceq_\Gamma \ t_n \ \text{ implies } \ t_1 \preceq_{\#} t_n.$$

The relation $\preceq_\Gamma$ mirrors the proof technique of estimation and it is based on the knowledge about data types and the argument-bounded functions in Γ.

Suppose we know that minus and p are 1-bounded, i.e. $\{\mathsf{minus},\mathsf{p}\} \subseteq \Gamma_1$. Then the non-strict inequality $\mathsf{minus}(\mathsf{p}(x),y) \preceq_\# x$ can be verified directly by estimating minus and p, cf. (1.4). So the general idea to establish $t_1 \preceq_\Gamma t_n$ is to test whether t_n is a subterm of t_1, as e.g. x is a subterm of $\mathsf{minus}(\mathsf{p}(x),y)$, where only subterms in argument-bounded positions p are inspected.

The estimation relation $\preceq_\Gamma$ provides a deductive requirement for the *non-strict* semantic relation $\preceq_\#$, cf. (1.7). But as termination proofs are based on the relation $\prec_\#$, we need a deductive means for the *strict* size ordering $\prec_\#$.

Here, the general idea is to scan an estimation $t_1 \preceq_\Gamma t_2 \preceq_\Gamma \ldots \preceq_\Gamma t_{n-1} \preceq_\Gamma t_n$ step by step. For each estimation step $t_i \preceq_\Gamma t_{i+1}$, where $t_i = g_i(\ldots t_{i+1} \ldots)$ and t_{i+1} is the p_i-th argument of the p_i-bounded function g_i, the corresponding "call" of the difference predicate $\Delta_{g_i}^{p_i}(\ldots t_{i+1} \ldots)$ is collected. From all these "calls", one computes the disjunction

$$\Delta_{g_1}^{p_1}(\ldots t_2 \ldots) \vee \Delta_{g_2}^{p_2}(\ldots t_3 \ldots) \vee \ldots \vee \Delta_{g_{n-1}}^{p_{n-1}}(\ldots t_n \ldots).$$

This formula is called the *difference equivalent* $\Delta_\Gamma(t_1,t_n)$ of t_1 and t_n. Since each literal of $\Delta_\Gamma(t_1,t_n)$ is built with a difference predicate, we have generated an equivalent requirement for $t_1 \prec_\# t_n$, i.e.

$$t_1 \preceq_\Gamma t_n \text{ implies } \Delta_\Gamma(t_1,t_n) \leftrightarrow t_1 \prec_\# t_n.$$

Apart from the estimation rule for argument-bounded functions, we may also define additional estimation rules based on data types. For instance, the estimation step

$$t_p \preceq_\Gamma cons(\ldots t_p \ldots)$$

can be performed for each constructor *cons* with a reflexive argument position p, and the corresponding difference equivalent $\Delta_\Gamma(t_p, cons(\ldots))$ is true. So by this rule we can conclude inequalities like $t_2 \preceq_\Gamma \mathsf{add}(t_1,t_2)$ and $t \preceq_\Gamma \mathsf{s}(t)$ (cf. the termination formula (1.2) of minus). Moreover, for each data type, "$t \preceq_\Gamma t$" can be used as an estimation step where $\Delta_\Gamma(t,t)$ is defined as false.

Using the estimation relation and the difference equivalent, our procedure for automated termination proofs is straightforward. For each recursive call in a defining equation $f(\ldots t \ldots) = \ldots f(\ldots r \ldots) \ldots$ we have to prove a termination formula of the form

$$b \rightarrow r \prec_\# t,$$

where b is the condition under which $f(\ldots r \ldots)$ is evaluated, cf. (1.1). Now we first test whether $r \preceq_\Gamma t$ holds (otherwise the termination proof fails). If $r \preceq_\Gamma t$ can be established, then the *termination hypothesis*

$$b \rightarrow \Delta_\Gamma(r,t)$$

is generated. Subsequently, an *induction theorem proving system* (e.g. one of those described in (Boyer and Moore, 1979; Bundy et al., 1990; Walther, 1994a; Bouhoula and Rusinowitch, 1995; Kapur and Zhang, 1995; Hutter and Sengler, 1996)) is used to verify the generated termination hypotheses. This verification is usually quite simple, i.e. the proof often succeeds using case analysis and propositional reasoning only. If all termination hypotheses can be proved, then the termination formulas hold for each recursion and consequently, the algorithm under consideration terminates.

For example, to verify termination of quot, $\mathsf{minus}(\mathsf{p}(x),y) \preceq_\Gamma x$ is established first. Then the difference equivalent $\Delta_\Gamma(\mathsf{minus}(\mathsf{p}(x),y),x)$ is computed as $\Delta^1_{\mathsf{minus}}(\mathsf{p}(x),y) \vee \Delta^1_{\mathsf{p}}(x)$ and the termination hypothesis (1.5) is generated. Finally, an induction theorem prover is called with this termination hypothesis, which is easily verified by case analysis and symbolic evaluation.

2.3. *Argument-Bounded Functions*

The generation of termination hypotheses is based on the argument-bounded functions in Γ and on their difference predicates. Hence, for an automation of our approach, we have to determine argument-boundedness and to synthesize difference predicates[5] automatically.

To recognize argument-bounded functions we define a decidable *algorithm schema* such that every instance of this schema computes an argument-bounded function. The main idea embodied in this schema is to construct a meta-induction proof to verify that a function is argument-bounded. The defining equations of the difference predicate are generated in parallel to the proof steps of the meta-induction.

For example, to verify that minus is 1-bounded, the right-hand side of minus' first defining equation, viz. x, is compared with the corresponding first input argument x. Since $x \preceq_\Gamma x$ holds, minus is 1-bounded at least for the first case. The corresponding difference equivalent $\Delta_\Gamma(x,x)$ is false, which means that in this case the result is never size-smaller than the input argument x. Therefore, we obtain $\Delta^1_{\mathsf{minus}}(x,\mathsf{0}) = \mathsf{false}$ as the first defining equation for the difference predicate $\Delta^1_{\mathsf{minus}}$.

For the second equation of minus the result 0 has to be compared with the corresponding input 0. We have $\mathsf{0} \preceq_\Gamma \mathsf{0}$ and $\Delta_\Gamma(\mathsf{0},\mathsf{0}) = \mathsf{false}$. Hence, minus is also 1-bounded in this case and we obtain $\Delta^1_{\mathsf{minus}}(\mathsf{0},\mathsf{s}(y)) = \mathsf{false}$ as the second defining equation for the difference predicate. This completes the base case of the meta-induction.

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

Finally the result $\mathsf{minus}(x,y)$ of the third (recursive) equation is compared with $\mathsf{s}(x)$. Because we assume $\mathsf{minus}(x,y) \preceq_\Gamma x$ as the induction hypothesis, it remains to verify $x \preceq_\Gamma \mathsf{s}(x)$. As this obviously holds (where the corresponding difference equivalent $\Delta_\Gamma(x,\mathsf{s}(x))$ is true), minus is 1-bounded for the recursive case, too. Now the difference equivalent $\Delta_\Gamma(\mathsf{minus}(x,y),\mathsf{s}(x))$ is computed as $\Delta^1_{\mathsf{minus}}(x,y) \vee \mathsf{true}$ which is simplified to true (here we may already use $\Delta^1_{\mathsf{minus}}(x,y)$ because we are in an inductive construction). Thus, $\mathsf{minus}(x,y)$ is size-smaller than $\mathsf{s}(x)$, and therefore, the last defining equation of $\Delta^1_{\mathsf{minus}}$ is $\Delta^1_{\mathsf{minus}}(\mathsf{s}(x),\mathsf{s}(y)) = \mathsf{true}$. This completes the step case of the meta-induction. In this way, minus is recognized as a 1-bounded function and the algorithm for $\Delta^1_{\mathsf{minus}}$ as given in Section 2.2 is synthesized automatically.

Note that the above meta-induction is based on the recursions of minus. Hence, the proof for the argument-boundedness of minus is only sound if minus terminates, because otherwise the induction relation used is not well founded. So before proving the termination of quot we must have proved termination of minus. Therefore, in this chapter we always demand that (apart from recursive calls) algorithms only call other algorithms whose termination has been verified *before*, which excludes mutually recursive algorithms (see (Giesl, 1997) for an extension of termination analysis to mutual recursion).

2.4. *Refinements and Summary*

We have presented a method to generate termination hypotheses, such that the truth of these hypotheses implies termination of the algorithm under consideration. The termination hypotheses represent the *idea* why an algorithm terminates. Hence, with our method the *creativity* for finding the right argument for termination has been mechanized for a certain class of algorithms.

Our approach is based on induction lemmata stating that certain operations are argument-bounded w.r.t. the size-ordering. Unlike the method of Boyer and Moore, our method is able to synthesize such induction lemmata automatically. The obtained lemmata are sound by construction, i.e. they do not have to be verified by a theorem prover.

The general procedure to compute the relevant knowledge for termination proofs works as follows. For each new algorithm f we first prove termination. Afterwards, for each argument position p it is tested whether f is p-bounded and if so, a difference predicate Δ^p_f is synthesized. In this way, the knowledge about argument-bounded algorithms is increased gradually by machine.

For functions with several arguments we proved termination by examining just one argument position. However, our approach is generalized in a straightforward way by taking *several* argument positions into account, cf. (Walther, 1994b). Numerous refinements of our method have been developed,

e.g. techniques for the simplification of difference predicates which ease the proof of termination hypotheses considerably. Further details and refinements can be found in (Walther, 1988; Walther, 1991; Walther, 1994b).

Our method works for polymorphic types as well and an adaptation to non-free data types can be found in (Sengler, 1996; Sengler, 1997). Based on our schema for argument-bounded functions, *McAllester* and *Arkoudas* (1996) suggest a programming discipline such that only terminating programs can be defined. Termination analysis can also be extended to languages with lazy evaluation strategy (Panitz and Schmidt-Schauß, 1997) and to higher-order functions by inspecting the decrease of their first-order arguments, cf. (Nielson and Nielson, 1996).

3. Termination Proofs with Term Orderings

The termination proof method just presented is restricted to *one single fixed* measure function. However, there are many useful algorithms which require a measure function $|.|$ different from the *size* measure function $|.|_{\#}$. Therefore, we now extend our approach to *arbitrary* well-founded *term orderings*.

3.1. *Term Orderings and Functional Programs*

Consider the data type tree for binary trees with the constructors nil and cons. The nullary function nil represents leafs and $\mathsf{cons}(t_1, t_2)$ is the tree whose root has the direct subtrees t_1 and t_2. The algorithm flatten linearizes trees such that all left subtrees are leaves, cf. Figure 1.

function $\mathsf{flatten} : \mathsf{tree} \to \mathsf{tree}$

$$\begin{aligned}
\mathsf{flatten}(\mathsf{nil}) &= \mathsf{nil}\\
\mathsf{flatten}(\mathsf{cons}(\mathsf{nil}, y)) &= \mathsf{cons}(\mathsf{nil}, \mathsf{flatten}(y))\\
\mathsf{flatten}(\mathsf{cons}(\mathsf{cons}(u, v), w)) &= \mathsf{flatten}(\mathsf{cons}(u, \mathsf{cons}(v, w)))
\end{aligned}$$

In the third defining equation, the size of the argument is not decreasing in the recursive call, but it remains the same (where the size of a tree is the number of its inner nodes, i.e. the number of occurrences of cons). So the size ordering cannot be used for the termination proof of flatten and therefore the method of Section 2 fails. To prove the termination of flatten we have to find a well-founded ordering on terms (a so-called *term ordering*) which satisfies the termination formulas

$$\begin{aligned}
&(1.8) && y && \prec && \mathsf{cons}(\mathsf{nil}, y),\\
&(1.9) && \mathsf{cons}(u, \mathsf{cons}(v, w)) && \prec && \mathsf{cons}(\mathsf{cons}(u, v), w).
\end{aligned}$$

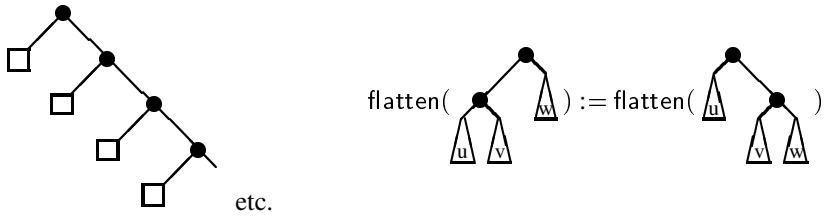

Figure 1. A tree in linear form and rotation of trees as performed by flatten.

Techniques for the automated generation of well-founded term orderings have been developed in the area of *term rewriting systems*. For instance, (1.8) and (1.9) are satisfied by a *polynomial ordering* (Lankford, 1979), where every n-ary function symbol is associated with an n-ary polynomial over the natural numbers. Then a ground term r is smaller than a ground term t with respect to the polynomial ordering iff the number $|r|$ corresponding to r is smaller than the number $|t|$ corresponding to t. As all ground terms are associated with natural numbers, polynomial orderings are well founded.

If nil is associated with the nullary polynomial 0 and $\mathsf{cons}(x,y)$ is associated with the polynomial $1+2x+y$ (i.e. $|\mathsf{cons}(x,y)| = 1+2|x|+|y|$), then $|\mathsf{cons}(\mathsf{nil},y)| = 1+|y|$. As $|y| < 1+|y|$ holds for all natural numbers $|y|$, this polynomial ordering satisfies the first requirement (1.8). Analogously, $|\mathsf{cons}(u,\mathsf{cons}(v,w))| = 2+2|u|+2|v|+|w|$ and $|\mathsf{cons}(\mathsf{cons}(u,v),w)| = 3+4|u|+2|v|+|w|$. As $2+2|u|+2|v|+|w| < 3+4|u|+2|v|+|w|$ is true for all naturals $|u|,|v|,|w|$, the polynomial ordering also satisfies (1.9) and the termination of flatten is proved.

However, in general this naive approach of using term orderings for termination proofs is not sound. For example, the algorithm

$$
\begin{array}{l}
\textit{function}\ \mathsf{f} : \mathsf{tree} \to \mathsf{tree} \\
\quad \mathsf{f}(\mathsf{nil}) \quad\quad\quad = \mathsf{nil} \\
\quad \mathsf{f}(\mathsf{cons}(u,v)) \;= \mathsf{f}(\mathsf{flatten}(\mathsf{cons}(u,v)))
\end{array}
$$

is obviously not terminating. Nevertheless there exists a well-founded term ordering $\prec$ satisfying the termination formula

$$\mathsf{flatten}(\mathsf{cons}(u,v)) \prec \mathsf{cons}(u,v). \tag{1.10}$$

For instance, let $r \prec t$ hold iff the leading function symbol of r is flatten and the leading symbol of t is cons. So the existence of a well-founded term ordering such that terms in the recursive calls are smaller than the corresponding input terms is *not sufficient* for the termination of a functional program.

The reason is that a term with the *defined*[6] function symbol flatten is used in f's recursive call, viz. $\mathsf{flatten}(\mathsf{cons}(u,v))$, whose *evaluation* must be compared with the input $\mathsf{cons}(u,v)$. Thus, (1.10) cannot hold for any well-founded ordering, since $\mathsf{flatten}(\mathsf{cons}(\mathsf{nil},\mathsf{nil}))$ is evaluated to $\mathsf{cons}(\mathsf{nil},\mathsf{nil})$. So a direct use of term orderings for termination proofs is only sound, if the arguments of the recursive calls contain no functions except constructors (as nil and cons), because terms built only with constructors are evaluated already.

A straightforward attempt to enable the use of term orderings for functional algorithms which call other algorithms is the restriction to term orderings which *respect the semantics* of the called algorithms. Under such orderings, *different terms* which denote the *same data object* (like $\mathsf{flatten}(\mathsf{cons}(\mathsf{nil},\mathsf{nil}))$ and $\mathsf{cons}(\mathsf{nil},\mathsf{nil})$) are equivalent.

But in general this restriction is too strong. Consider the following purge-and-sort algorithm sort (i.e. sort also eliminates duplicates).

function $\mathsf{sort} : \mathsf{list} \to \mathsf{list}$
$\mathsf{sort}(x) \;=\; \mathsf{if}(\mathsf{emptyp}(x),\ \mathsf{empty},\ \mathsf{add}(\mathsf{min}(x),\mathsf{sort}(\mathsf{rm}(\mathsf{min}(x),x))))$

This algorithm calls three other algorithms emptyp, min, and rm, where $\mathsf{emptyp}(x)$ checks if x is empty, $\mathsf{min}(x)$ computes the minimum of a non-empty list x, and $\mathsf{rm}(n,x)$ removes *all* occurrences of n from x.

However, none of the orderings typically used in the area of term rewriting systems both respects the semantics of min and rm and makes inputs greater than the corresponding recursive calls of sort. (In fact, most term orderings that are amenable to automation[7] are *simplification orderings* (Dershowitz, 1987; Steinbach, 1995b), i.e. orderings which possess the *subterm property* ($t \prec f(\ldots t \ldots)$). Such orderings do not respect the semantics of the algorithm min, because $\mathsf{min}(\mathsf{add}(\mathsf{0},\mathsf{empty}))$ evaluates to 0, but $\mathsf{min}(\mathsf{add}(\mathsf{0},\mathsf{empty}))$ can never be equivalent to its subterm 0 w.r.t. a simplification ordering.)

Another straightforward attempt 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 kind of (conditional) term rewriting systems with *innermost evaluation* strategy. But due to their special form and this evaluation strategy we may use a different approach for termination proofs of functional programs than it is necessary for term rewriting systems.

[6] Functions whose semantics are determined by algorithms are called *defined*.

[7] There also exist techniques which can orient *every* terminating term rewriting system (e.g. semantical path orderings (Kamin and Levy, 1980), transformation orderings (Bellegarde and Lescanne, 1990), or semantic labelling (Zantema, 1995)). But the disadvantage of these powerful approaches is that up to now there are only few suggestions for their automated generation (Steinbach, 1995a).

For instance, for functional programs it is sufficient to compare the input *arguments* with the *arguments* in the recursive calls, while for term rewriting systems left- and right-hand sides of *all rules* have to be compared (and moreover, the ordering has to be *monotonic*), cf. (Dershowitz, 1987).

Therefore by transforming functional programs into term rewriting systems we impose unnecessarily strong requirements for the termination proof. For instance, no simplification ordering can prove termination of the term rewriting system corresponding to sort, min, and rm, since it contains a rule

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

whose left-hand side is embedded in its right-hand side (and the same holds for the system resulting from quot, minus, and p from Section 2).

3.2. *Elimination of Defined Function Symbols*

In the preceding section we showed that term orderings can be used for termination proofs of algorithms like flatten, where the size ordering (and therefore the approach of Section 2) fails. But on the other hand, the estimation technique of Section 2 proves termination of algorithms like sort which call other algorithms in the arguments of their recursive calls, whereas for such algorithms the direct use of term orderings is unsound. To benefit from the advantages of both approaches, we need a method which combines the handling of such algorithms with the flexibility of arbitrary term orderings (i.e. we need a proper extension of our approach in Section 2). Since the straightforward solutions impose too strong requirements such that termination proofs *often fail*, we now develop a method which overcomes these problems. To prove the termination of sort we have to show the existence of a well-founded ordering $\prec$ (on the *evaluated* terms) satisfying the termination formula

$$(1.11)\quad \neg\mathsf{emptyp}(x) \;\to\; \mathsf{rm}(\mathsf{min}(x),x) \prec x,$$

where $r \prec t$ abbreviates "evaluation of r" $\prec$ "evaluation of t".

As demonstrated, the application of methods for the synthesis of well-founded term orderings is only sound if the termination formulas do not contain defined function symbols (like rm and min). Therefore we develop a calculus to transform termination formulas like (1.11) into formulas *without defined function symbols*.

This calculus transforms a set of termination formulas TF_0 of an algorithm into sets TF_1, TF_2, etc. until we obtain a set of formulas TF_n containing no defined function symbols any more, cf. Figure 2. This transformation is an *abduction* process, i.e. $\mathrm{TF}_{i+1} \models \mathrm{TF}_i$ holds for all i. Hence, if a relation $\prec$ satisfies TF_{i+1}, then it also satisfies TF_i.

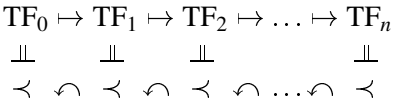

Figure 2. Elimination of defined function symbols from termination formulas.

The formulas in TF_n resulting from the transformation process contain no defined function symbols. Therefore the naive approach of using a term ordering $\prec$ satisfying TF_n is sound. As TF_n implies TF_0, this ordering $\prec$ also satisfies the original termination formulas in TF_0 and therefore the existence of a term ordering satisfying the requirements in TF_n is sufficient for the termination of the algorithm.

The derivation tree in Figure 3 illustrates the transformation of sort's termination formula (1.11). Every node in the tree is transformed into its successors by application of one transformation rule. Leaves 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, every node is implied by the formulas of all its successors. Hence, the formulas at the leaves imply the termination formula at the root of the tree. Thus, the existence of a well-founded term ordering satisfying the requirements at the leaves of the tree in Figure 3 is sufficient for termination of sort. In the following our termination proof method is presented in three steps.

3.2.1. *Estimation and Generalization*
The termination formula (1.11) of sort contains the defined function symbols rm and min. The central idea of our procedure is an *estimation* of defined function symbols by new *free* function symbols (i.e. function symbols like constructors which are not defined by algorithms). Therefore rm is replaced

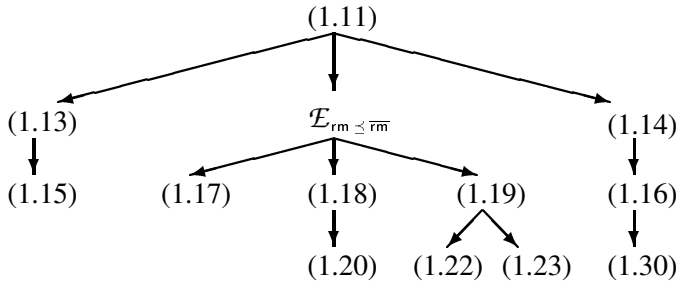

Figure 3. Termination proof of sort.

by a new free 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.

$$\mathsf{rm}(n,x) \preceq \overline{\mathsf{rm}}(n,x). \tag{1.12}$$

Unlike rm, the free function symbol $\overline{\mathsf{rm}}$ has *no fixed semantics* (i.e. there is no algorithm for $\overline{\mathsf{rm}}$) and our aim is to transform the termination formula (1.11) into inequalities which contain free symbols like $\overline{\mathsf{rm}}$, but no defined symbols like rm. If these resulting inequalities are satisfied by a well-founded term ordering, then the termination of sort is proved.

This generalizes the approach of Section 2 where argument-bounded functions were *estimated* by their *arguments* according to the *size ordering*. For example, (1.12) holds for $\overline{\mathsf{rm}}(n,x) := x$ and $\preceq := \preceq_{\#}$, since rm is 2-bounded. However, now we perform estimations without fixing the *upper bounds* and the *ordering used*. Instead we generate constraints and *postpone* the search for suitable upper bounds and orderings satisfying these constraints to the end of the termination proof.

Assume that we know a set of so-called *estimation inequalities* $\mathcal{E}_{\mathsf{rm} \preceq \overline{\mathsf{rm}}}$ (without defined function symbols) which imply (1.12). Then demanding

$$\overline{\mathsf{rm}}(\mathsf{min}(x),x) \preceq x \tag{1.13}$$

and $\mathcal{E}_{\mathsf{rm} \preceq \overline{\mathsf{rm}}}$ implies the *non-strict* version of termination formula (1.11):

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

But to ensure the *strict* inequality (1.11), one of the inequalities (1.13) and $\mathsf{rm}(\mathsf{min}(x),x) \prec \overline{\mathsf{rm}}(\mathsf{min}(x),x)$ must be strict, whenever $\neg\mathsf{emptyp}(x)$ holds.

Assume also that we know a *strictness predicate* $\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}$ which is true iff the result of rm is strictly smaller than the result of $\overline{\mathsf{rm}}$. Then $\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}(\mathsf{min}(x),x)$ 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 formula (1.11) of sort by (1.13), $\mathcal{E}_{\mathsf{rm} \preceq \overline{\mathsf{rm}}}$, and the formula

$$\neg\mathsf{emptyp}(x) \rightarrow \delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}(\mathsf{min}(x),x) \vee \overline{\mathsf{rm}}(\mathsf{min}(x),x) \prec x. \tag{1.14}$$

The transformation of (1.11) into (1.13), $\mathcal{E}_{\mathsf{rm} \preceq \overline{\mathsf{rm}}}$, and (1.14) corresponds to the step from the root of the derivation tree in Figure 3 to its direct successors. This transformation is an abduction step, because (1.13), $\mathcal{E}_{\mathsf{rm} \preceq \overline{\mathsf{rm}}}$, and (1.14) are sufficient for the termination formula (1.11), and it is obtained by the *estimation rule* in Figure 4.

This rule[8] embodies a main principle of our transformation. Similar to Section 2, for a termination formula $b \rightarrow r \prec t$ we ensure that the *non-strict*

[8] We also use an estimation rule for *non-strict* inequalities $g(\ldots) \preceq t$ which only contains the first two consequences of the rule in Figure 4.

$$\frac{b \;\to\; g(r_1,\ldots,r_n) \prec t}{\begin{array}{c}\bar{g}(r_1,\ldots,r_n) \preceq t \\ \mathcal{E}_{g \preceq \bar{g}} \\ b \;\to\; \delta_{g \prec \bar{g}}(r_1,\ldots,r_n) \vee \bar{g}(r_1,\ldots,r_n) \prec t\end{array}}$$

Figure 4. Estimation rule to estimate a defined function symbol g.

unconditional inequality $r \preceq t$ holds. The advantage of omitting the condition b (which contains semantical information) is that defined symbols in $r \preceq t$ can be eliminated by repeated application of *purely syntactical* inference rules (like estimation). A *semantical* rule which considers the condition b is only necessary to guarantee that one of the estimation steps in the derivation is strict, cf. Section 3.2.3.

The resulting formulas (1.13) and (1.14) still contain the defined symbol min. To eliminate min we can again use estimation and replace it by a new free symbol $\overline{\mathsf{min}}$. But apart from estimation there is another (obvious) method to eliminate defined symbols like min, viz. replacing terms like $\mathsf{min}(x)$ in inequalities[9] by fresh variables. In this way, (1.13) and (1.14) are generalized to

$$\overline{\mathsf{rm}}(z,x) \;\preceq\; x, \tag{1.15}$$

$$\neg\mathsf{emptyp}(x) \;\to\; \delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}(\mathsf{min}(x),x) \vee \overline{\mathsf{rm}}(z,x) \prec x. \tag{1.16}$$

Each *generalization* is also an abduction, because if (1.15) resp. (1.16) hold for all z, then (1.13) resp. (1.14) hold as well. As (1.15) contains no defined symbols, this inequality corresponds to a leaf in the derivation tree.

3.2.2. *Estimation Inequalities and Strictness Predicates*

Our estimation technique is based on *estimation inequalities* like $\mathcal{E}_{\mathsf{rm} \preceq \overline{\mathsf{rm}}}$ and on *strictness predicates* like $\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}$ and in this section we show how they are computed. The estimation inequalities $\mathcal{E}_{\mathsf{rm} \preceq \overline{\mathsf{rm}}}$ guarantee that $\overline{\mathsf{rm}}$ is an upper bound for rm, where rm is defined as

$$\begin{array}{l}\textit{function}\ \mathsf{rm} : \mathsf{nat} \times \mathsf{list} \to \mathsf{list} \\ \quad \mathsf{rm}(n, \mathsf{empty}) \;\; = \mathsf{empty} \\ \quad \mathsf{rm}(n, \mathsf{add}(m,y)) = \mathsf{if}(\mathsf{eq}(n,m),\ \mathsf{rm}(n,y),\ \mathsf{add}(m, \mathsf{rm}(n,y)))\,.\end{array}$$

Here, eq computes the equality predicate on naturals.

[9] To perform estimation and generalization repeatedly, these techniques may also be applied to *inequalities* $g(\ldots) \prec t$ occurring in disjunctions of the form $b \to \delta_1(\ldots) \vee \ldots \vee \delta_n(\ldots) \vee g(\ldots) \prec t$. A method to eliminate the defined symbols from the terms $\delta(\ldots)$ is presented in Section 3.2.3.

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

$$\begin{aligned}
&(1.17) && \mathsf{empty} && \preceq && \overline{\mathsf{rm}}(n, \mathsf{empty}),\\
&(1.18) && \mathsf{rm}(n,y) && \preceq && \overline{\mathsf{rm}}(n, \mathsf{add}(m,y)),\\
&(1.19) && \mathsf{add}(m, \mathsf{rm}(n,y)) && \preceq && \overline{\mathsf{rm}}(n, \mathsf{add}(m,y)).
\end{aligned}$$

We cannot define $\mathcal{E}_{\mathsf{rm} \preceq \overline{\mathsf{rm}}} = \{(1.17), (1.18), (1.19)\}$ as (1.18) and (1.19) still contain the defined symbol rm. Defined function symbols occurring in such inequalities have to be eliminated by *estimation* or *generalization* again.

But the problem here is that rm *itself* appears in the inequalities (1.18) and (1.19). Therefore we cannot use the (non-strict version of the) estimation rule in Figure 4 for the estimation of rm, because we do not know the estimation inequalities $\mathcal{E}_{\mathsf{rm} \preceq \overline{\mathsf{rm}}}$ yet.

The construction of *estimation inequalities* corresponds to the problem of proving *argument-boundedness* in Section 2. Therefore, we solve it in a similar way by computing $\mathcal{E}_{\mathsf{rm} \preceq \overline{\mathsf{rm}}}$ *inductively*. The base case of the inductive construction corresponds to rm's non-recursive equation. Inequality (1.17) ensures that in this case $\overline{\mathsf{rm}}$ is an upper bound for rm.

In the step case we have to guarantee that (1.18) and (1.19) hold, i.e. for inputs of the form $(n, \mathsf{add}(m,y))$ the result of rm must be smaller or equal than the result of $\overline{\mathsf{rm}}$. As *induction hypothesis* we assume that this estimation is already correct for the arguments (n,y), i.e. $\mathsf{rm}(n,y) \preceq \overline{\mathsf{rm}}(n,y)$. Therefore

$$(1.20) \quad \overline{\mathsf{rm}}(n,y) \preceq \overline{\mathsf{rm}}(n, \mathsf{add}(m,y))$$

is sufficient for (1.18) and we can replace (1.18) by inequality (1.20) which does not contain defined function symbols.

So to eliminate the defined symbol rm from (1.18) we use the (non-strict) estimation rule (Figure 4) and due to an inductive argument we can omit the second consequence, viz. $\mathcal{E}_{\mathsf{rm} \preceq \overline{\mathsf{rm}}}$, of this inference rule.

For the transformation of (1.19) we again use the induction hypothesis $\mathsf{rm}(n,y) \preceq \overline{\mathsf{rm}}(n,y)$. However, to eliminate the defined symbol rm from (1.19) by estimation, we would need

$$(1.21) \quad \mathsf{add}(m, \mathsf{rm}(n,y)) \preceq \mathsf{add}(m, \overline{\mathsf{rm}}(n,y)).$$

To imply (1.21), in addition to the induction hypothesis 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. we demand the requirement

$$(1.22) \quad u \prec v \;\rightarrow\; \mathsf{add}(m,u) \prec \mathsf{add}(m,v).$$

Now (by the induction hypothesis), (1.19) is implied by (1.22) and

(1.23) $\mathsf{add}(m,\overline{\mathsf{rm}}(n,y)) \preceq \overline{\mathsf{rm}}(n,\mathsf{add}(m,y))$.

The monotonicity problem always appears except when estimating the leading function symbol. To estimate a function within a term at a position π, the rule in Figure 4 is extended by a consequence which demands that the term is monotonic in the position π. Subsequently any defined symbols in monotonicity formulas like (1.22) are eliminated by generalization.

Note that the monotonicity problem results from the consideration of *arbitrary* term orderings. If we restrict ourselves to the size ordering of Section 2, i.e. $\preceq := \preceq_{\#}$, then monotonicity requirements like (1.22) always hold for reflexive arguments of constructors.

Now we have finished our inductive construction of $\mathcal{E}_{\mathsf{rm} \preceq \overline{\mathsf{rm}}}$ (as illustrated in Figure 3) and obtain

$$\begin{aligned}
(1.17)\quad \mathcal{E}_{\mathsf{rm} \preceq \overline{\mathsf{rm}}} \quad = \quad & \{\mathsf{empty} \preceq \overline{\mathsf{rm}}(n,\mathsf{empty}),\\
(1.20)\quad & \overline{\mathsf{rm}}(n,y) \preceq \overline{\mathsf{rm}}(n,\mathsf{add}(m,y)),\\
(1.22)\quad & u \prec v \;\rightarrow\; \mathsf{add}(m,u) \prec \mathsf{add}(m,v),\\
(1.23)\quad & \mathsf{add}(m,\overline{\mathsf{rm}}(n,y)) \preceq \overline{\mathsf{rm}}(n,\mathsf{add}(m,y))\}.
\end{aligned}$$

So in general *estimation inequalities* $\mathcal{E}_{g \preceq \bar{g}}$ are computed as follows.

1. For each result r of the algorithm g we build the formula $r \preceq \bar{g}(\ldots)$.
2. Then the defined symbols in $r \preceq \bar{g}(\ldots)$ are eliminated by estimation or generalization. When g *itself* is estimated, the second consequence ($\mathcal{E}_{g \preceq \bar{g}}$) of the (non-strict) estimation rule is omitted.

The algorithm for the *strictness predicate* $\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}$ is also constructed by induction (corresponding to the construction of *difference predicates* in Section 2). The predicate $\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}(n,x)$ has to return true iff $\mathsf{rm}(n,x)$ is strictly smaller than $\overline{\mathsf{rm}}(n,x)$. By analysis according to the result terms of rm we obtain the following defining equations for $\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}$:

$$\begin{aligned}
& \delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}(n,\mathsf{empty}) \quad = \; \mathsf{empty} \prec \overline{\mathsf{rm}}(n,\mathsf{empty})\\
& \delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}(n,\mathsf{add}(m,y)) \quad = \; \mathsf{if}(\mathsf{eq}(n,m),\\
(1.24)\quad & \qquad \mathsf{rm}(n,y) \prec \overline{\mathsf{rm}}(n,\mathsf{add}(m,y)),\\
(1.25)\quad & \qquad \mathsf{add}(m,\mathsf{rm}(n,y)) \prec \overline{\mathsf{rm}}(n,\mathsf{add}(m,y)))
\end{aligned}$$

However, the inequalities (1.24) and (1.25) still contain the defined function symbol rm. Therefore we eliminate this symbol by estimation again. So inequality (1.24) is transformed into (1.20), $\mathcal{E}_{\mathsf{rm} \preceq \overline{\mathsf{rm}}}$, and

(1.26) $\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}(n,y) \vee \overline{\mathsf{rm}}(n,y) \prec \overline{\mathsf{rm}}(n,\mathsf{add}(m,y))$.

We construct $\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}$ *inductively*, i.e. when defining $\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}(n, \mathsf{add}(m,y))$ we use "$\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}(n,y) = \mathsf{rm}(n,y) \prec \overline{\mathsf{rm}}(n,y)$" as an induction hypothesis. This results in the recursive call $\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}(n,y)$ of (1.26).

Note that (1.20) is already included in $\mathcal{E}_{\mathsf{rm} \preceq \overline{\mathsf{rm}}}$. Hence, as long as $\mathcal{E}_{\mathsf{rm} \preceq \overline{\mathsf{rm}}}$ holds we only have to consider the third consequence of the estimation rule in Figure 4 and replace (1.24) by (1.26). We proceed in the same way for inequality (1.25) and obtain the following algorithm for $\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}$.

$$
\begin{array}{l}
\textit{function}\ \delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}} : \mathsf{nat} \times \mathsf{list} \to \mathsf{bool} \\
\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}(n, \mathsf{empty}) \quad = \mathsf{empty} \prec \overline{\mathsf{rm}}(n, \mathsf{empty}) \\
\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}(n, \mathsf{add}(m,y)) = \mathsf{if}(\mathsf{eq}(n,m), \\
\qquad\qquad\qquad \delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}(n,y) \vee \overline{\mathsf{rm}}(n,y) \prec \overline{\mathsf{rm}}(n, \mathsf{add}(m,y)), \\
\qquad\qquad\qquad \delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}(n,y) \vee \mathsf{add}(m, \overline{\mathsf{rm}}(n,y)) \prec \overline{\mathsf{rm}}(n, \mathsf{add}(m,y)))
\end{array}
$$

So *algorithms* for *strictness predicates* $\delta_{g \prec \bar{g}}$ are constructed as follows.

1. Each result r of the algorithm g is replaced by $r \prec \bar{g}(\ldots)$.
2. Then the defined function symbols in $r \prec \bar{g}(\ldots)$ are eliminated by estimation or generalization. When using estimation all consequences of the estimation rule except the third one are omitted.

3.2.3. *Elimination of Strictness Predicates*

By estimation and generalization of defined function symbols each termination formula $b \to r \prec t$ is transformed into a (possibly empty) set of non-strict inequalities and a formula of the form

$$(1.27) \quad b \to \delta_1(\ldots) \vee \ldots \vee \delta_n(\ldots) \vee r' \prec t.$$

For example, the termination formula (1.11) of sort has been transformed into inequality (1.15), the estimation inequalities $\mathcal{E}_{\mathsf{rm} \preceq \overline{\mathsf{rm}}}$, and the formula

$$(1.16) \quad \neg\mathsf{emptyp}(x) \to \delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}(\mathsf{min}(x), x) \vee \overline{\mathsf{rm}}(z,x) \prec x.$$

The formula (1.27) corresponds to the notion of *termination hypotheses* from Section 2 and the unconditional part of (1.27), viz. $\delta_1(\ldots) \vee \ldots \vee \delta_n(\ldots) \vee r' \prec t$, is the *difference equivalent* $\Delta_\prec(r,t)$ of r and t. Hence, this extends the notion of difference equivalents to arbitrary term orderings $\prec$. Similar to Section 2 we have

$$r \preceq t \ \text{ implies } \ \Delta_\prec(r,t) \leftrightarrow r \prec t.$$

While (1.15) and $\mathcal{E}_{\mathsf{rm} \preceq \overline{\mathsf{rm}}}$ contain no defined symbols, formula (1.16) contains the strictness predicate $\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}$ which is *defined* by an algorithm. To complete

the elimination of defined symbols we now have to eliminate the strictness predicate $\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}$ from requirement (1.16).

For that purpose we choose some of the *inequalities occurring in* (1.16) *and in the algorithm* $\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}$, i.e. inequalities from

$$
\begin{array}{lrcl}
(1.28) & \overline{\mathsf{rm}}(z,x) & \prec & x, \\
(1.29) & \mathsf{empty} & \prec & \overline{\mathsf{rm}}(n,\mathsf{empty}), \\
(1.30) & \overline{\mathsf{rm}}(n,y) & \prec & \overline{\mathsf{rm}}(n,\mathsf{add}(m,y)), \\
(1.31) & \mathsf{add}(m,\overline{\mathsf{rm}}(n,y)) & \prec & \overline{\mathsf{rm}}(n,\mathsf{add}(m,y)).
\end{array}
$$

Our method selects a subset of these inequalities which is sufficient for (1.16), where *small* subsets are preferable in order to minimize the number of resulting constraints. Of course, (1.28) is sufficient for (1.16). However, if (1.16) is transformed into (1.28) then the termination proof of sort fails. The reason is that there exists no well-founded ordering satisfying both $\mathcal{E}_{\mathsf{rm} \preceq \overline{\mathsf{rm}}}$ and (1.28).

But (1.16) is also implied by inequality (1.30) from the second defining equation of $\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}$. Essentially, the reason is that every non-empty list x contains its minimum. Hence when evaluating $\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}(\mathsf{min}(x),x)$, after a finite number of recursive calls $\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}$ is called with a list which begins with its minimum. Then the if-condition of the second defining equation is satisfied and therefore $\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}(\mathsf{min}(x),x)$ returns true if (1.30) is true.

Different to the syntactical inference rules in Section 3.2.1 and 3.2.2 we now had to consider the *semantics* of the strictness predicate $\delta_{\mathsf{rm} \prec \overline{\mathsf{rm}}}$. To eliminate strictness predicates we have to *prove* that certain inequalities (like (1.30)) are sufficient for formulas like (1.16). To perform such proofs automatically an induction theorem proving system is used (similar to the verification of termination hypotheses in Section 2).

So to *eliminate defined strictness predicates* from a formula of the form $b \to \delta_1(\ldots) \vee \ldots \vee \delta_n(\ldots) \vee r' \prec t$ our method proceeds as follows.

1. Let Φ be the set containing $r' \prec t$ and all inequalities from the algorithms $\delta_1, \ldots, \delta_n$ and their auxiliary algorithms.
2. Then the formula is replaced by a minimal subset of Φ that is sufficient for $b \to \delta_1(\ldots) \vee \ldots \vee \delta_n(\ldots) \vee r' \prec t$.

By replacing (1.16) with (1.30) we have finished the transformation of sort's termination formula into inequalities without defined symbols, i.e. we have constructed the derivation tree in Figure 3. To prove sort's termination one now has to find a well-founded term ordering satisfying the constraints (1.15), (1.17), (1.20), (1.22), (1.23), (1.30) at the leaves of the tree:

$$(1.15)\qquad \overline{\mathsf{rm}}(z,x) \preceq x,$$
$$(1.17)\qquad \mathsf{empty} \preceq \overline{\mathsf{rm}}(n,\mathsf{empty}),$$
$$(1.20)\qquad \overline{\mathsf{rm}}(n,y) \preceq \overline{\mathsf{rm}}(n,\mathsf{add}(m,y)),$$
$$(1.22)\qquad u \prec v \;\rightarrow\; \mathsf{add}(m,u) \prec \mathsf{add}(m,v),$$
$$(1.23)\qquad \mathsf{add}(m,\overline{\mathsf{rm}}(n,y)) \preceq \overline{\mathsf{rm}}(n,\mathsf{add}(m,y)),$$
$$(1.30)\qquad \overline{\mathsf{rm}}(n,y) \prec \overline{\mathsf{rm}}(n,\mathsf{add}(m,y)).$$

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

Note that this polynomial ordering is just the *size* ordering used in Section 2. In fact, sort's termination can also be proved with the approach of Section 2. However, the technique just presented is a proper extension of Section 2, i.e. every termination proof with the method of Section 2 can also be performed with the method from this section, but not vice versa. The reason is that the extended method also proves termination for algorithms like flatten where orderings different from *size* are needed. While flatten did not use auxiliary algorithms, our method also handles examples like the following reachability algorithm on directed graphs, which calls another algorithm union in its recursive call and which also requires an ordering different from size.

$$\begin{aligned}
&\textit{function}\ \mathsf{rch} : \mathsf{nat}\times\mathsf{nat}\times\mathsf{graph}\times\mathsf{graph}\to\mathsf{bool}\\
&\quad \mathsf{rch}(x,y,\varepsilon,h) &&= \mathsf{false}\\
&\quad \mathsf{rch}(x,y,\mathsf{edge}(u,v,i),h) &&= \mathsf{if}\,(\mathsf{eq}(x,u),\ \mathsf{if}\,(\mathsf{eq}(y,v),\ \mathsf{true},\ \mathsf{rch}(x,y,i,h)\vee\\
&&&\qquad\qquad\qquad\qquad \mathsf{rch}(v,y,\mathsf{union}(i,h),\varepsilon)),\\
&&&\qquad\qquad \mathsf{rch}(x,y,i,\mathsf{edge}(u,v,h))\,)
\end{aligned}$$

Here, ε (the empty graph) and $\mathsf{edge}(x,y,g)$ (the graph g extended by an edge from x to y) are the constructors of type graph and $\mathsf{rch}(x,y,g,\varepsilon)$ returns true iff there is a path from node x to node y in the directed graph g. Termination of rch can be proved with our method where the resulting constraints are satisfied by a polynomial ordering of degree 2, whereas there is no termination proof using the size ordering. A collection of numerous such algorithms can be found in (Giesl, 1995d).

For the computation of well-founded term orderings which satisfy the derived constraints we use procedures developed in the area of term rewriting systems and computer algebra. For instance, the algorithm of *G. E. Collins* (1975) decides whether there exists a polynomial ordering over the reals of a given degree satisfying a set of constraints. A procedure to generate polynomial orderings using an efficient, incomplete modification of Collins' algorithm is given in (Giesl, 1995a).

3.3. *Refinements and Summary*

We have presented a method for automated termination proofs of functional programs which extends the approach of Section 2 by allowing *arbitrary* term orderings. When using 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 formulas are *generalized* or *estimated* by new free function symbols $\bar{g}$ (Section 3.2.1). To guarantee that $\bar{g}$ is an upper bound for g we demand *estimation inequalities* $\mathcal{E}_{g \preceq \bar{g}}$. By the estimation of a function symbol g we also obtain a formula containing the *strictness predicate* $\delta_{g \prec \bar{g}}$. This predicate indicates whether the result of $\bar{g}$ is *strictly* greater than the result of g (Section 3.2.2). Finally the (defined) strictness predicate has to be eliminated (using an induction theorem prover) (Section 3.2.3).

Our method is easily extended to algorithms with *several* arguments. For that purpose we introduce a new free *tuple* function symbol ν and instead of two *tuples* $(r_1 \dots r_n)$ and $(t_1 \dots t_n)$ we compare the *terms* $\nu(r_1 \dots r_n)$ and $\nu(t_1 \dots t_n)$. So for instance, we demand $\nu(x, y) \prec \nu(\mathsf{s}(x), \mathsf{s}(y))$ to prove termination of the algorithm minus.

The automatic transformation of termination formulas into formulas without defined function symbols is performed in finitely many steps as each transformation rule decreases the number of defined symbols. Therefore derivation trees only contain paths of *finite* length. But our procedure contains two *choice points*. First, defined function symbols can be eliminated by estimation *or* by generalization. Second, for disjunctions like (1.16) there may be more than one subset of inequalities sufficient for the elimination of strictness predicates. So there can be several different derivation trees for one termination formula. But as the number of derivation trees for one termination formula is also *finite* (and *small*), one can backtrack if no well-founded term ordering satisfying the constraints at the leaves can be found. To improve the efficiency of the method, we have also developed heuristics for choosing the "right" derivation tree which have proved successful in practice. (We tested our method on a large data base of examples and for the vast majority, termination could be proved within a few seconds.) For a more detailed description of our approach and further refinements see (Giesl, 1995b; Giesl, 1995c; Giesl, 1995d; Giesl, 1997).

Our method has also been adapted for termination proofs in other areas. In particular, a method for termination proofs of *term rewriting systems* based on the comparison of *arguments* (instead of rules) and on our estimation technique can be found in (Arts and Giesl, 1996; Arts and Giesl, 1997a; Arts and Giesl, 1997b; Arts and Giesl, 1998).

4. Termination Analysis for Partial Functions

A functional program P (without mutual recursion) can be written as a sequence $\langle f_1, \ldots, f_k \rangle$ of algorithms, such that no algorithm f_i is called by an algorithm f_j preceding f_i in the sequence. A *functional program P terminates* iff its last algorithm, viz. the "main" algorithm f_k, terminates for *all* inputs (*total termination*). If all "auxiliary" algorithms f_i with $i < k$ also terminate totally, then termination of P is verified by proving total termination for all functional algorithms $f_1, \ldots, f_k$ step by step, e.g. by using one of the methods developed so far. However, demanding total termination for all "auxiliary" algorithms is much too strong for termination of P, cf. e.g. (Manna, 1974; Boyer and Moore, 1988). One only has to verify total termination of f_k, whereas the "auxiliary" algorithms f_i with $i < k$ only have to terminate for *some* inputs (*partial termination*), viz. the inputs which result from the (direct or indirect) calls of f_i by f_k. In particular, this approach is necessary when extending termination analysis to *imperative* programs, cf. Section 4.1.

To prove partial termination of an algorithm $f : s_1 \times \ldots \times s_n \to s$, a totally terminating algorithm $\theta_f : s_1 \times \ldots \times s_n \to \mathsf{bool}$ (a *termination predicate* for f) is synthesized, such that the evaluation of $\theta_f(x_1, \ldots, x_n)$ to true implies that evaluation of $f(x_1, \ldots, x_n)$ terminates. Then one only has to verify the truth of $\theta_f(t_1, \ldots, t_n)$ for each call $f(t_1, \ldots, t_n)$ which can be executed in the functional program P. So for

function $\mathsf{f}_1 : \mathsf{nat} \to \mathsf{nat}$	*function* $\mathsf{f}_2 : \mathsf{nat} \to \mathsf{nat}$
$\mathsf{f}_1(\mathsf{0}) = \mathsf{f}_1(\mathsf{0})$	$\mathsf{f}_2(\mathsf{0}) = \mathsf{0}$
$\mathsf{f}_1(\mathsf{s}(x)) = \mathsf{0}$	$\mathsf{f}_2(\mathsf{s}(x)) = \mathsf{f}_1(\mathsf{s}(x))$

and $P = \langle \mathsf{f}_1, \mathsf{f}_2 \rangle$, we only have to prove that $\theta_{\mathsf{f}_1}(\mathsf{s}(t))$ evaluates to true for all data objects t, i.e. we only have to prove partial termination of f_1. This is sufficient for the total termination of the main function f_2 since $\mathsf{f}_1(\mathsf{0})$ is never called when executing the functional program P.

Considered in this framework, our methods for proving total termination from Section 2 and 3 aim to synthesize termination predicates such that $\theta_f(t_1, \ldots, t_n)$ holds for *all* $t_1, \ldots, t_n$. Therefore, in order to obtain termination predicates for algorithms which do not terminate totally, further developments of our termination methods are required.

4.1. *Termination Analysis for Imperative Programs*

A straightforward approach to prove termination of imperative programs is to transform them into functional ones. See e.g. (Henderson, 1980) for an automated translation. If termination of the resulting functions can be proved,

then termination of the original imperative program is verified. As an example consider the following imperative program which adds the multiplication of x's and y's initial values to r, i.e. $r := x * y + r$. For that purpose, the program uses the total auxiliary functions even, plus, and double.

$$
\begin{array}{lll}
\textit{while} & \neg\mathsf{eq}(x,0) & \\
\textit{do} & \textit{if } \mathsf{even}(x) \textit{ then} & z := 0; \\
 & & \textit{while } \neg\mathsf{eq}(x,z) \\
 & & \quad \textit{do } x := \mathsf{p}(x); \\
 & & \qquad z := \mathsf{s}(z) \quad \textit{od}; \\
 & & y := \mathsf{double}(y) \\
 & \textit{else} & x := \mathsf{p}(x); \\
 & & r := \mathsf{plus}(r,y) \quad \textit{fi} \quad \textit{od}
\end{array}
\qquad \Big\} \text{ sets } x \text{ to } \tfrac{x}{2}
$$

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 following function half which takes the input values of the variables x and z as arguments and returns the output value of x. (Of course, a similar function returning the output value of z could also be constructed if required.) If the loop-condition is satisfied, i.e. if $\mathsf{eq}(x,z)$ is false, then half is called recursively with the new values of x and z. Otherwise, half returns the value of x.

$$
\begin{array}{l}
\textit{function } \mathsf{half} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat} \\
\quad \mathsf{half}(x,z) \;=\; \mathsf{if}(\,\mathsf{eq}(x,z),\; x,\; \mathsf{half}(\mathsf{p}(x),\mathsf{s}(z))\,)
\end{array}
$$

The algorithm $\mathsf{half}(x,z)$ computes the arithmetical mean $\frac{x+z}{2}$ of two numbers if it terminates. Using the auxiliary function half, the whole imperative program is translated into the function times.

$$
\begin{array}{l}
\textit{function } \mathsf{times} : \mathsf{nat} \times \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat} \\
\mathsf{times}(x,y,r) = \mathsf{if}(\,\mathsf{eq}(x,0),\; r,\; \mathsf{if}(\,\mathsf{even}(x), \mathsf{times}(\mathsf{half}(x,0), \mathsf{double}(y), r), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{times}(\mathsf{p}(x), y, \mathsf{plus}(r,y))\,)\,)
\end{array}
$$

Obviously, the function times is equivalent to the original imperative program and in particular, termination of times is equivalent to the termination of the imperative program. Note that in general the auxiliary functions resulting from such a translation are *partial* even if the original imperative program terminates totally. The reason is that in imperative programs, termination of *while*-loops often depends on their contexts, i.e. on the preconditions that hold before entering the *while*-loop.

For instance, in our example the inner *while*-loop is only entered with an *even* input x. However, this restriction on x is not present in the function half. Therefore times terminates totally, but its auxiliary function half is partial, as

$\mathsf{half}(x,z)$ only terminates iff x is not smaller than z and if $x-z$ is even. Hence, to prove (total) termination of times one needs a method for termination analysis of *partial* functions.

4.2. *Generation of Termination Predicates*

In this section we present a procedure for the synthesis of termination predicates. Our aim is to generate termination predicates "as weak as possible", i.e. predicates which return true as often as possible, but of course in general this goal cannot be reached as the domains of functions are undecidable. Recall that there are two requirements for termination predicates θ_f.

(a) θ_f is *partially correct*, i.e. if $\theta_f(t_1,\ldots,t_n)$ evaluates to true, then the evaluation of $f(t_1,\ldots,t_n)$ terminates.
(b) θ_f *terminates totally*.

For instance, to satisfy requirement (a) a termination predicate θ_{half} for the algorithm half from Section 4.1 can be generated by replacing the result of the non-recursive case by true and by replacing the recursive call of half by a corresponding recursive call of θ_{half}.

function $\theta_{\mathsf{half}} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{bool}$
$\theta_{\mathsf{half}}(x,z) \;=\; \mathsf{if}(\mathsf{eq}(x,z),\ \mathsf{true},\ \theta_{\mathsf{half}}(\mathsf{p}(x),\mathsf{s}(z)))$

This algorithm returns true iff it terminates. As the recursions of θ_{half} correspond to the recursions of half, the algorithm half terminates whenever θ_{half} yields true. Hence, the above algorithm for θ_{half} is *partially correct*.

However, the above algorithm does not satisfy requirement (b), i.e. it is not totally terminating. In other words, there is no well-founded ordering $\prec$ satisfying the termination formula

$$\neg\mathsf{eq}(x,z) \;\to\; \mathsf{p}(x) \prec x$$

of this algorithm. The central idea for the transformation of the above algorithm into a totally terminating one is to choose some well-founded ordering $\prec$ and to enter the recursive call $\theta_{\mathsf{half}}(\mathsf{p}(x),\mathsf{s}(z))$ only for those inputs x and z where x is greater than the corresponding argument $\mathsf{p}(x)$ in the recursive call and to return false for all other inputs.

A successful heuristic for the choice of $\prec$ is to use a well-founded ordering which satisfies at least the *non-strict* version of the termination formula $\mathsf{p}(x) \preceq x$ (and which satisfies the strict inequality $\mathsf{p}(x) \prec x$ "as often as possible"). To determine such an ordering we use the methods of Section 2 or 3. For example, by estimation of the defined symbol p, the inequality $\mathsf{p}(x) \preceq x$ is transformed into the constraints $\overline{\mathsf{p}}(x) \preceq x$ and $\mathcal{E}_{\mathsf{p} \trianglelefteq \overline{\mathsf{p}}}$ which are satisfied by

the size ordering (where $\overline{\mathsf{p}}(x)$ is associated with the polynomial x). Hence, we demand that $\theta_{\mathsf{half}}(x,z)$ may only enter its recursive call if $\mathsf{p}(x) \prec_{\#} x$ holds.

Now the methods of Section 2 or 3 are used to compute an equivalent requirement for $\mathsf{p}(x) \prec_{\#} x$, viz. the *difference equivalent* $\Delta_{\prec_{\#}}(\mathsf{p}(x),x)$ which corresponds to $\neg\mathsf{eq}(x,\mathsf{0})$. In this way we obtain the following algorithm for the termination predicate of half.

function $\theta_{\mathsf{half}} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{bool}$
$\theta_{\mathsf{half}}(x,z) \;=\; \mathsf{if}(\,\mathsf{eq}(x,z),\ \mathsf{true},\ \mathsf{if}(\,\mathsf{eq}(x,\mathsf{0}),\ \mathsf{false},\ \theta_{\mathsf{half}}(\mathsf{p}(x),\mathsf{s}(z))\,)\,)$

The incorporation of the difference equivalent guarantees that the evaluation of $\theta_{\mathsf{half}}(x,z)$ can only lead to a recursive call $\theta_{\mathsf{half}}(\mathsf{p}(x),\mathsf{s}(z))$ if $\mathsf{p}(x)$ is smaller than the input x.

The above algorithm defines a termination predicate for half, i.e. θ_{half} computes a *total* function and the truth of $\theta_{\mathsf{half}}(\ldots)$ is sufficient for the termination of $\mathsf{half}(\ldots)$. In fact, $\theta_{\mathsf{half}}(x,z)$ is true iff x is greater than or equal to z and if $x-z$ is even. Hence, in this example we have even generated the weakest possible termination predicate, i.e. θ_{half} returns true not only for a subset but for *all* elements of the domain of half.

To automate the above synthesis of termination predicates we associate a boolean term $\Theta(t)$ with each term t such that evaluation of $\Theta(t)$ always terminates and evaluation of t terminates whenever $\Theta(t)$ evaluates to true[10]. $\Theta(t)$ is called a *termination condition* for t and its definition is based on the termination predicates:

(i) $\Theta(x) :\equiv \mathsf{true}$, for variables x,
(ii) $\Theta(\mathsf{if}(b,r_1,r_2)) :\equiv \Theta(b) \mathbin{\&} \mathsf{if}(b,\,\Theta(r_1),\,\Theta(r_2))$,
(iii) $\Theta(g(r_1 \ldots r_n)) :\equiv \Theta(r_1) \mathbin{\&} \ldots \mathbin{\&} \Theta(r_n) \mathbin{\&} \theta_g(r_1 \ldots r_n)$, for functions $g \neq \mathsf{if}$,

where "$t_1 \& t_2$" abbreviates "$\mathsf{if}(t_1,t_2,\mathsf{false})$" and "$t_1 \& t_2 \& \ldots$" abbreviates "$t_1 \& (t_2 \& \ldots)$" to ease readability.

For example, the termination condition $\Theta(f(g(h(t))))$ is computed as $\Theta(t) \mathbin{\&} \theta_h(t) \mathbin{\&} \theta_g(h(t)) \mathbin{\&} \theta_f(g(h(t)))$, because due to our eager language in this term h and g are evaluated before evaluating f. Now if $f(t_1,\ldots,t_n) = r$ is a defining equation of f, then our aim is to use $\theta_f(t_1,\ldots,t_n) = \Theta(r)$ as a defining equation for the corresponding termination predicate.

However, for total termination of θ_f the arguments of recursive calls have to be smaller than the corresponding inputs. Therefore, for recursive calls in a defining equation $f(\ldots t_i \ldots) = \ldots f(r_1 \ldots r_i \ldots r_n) \ldots$ we add a requirement to the termination condition of $f(r_1 \ldots r_i \ldots r_n)$ which ensures $r_i \prec t_i$ (for some

[10] More precisely, this implication holds for each substitution σ of t's variables by data objects. For all such σ, evaluation of $\sigma(\Theta(t))$ is terminating and $\sigma(\Theta(t)) = \mathsf{true}$ implies that the evaluation of $\sigma(t)$ is also terminating.

fixed i). To this end, instead of (iii) we use

(iv) $\Theta(f(r_{1\ldots}r_n)) :\equiv \Theta(r_1)\wedge\ldots\wedge\Theta(r_n)\wedge\Delta_{\prec}(r_i,t_i)\wedge\theta_f(r_{1\ldots}r_n)$

when computing the termination condition for a result term of f.

So for constructors and functions f whose total termination has been verified we define $\theta_f(x_1,\ldots,x_n) :\equiv \mathsf{true}$. For all other functions f we use the following construction principle for termination predicates.

From each defining equation $f(t_1,\ldots,t_n) = r$ of f construct a defining equation $\theta_f(t_1,\ldots,t_n) = \Theta(r)$ for θ_f.

This construction of termination predicate algorithms can be automated directly. For instance, θ_{half} was built according to this procedure (where we simplified expressions like $\mathsf{if}(\mathsf{true},t,\ldots)$ to t).

Now termination of the imperative program from Section 4.1 can be verified automatically. For the function times, our method synthesizes

$$\begin{array}{l}\textit{function}\ \theta_{\mathsf{times}} : \mathsf{nat}\times\mathsf{nat}\times\mathsf{nat}\to\mathsf{bool}\\ \theta_{\mathsf{times}}(x,y,r) = \mathsf{if}(\mathsf{eq}(x,0),\\ \qquad\qquad \mathsf{true},\\ \qquad\qquad \mathsf{if}(\mathsf{even}(x),\mathsf{if}(\theta_{\mathsf{half}}(x,0),\theta_{\mathsf{times}}(\mathsf{half}(x,0),\mathsf{double}(y),r),\mathsf{false}),\\ \qquad\qquad\qquad \theta_{\mathsf{times}}(\mathsf{p}(x),y,\mathsf{plus}(r,y))))).\end{array}$$

In the above algorithm, the difference equivalents $\Delta_{\prec_{\#}}(\mathsf{half}(x,0),x)$ and $\Delta_{\prec_{\#}}(\mathsf{p}(x),x)$ (both of which correspond to $\neg\mathsf{eq}(x,0)$) have been simplified to true, because they are called under the condition $\neg\mathsf{eq}(x,0)$.

Subsequently, the conjecture $\theta_{\mathsf{times}}(x,y,r) = \mathsf{true}$ can be shown by an induction proof. In this way, total termination of times and thereby termination of the original imperative program is verified.

4.3. *Refinements and Summary*

To analyze the termination of an algorithm f, a *termination predicate* θ_f is synthesized, which represents a sufficient condition for f's termination. For this synthesis, the methods of Section 2 or 3 are used to generate a well-founded ordering satisfying all non-strict termination formulas and as many strict termination formulas as possible. (If the ordering satisfies *all* termination formulas, then total termination of f is proved. So the approaches of Section 2 and 3 are a special case of this termination analysis procedure.)

Our approach is successfully applied for termination analysis of *imperative programs*. Such a program can be translated into an equivalent functional program, where each *while*-loop corresponds to a (partially terminating) func-

tional algorithm. Now the termination predicates for these partial "loop functions" can be used to prove termination of the whole imperative program.

A refinement of our method such that *several* arguments of an algorithm are considered is obtained by using *tuple* symbols, cf. Section 3.3. Moreover, to ease subsequent reasoning we developed several techniques to *simplify* the generated termination predicates. Further details can be found in (Brauburger and Giesl, 1996; Brauburger, 1997).

5. CONCLUSION

We have presented methods for a fully automated termination analysis of functional programs. We started with a technique to prove termination by estimating argument-bounded functions w.r.t. the size ordering. This approach has been implemented in the induction theorem prover INKA (Biundo et al., 1986; Hutter and Sengler, 1996) and it proved successful on many examples. A collection of 60 such algorithms (including classical sorting algorithms and algorithms for standard arithmetical operations) can be found in (Walther, 1991). Moreover, this method also proves termination of almost all examples from the data base of (Boyer and Moore, 1979) automatically (where one algorithm (greatest.factor) must be slightly modified). However, for three algorithms of this data base (viz. normalize, gopher, samefringe) this technique fails, as they terminate with a well-founded ordering different from size.

Hence, we generalized our first approach to allow an automatic synthesis of other well-founded relations. The resulting method is the most powerful technique for automated termination proofs developed so far, because in previous approaches the orderings used were either fixed or had to be provided by the user. We also implemented this method within the INKA prover (using a procedure for the automated generation of polynomial orderings (Giesl, 1995a)) and it performed successfully on a large collection of benchmarks (including all 82 algorithms from (Boyer and Moore, 1979) and all 60 examples from (Giesl, 1995d)). This collection contains many practically relevant examples from different areas of computer science (e.g. algorithms on numbers, lists, graphs, trees, strings, terms etc.) and several well-known challenge problems from the literature (such as *McCarthy*'s f_91 function).

Finally, we showed that our approach can also be used for partial functions by synthesizing termination predicates which approximate the domains of such functions. See (Brauburger and Giesl, 1996) for a collection of 32 examples which demonstrate that for many algorithms this approach is able to generate sophisticated termination predicates (which are not only sufficient for termination, but which even describe the *exact* domains of the functions).

ACKNOWLEDGEMENTS

This work was supported by the *Schwerpunktprogramm Deduktion* of the *Deutsche Forschungsgemeinschaft* under grants Wa 652/7-1,2.

REFERENCES

Arts, T. and J. Giesl: 1996, 'Termination of Constructor Systems'. In: *Proc. RTA '96*. New Brunswick, NJ, pp. 63–77. LNCS 1103.

Arts, T. and J. Giesl: 1997a, 'Automatically Proving Termination Where Simplification Orderings Fail'. In: *TAPSOFT '97*. Lille, France, pp. 261–272. LNCS 1214.

Arts, T. and J. Giesl: 1997b, 'Proving Innermost Normalisation Automatically'. In: *Proc. RTA '97*. Sitges, Spain, pp. 157–171. LNCS 1232.

Arts, T. and J. Giesl: 1998, 'Modularity of Termination Using Dependency Pairs'. In: *Proc. RTA '98*. Tsukuba, Japan. LNCS.

Bellegarde, F. and P. Lescanne: 1990, 'Termination by Completion'. *Applicable Algebra in Engineering, Communication and Computing* **1**, 79–96.

Biundo, S., B. Hummel, D. Hutter, and C. Walther: 1986, 'The Karlsruhe Induction Theorem Proving System'. In: *Proc. CADE-8*. Oxford, pp. 672–674. LNCS 230.

Bouhoula, A. and M. Rusinowitch: 1995, 'Implicit Induction in Conditional Theories'. *Journal of Automated Reasoning* **14**, 189–235.

Boyer, R. S. and J. S. Moore: 1979, *A Computational Logic*. Academic Press.

Boyer, R. S. and J. S. Moore: 1988, 'The Addition of Bounded Quantification and Partial Functions to A Computational Logic and Its Theorem Prover'. *Journal of Automated Reasoning* **4**, 117–172.

Brauburger, J. and J. Giesl: 1996, 'Termination Analysis for Partial Functions'. In: *Proc. SAS '96*. Aachen, Germany, pp. 113–127. LNCS 1302. Extended version available as Report IBN 96/33, TU Darmstadt. http://www.inferenzsysteme.informatik.tu-darmstadt.de/~reports/notes/ibn-96-33.ps.

Brauburger, J.: 1997, 'Automatic Termination Analysis for Partial Functions Using Polynomial Orderings'. In: *SAS '97*. Paris, France, pp. 330–344. LNCS 1145.

Bundy, A., F. van Harmelen, C. Horn, and A. Smaill: 1990, 'The OYSTER-CLAM System'. In: *Proc. CADE-10*. Kaiserslautern, Germany, pp. 647–648. LNAI 449.

Collins, G. E.: 1975, 'Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition'. In: *Proc. 2nd GI Conf. on Automata Theory and Formal Languages*. Kaiserslautern, Germany, pp. 134–183. LNCS 33.

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

Dershowitz, N.: 1987, 'Termination of Rewriting'. *J. Symb. Comp* **3**, 69–115.

Giesl, J.: 1995a, 'Generating Polynomial Orderings for Termination Proofs'. In: *Proc. RTA '95*. Kaiserslautern, Germany, pp. 426–431. LNCS 914.

Giesl, J.: 1995b, 'Automated Termination Proofs with Measure Functions'. In: *Proc. 19th Ann. German Conf. on AI*. Bielefeld, Germany, pp. 149–160. LNAI 981.

Giesl, J.: 1995c, 'Termination Analysis for Functional Programs using Term Orderings'. In: *Proc. SAS '95*. Glasgow, Scotland, pp. 154–171. LNCS 983.

Giesl, J.: 1995d, *Automatisierung von Terminierungsbeweisen für rekursiv definierte Algorithmen*. Doctoral Dissertation, Infix-Verlag, Sankt Augustin, Germany.

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

Henderson, P.: 1980, *Functional Programming*. Prentice-Hall, London.

Hutter, D. and C. Sengler: 1996, 'INKA: The Next Generation'. In: *Proc. CADE-13*. New Brunswick, NJ, pp. 288–292. LNAI 1104.

Kamin, S. and J.-J. Levy: 1980, 'Two Generalizations of the Recursive Path Ordering'. Unpublished Note, Dept. of Computer Sc., Univ. Illinois, Urbana, IL.

Kapur, D. and H. Zhang: 1995, 'An Overview of Rewrite Rule Laboratory (RRL)'. *J. Computer Math. Appl.* **29**, 91–114.

Lankford, D. S.: 1979, 'On Proving Term Rewriting Systems are Noetherian'. Technical Report Memo MTP-3, Math. Dept., Louisiana Tech. Univ., Ruston, LA.

Manna, Z.: 1974, *Mathematical Theory of Computation*. McGraw-Hill.

McAllester, D. and K. Arkoudas: 1996, 'Walther Recursion'. In: *Proc. CADE-13*. New Brunswick, NJ, pp. 643–657. LNAI 1104.

Nielson, F. and H. R. Nielson: 1996, 'Operational Semantics of Termination Types'. *Nordic Journal of Computing* **3**, 144–187.

Panitz, S. E. and M. Schmidt-Schauß: 1997, 'TEA: Automatically Proving Termination of Programs in a Non-Strict Higher-Order Functional Language'. In: *Proc. SAS '97*. Paris, France, pp. 345–360. LNCS 1145.

Plümer, L.: 1990, *Termination Proofs for Logic Programs*. Springer. LNAI 446.

Sengler, C.: 1996, 'Termination of Algorithms over Non-Freely Generated Data Types'. In: *Proc. CADE-13*. New Brunswick, NJ, pp. 121–135. LNAI 1104.

Sengler, C.: 1997, *Induction on Non-Freely Generated Data Types*. Doctoral Dissertation, Infix-Verlag, Sankt Augustin, Germany.

Steinbach, J.: 1995a, 'Automatic Termination Proofs with Transformation Orderings'. In: *Proc. RTA '95*. Kaiserslautern, Germany, pp. 11–25. LNCS 914.

Steinbach, J.: 1995b, 'Simplification Orderings: History of Results'. *Fundamenta Informaticae* **24**, 47–87.

Ullman, J. D. and A. van Gelder: 1988, 'Efficient Tests for Top-Down Termination of Logical Rules'. *Journal of the ACM* **35**, 345–373.

Walther, C.: 1988, 'Argument–Bounded Algorithms as a Basis for Automated Termination Proofs'. In: *Proc. CADE-9*. Argonne, IL, pp. 602–621. LNCS 310.

Walther, C.: 1991, *Automatisierung von Terminierungsbeweisen*. Vieweg, Germany.

Walther, C.: 1994a, '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.

Walther, C.: 1994b, 'On Proving the Termination of Algorithms by Machine'. *Artificial Intelligence* **71**, 101–157.

Zantema, H.: 1995, 'Termination of Term Rewriting by Semantic Labelling'. *Fundamenta Informaticae* **24**, 89–105.