ELSEVIER

Artificial Intelligence 71 (1994) 101–157

Artificial Intelligence

On proving the termination of algorithms by machine☆

Christoph Walther*

Institut für Programm- und Informationssysteme, Fachbereich Informatik, Technische Hochschule Darmstadt, Alexanderstraße 10, D 64283 Darmstadt, Germany

Revised October 1993

Abstract

Proving the termination of a recursively defined algorithm requires a certain creativity of the (human or automated) reasoner for inventing a hypothesis whose truth implies that the algorithm terminates. We present a reasoning method for simulating this kind of creativity by machine. The proposed method works automatically, i.e. without any human support. We show, (1) how a termination hypothesis for an algorithm is synthesized by machine, (2) which knowledge about algorithms is required for an automated synthesis, and (3) how this knowledge is computed. Our method solves the problem for a relevant class of algorithms, including classical sorting algorithms and algorithms for standard arithmetical operations, which are given in a pure functional notation. The soundness of the method is proved and several examples are presented for illustrating the performance of the proposal. The method has been implemented and proved successful in practice.

1. Introduction

A central problem in the development of correct software is to verify that algorithms always terminate, provided the intended operations have decidable domains. Non-terminating algorithms compute partial operations, hence machine resources are wasted if a given input is not in the domain of the implemented operation. Also manpower is wasted with the debugging of those algorithms and the frustration caused by non-terminating programs is a common experience of programmers and computer scientists. Therefore techniques to verify termination

☆ This work was supported in part by the *Sonderforschungsbereich 314* "Künstliche Intelligenz und Wissensbasierte Systeme" of the *Deutsche Forschungsgemeinschaft* while the author was affiliated with the Universität Karlsruhe. A preliminary version of this work was presented at the *9th International Conference on Automated Deduction*, Argonne, IL, May 1988.

* E-mail: walther@inferenzsysteme.informatik.th-darmstadt.de.

SSDI 0004-3702(93)E0088-4

are of considerable interest, but since the halting problem is the "classical" undecidable problem there is no procedure which proves or disproves the termination of all algorithms.

If we have an idea why an algorithm terminates, we can use an automated theorem prover for verification. But *finding* the right argument which implies termination usually requires some *creativity* and it is our aim to simulate this kind of creativity by machine. Hence we are concerned here essentially with theorem *discovery* rather than with theorem *proving*.

The termination of a recursively defined algorithm is proved by invention of a *well-founded order relation* for the algorithm and subsequent verification that the arguments in each recursive call are smaller—in the sense of the invented order—than the initially given input. The basic approach is to invent a *termination function*, also called *convergence function* or *bound function*, which decreases the arguments in each recursive call in the sense of some *known* well-founded relation. However, finding a successful termination function is the crucial step in this approach and this is the creativity we intend to mechanize.

This paper is concerned with the automation of termination proofs for a certain class of algorithms. We consider only algorithms which terminate according to well-founded relations based on the so-called *size order*. This order compares data objects by their *size*, e.g. stacks are compared by their depth, lists by their length, trees by the number of their nodes, etc. Suppose, for instance, that we want to prove the termination of a sorting algorithm for lists of natural numbers. This data structure can be defined in our notation by

structure *empty add*(*head* : *number tail* : *list*) : *list* ,

where *number* (standing for the natural numbers) denotes another data structure defined elsewhere. The symbols *empty* and *add* are the *constructors* of *list*, i.e. each *list* equals *empty* or else can be constructed by applying *add* to elements of the data structures *number* and *list*. The symbols *head* and *tail* are the *selectors* of *add* and serve as kinds of inverse operations to the constructor, yielding the first element of a non-empty list and the list with the first element removed respectively.

As an example of an algorithm in our notation we define an algorithm *remove* which eliminates all occurrences of a *number n* from a *list x*:

function *remove*(*n* : *number x* : *list*) : *list* $\Leftarrow$

if $x = empty$ **then** x

if $x \neq empty \wedge head(x) = n$ **then** $remove(n\ tail(x))$

if $x \neq empty \wedge head(x) \neq n$ **then** $add(head(x)\ remove(n\ tail(x)))$.

Given another algorithm **function** *minimum*(*x* : *list*) : *number* $\Leftarrow \cdots$ which returns a minimal element of a non-empty *list x* (in the sense of some order relation for *number* defined elsewhere), we define a sorting algorithm for *list* by

function $sort(x : list) : list \Leftarrow$

if $x = empty$ **then** $empty$

if $x \neq empty$ **then**

$add(minimum(x)\ sort(remove(minimum(x)\ x)))$.[1]

To verify the termination of *sort* a typical argumentation would read: "We have to find a well-founded relation $<_{\mathcal{R}}$ such that $remove(minimum(x)\ x) <_{\mathcal{R}} x$ whenever $x \neq empty$ holds. Our first observation is that $remove(n\ x)$ returns x or a *list* which is shorter than x, in symbols

$$remove(n\ x) \leqslant_{\#} x\ , \tag{1.1}$$

hence $<_{\#}$ (standing for "shorter than") could be the well-founded relation we are looking for. Now searching for a condition which implies $remove(n\ x) <_{\#} x$ we find that this requirement is satisfied iff n is a member of x. So if we define *list*-membership by

function $member(n : number\ x : list) : bool \Leftarrow$

if $x = empty$ **then** $false$

if $x \neq empty \wedge head(x) = n$ **then** $true$

if $x \neq empty \wedge head(x) \neq n$ **then** $member(n\ tail(x))$, (1.2)

we can prove that

$$\forall n : number\ \forall x : list \quad member(n\ x) \leftrightarrow remove(n\ x) <_{\#} x\ . \tag{1.3}$$

Our next observation is that each non-empty *list* contains its minimum, i.e. we can also prove

$$\forall x : list \quad x \neq empty \rightarrow member(minimum(x)\ x)\ . \tag{1.4}$$

From (1.4) and (1.3) we obtain finally

$$\forall x : list \quad x \neq empty \rightarrow remove(minimum(x)\ x) <_{\#} x \tag{1.5}$$

and the termination of *sort* is proved".[2]

Our termination proof procedure mechanizes this chain of reasoning in the spirit of the generate-and-test paradigm: Analyzing algorithms such as *remove*, a machine *recognizes* properties such as (1.1) and *synthesizes* algorithms as (1.2) such that properties as (1.3) hold. Then *termination hypotheses* such as (1.4) are

---

[1] This algorithm also "purges" a list, i.e. multiple occurrences of list elements are eliminated. We could have defined a "real' sorting algorithm using *delete* [2] instead of *remove* but we prefer this version of *sort* because we need *remove* for subsequent illustrations. It is easily seen that our technique works for *delete* in the same way as it does for *remove*.

[2] One may observe that the definition of *minimum* is irrelevant for the termination of *sort* as long as formula (1.4) holds. Hence *sort* also terminates if e.g. *minimum* returns the last element of a non-empty *list* (in which case *sort* reverses a purged list), or if $minimum(x)$ returns $head(x)$ (in which case *sort* computes the purged list).

computed which are sufficient for the termination of an algorithm such as *sort*. Finally an *induction theorem prover* is used as a "tester" for verifying the generated termination hypotheses and this completes the termination proof. Since we do not need special theorem proving abilities we only discuss the "generate" mode of our technique and assume the availability of an induction theorem prover.

The remainder of this paper is organized as follows: The environment in which we define and use data structures and algorithms is settled in Section 2 and termination of algorithms is discussed in Section 3. Section 4 shows how we prove inequalities by the technique of estimation: We define the *estimation calculus*, i.e. a calculus which provides a deductive notion for the (semantical) size order. There we assume the availability of certain knowledge about algorithms which is relevant for a termination proof, such as (1.1), (1.2) and (1.3) in the example above. Using the estimation calculus, termination hypotheses for algorithms such as (1.4) can be generated by machine as shown in Section 5. Section 6 is concerned with an automatic acquisition of the knowledge which is necessary for the generation of termination hypotheses: We present a method for recognizing whether an algorithm is *argument-bounded*, i.e. whether the size of an algorithm's result is always smaller than or equal to the size of one of its input arguments. With this technique, for instance, *remove* is recognized as argument-bounded and a fact such as (1.1) is found by machine. Then we show how a so-called *difference algorithm* can be synthesized for each argument-bounded algorithm. Such an algorithm returns *true* for a list of input arguments, if and only if the given argument-bounded algorithm (applied to the same input arguments) returns a result *strictly* smaller than one of its arguments. We also show how each difference algorithm is optimized. For our example *member* is synthesized as the optimized difference algorithm for *remove* such that a property like (1.3) holds. Shortcomings of our proposal are discussed in Section 7, and Section 8 presents related work. A collection of sorting algorithms whose termination has been automatically proved by an implementation of the method is given in Appendix A.

2. Formal preliminaries

2.1. Data structures and algorithms

Suppose we have a collection of data structures and algorithms operating on them. For instance, we may have a data structure *number* given as

$$\textbf{structure}\ 0\ succ(pred : number) : number \tag{2.1}$$

that denotes the set of natural numbers with successor function *succ*, predecessor function *pred* and 0 for zero, and we may have an algorithm

$$\begin{aligned}&\textbf{function}\ plus(x,y : number) : number \Leftarrow\\ &\quad \textbf{if}\ x = 0\ \textbf{then}\ y\\ &\quad \textbf{if}\ x \neq 0\ \textbf{then}\ succ(plus(pred(x)\ y))\,,\end{aligned} \tag{2.2}$$

which computes addition. The domains our algorithms are operating on are given as sets of *constructor ground terms*, e.g. the data structure *number* defines a set $\mathcal{T}(\Sigma^c)_{number} = \{0, succ(0), succ(succ(0)), \ldots\}$ of constructor ground terms of sort *number* to represent the set of natural numbers. This means that 0 and *succ* are used as the *constructors* of the data structure *number*. Each argument position of a constructor is associated with a so-called *selector* (or *destructor*). Here we have *pred* as the (only) selector of *succ*, i.e. *pred* denotes the function which maps a *number* to a *number* such that $pred(succ(q)) = q$ is satisfied. As another example we have already seen the data structure *list*

$$\textbf{structure}\ empty\ add(head : number\ tail : list) : list\,, \tag{2.3}$$

which defines the set $\mathcal{T}(\Sigma^c)_{list} = \{empty,\ add(0\ empty),\ add(succ(0)\ empty),\ add(0\ add(succ(0)\ empty)), \ldots\}$ of constructor ground terms of sort *list* such that $head(add(n\ k)) = n$ and $tail(add(n\ k)) = k$ is satisfied. We may also define a data structure *bool* by

$$\textbf{structure}\ true\ false : bool\,, \tag{2.4}$$

to define predicates like, for instance, the "less-than" order on numbers by

$$\begin{aligned}&\textbf{function}\ lt(x, y : number) : bool \Leftarrow\\ &\quad \textbf{if}\ y = 0\ \textbf{then}\ false\\ &\quad \textbf{if}\ y \neq 0\ \wedge\ x = 0\ \textbf{then}\ true\\ &\quad \textbf{if}\ y \neq 0\ \wedge\ x \neq 0\ \textbf{then}\ lt(pred(x)\ pred(y))\,.\end{aligned} \tag{2.5}$$

Subsequently we write boolean-valued functions like predicates, e.g. we use $lt(x\ y)$ as an abbreviation for $lt(x\ y) = true$ and $\neg lt(x\ y)$ abbreviates $lt(x\ y) = false$.

We always demand that an algorithm f is *well-formed*, i.e. each function symbol used in the definition of f either is introduced before by a data structure or another algorithm or otherwise is the new symbol f in which case the algorithm is *recursively defined*. Now assume that the conditions of the cases in an algorithm exclude each other, as for instance the conditions of the algorithm *lt*, viz. $y = 0$, $y \neq 0\ \wedge\ x = 0$ and $y \neq 0 \wedge x \neq 0$, do. We call these algorithms *deterministic* and by this requirement we can think of an interpreter I for the algorithms which implements the language in the following way: For each *ground term* given as input (i.e. a term without variable symbols), the interpreter *evaluates* the term by applying the algorithms which are "called" in the term to the evaluation of their arguments until no further applications are possible. If, for instance, $plus(succ(0)\ succ(0))$ is provided as input, *plus* is applied to $(succ(0)\ succ(0))$ yielding $succ(plus(0\ succ(0)))$, then *plus* is applied to $(0\ succ(0))$ yielding $succ(0)$, and therefore $succ(succ(0))$ is the final result of the evaluation.

Next assume that all algorithms in the collection *terminate*, as *plus* and *lt* do, and therefore the evaluation of each ground term halts after finitely many steps with a ground term as result. Then we may view the interpreter I as a *total* mapping $I : \mathcal{T}(\Sigma) \rightarrow \mathcal{T}(\Sigma)$ from the set $\mathcal{T}(\Sigma)$ of ground terms (built with the function symbols Σ introduced by the collection) to itself.

Finally assume that each algorithm is *case-complete*, i.e. the case analysis of an algorithm is complete in the sense that for each n-tuple of constructor ground terms given as an input to an algorithm with n arguments, *at least* one of the case conditions is satisfied for this input. The algorithm *lt* is case-complete, for instance, because either $y = 0$, $y \neq 0 \wedge x = 0$ or $y \neq 0 \wedge x \neq 0$ is true for each pair of constructor terms of sort *number* substituted for x and y. Now we may view the interpreter I as a total mapping $I : \mathcal{T}(\Sigma) \rightarrow \mathcal{T}(\Sigma^c)$ from the set $\mathcal{T}(\Sigma)$ of ground terms to the set $\mathcal{T}(\Sigma^c)$ of *constructor* ground terms. This is because by termination each evaluation ends with a non-recursive case, by case-completeness the condition of this case is satisfied and well-formedness then guarantees that a constructor ground term is returned.

We may also evaluate terms containing *variables* by using *variable bindings* of form $[\![x^*/q^*]\!]$, where x^* is a list of distinct variables and q^* is a list of constructor ground terms. We write, for instance, $I[\![n, m/succ(0), 0]\!]\ (plus(n\ succ(m))) = succ(succ(0))$ to indicate that the interpreter I evaluates $plus(n\ succ(m))$ under the variable binding $[\![n, m/succ(0), 0]\!]$ to $succ(succ(0))$.

2.2. Admissible specifications and theories

Statements about the data structures and algorithms can be formulated as formulas of a many-sorted first-order language $\mathcal{F}(\Sigma, \mathcal{V})$. The names of the data structures, e.g. *number*, *list* and *bool*, are used as *sort symbols* of our many-sorted language and we assume a set of variable symbols $\mathcal{V}_s$ for each sort symbol s. The function symbols introduced by the data structures and algorithms, e.g. *succ*, *pred*, *plus*, *lt*, ..., define the signature Σ from which the set of *terms* $\mathcal{T}(\Sigma, \mathcal{V})$ is constructed, where $\mathcal{V}$ is the set of all variable symbols. The only predicate symbols are TRUE and FALSE (for truth and falsity) and $=$ which is a binary predicate denoting equality. Using TRUE, FALSE and equations of the form $t_1 = t_2$ as atomic formulas, connectives like $\rightarrow$, $\wedge$, $\vee$, etc. and the quantifiers $\forall$ and $\exists$, the set $\mathcal{F}(\Sigma, \mathcal{V})$ of many-sorted first-order formulas is defined as usual. We write $x \in \mathcal{V}_s$ to denote that x is a variable of sort s, and if w is a finite and non-empty sequence of sort symbols, i.e. $w = s_1 \ldots s_n$, then $\mathcal{V}_w$ stands for the cartesian product $\mathcal{V}_{s_1} \times \cdots \times \mathcal{V}_{s_n}$ and $\mathcal{T}(\Sigma^c)_w$ abbreviates $\mathcal{T}(\Sigma^c)_{s_1} \times \cdots \times \mathcal{T}(\Sigma^c)_{s_n}$. If we write "$x^* \in \mathcal{V}_w$", it is implicitly assumed that x^* consists of *distinct* variables.

A collection of data structures and well-formed algorithms is called an *admissible specification* iff each algorithm in the collection terminates, is deterministic and is case-complete. So for each admissible specification there is exactly one interpreter I such that each ground term t is evaluated by I to a constructor ground term $I(t)$ as the result of the evaluation and we may use I for defining an *interpretation* $\mathcal{I}$ for the formulas in $\mathcal{F}(\Sigma, \mathcal{V})$.

The universe of $\mathcal{I}$ is the set $\mathcal{T}(\Sigma^c)$ of constructor ground terms and $\mathcal{I}$ assigns each function symbol $f \in \Sigma$ the mapping computed by the interpreter I. We write $\mathcal{I} \models \varphi$ if $\mathcal{I}$ satisfies the formula φ where $\mathcal{I} \models \forall x^* : w\ \varphi$ iff $\mathcal{I}[\![x^*/q^*]\!] \models \varphi$ for all n-tuples of *constructor* ground terms $q^* \in \mathcal{T}(\Sigma^c)_w$ which can be substituted for

the variables in $x^* \in \mathcal{V}_w$. The meaning of TRUE and FALSE, of the connectives and of the equality sign is as usual. $\mathcal{I}$ is a *model* of a formula φ (or a set of formulas Φ) iff $\mathcal{I} \models \varphi$ (or $\mathcal{I} \models \Phi$, i.e. $\mathcal{I} \models \varphi$ for all $\varphi \in \Phi$). We call $\mathcal{I}$ a *standard interpretation* for S iff $\mathcal{I}$ is obtained from the interpreter of an admissible specification S as just defined.

A statement φ about the data structures and algorithms in an admissible specification is *true* iff $\mathcal{I} \models \varphi$. The set of all true statements is given by the *theory* $Th(\mathcal{I})$ of $\mathcal{I}$, i.e. the set of all closed first-order formulas φ such that $\mathcal{I} \models \varphi$. Since there is only one standard interpretation $\mathcal{I}$ for an admissible specification S we may also associate the *specification* S with a theory $Th(S) := Th(\mathcal{I})$. The theory $Th(S)$ is the set of all true first-order formulas about the expressions in S and we say that an admissible specification *specifies a theory*. For instance, if S consists only of the data structure *number* and the algorithm *plus*, then $Th(S)$ is the set of all true statements about 0, *pred*, *succ* and *plus*, e.g. commutativity and associativity of *plus*, cancellation properties of *plus*, etc. We therefore write

$$[\forall x, y : number\ plus(x\ y) = plus(y\ x)] \in Th(S)\,,$$
$$[\forall x, y, z : number\ plus(x\ plus(y\ z)) = plus(plus(x\ y)\ z)] \in Th(S)\,,$$

but

$$[\forall x : number\ x = plus(x\ x)] \notin Th(S)\,,$$

etc.

An admissible specification S_{n+1} can be obtained by extending an admissible specification S_n with a new data structure or a new algorithm, where we always start with the *initial* (admissible) specification S_0 which consists only of the data structure *bool*. On extending S_n by a new data structure only trivial syntactic features have to be tested, e.g. that the function symbols for the constructors and selectors are "new" with respect to the symbols already used in S_n. On extension by an algorithm, well-formedness, case-completeness, determinism and termination have to be verified. The test for well-formedness is trivial. The tests for case-completeness and determinism can be avoided by an *if–then–else* conditional, or otherwise an induction theorem prover (cf. Section 2.3) is used for verification. The termination test is the subject of the subsequent developments.

2.3. Theorem proving

To verify a statement φ about the expressions in an admissible specification, i.e. to test whether $\varphi \in Th(S)$, some deduction is needed. Therefore each data structure and each algorithm of the specification is *translated* into a set of formulas yielding the set of *axioms* $\Phi(S)$ of the specification S. Then a (sound) first-order calculus is used to infer the statement φ from the axioms $\Phi(S)$.

The axioms are obtained by a *uniform* translation process. Each data structure s of the specification is translated into a set REP_s of *representation formulas* for s

which define properties of the constructors and selectors. For instance, the data structure *list* yields the axioms

$$\forall k : list\ \forall n : number \quad empty \neq add(n\ k)\ , \tag{2.6}$$

$$\forall k_1, k_2 : list\ \forall n_1, n_2 : number \quad add(n_1\ k_1) = add(n_2\ k_2) \rightarrow n_1 = n_2 \wedge k_1 = k_2\ , \tag{2.7}$$

$$\forall k : list \quad k = empty \vee k = add(head(k)\ tail(k))\ , \tag{2.8}$$

$$\forall k : list\ \forall n : number \quad head(add(n\ k)) = n\ , \tag{2.9}$$

$$\forall k : list\ \forall n : number \quad tail(add(n\ k)) = k\ . \tag{2.10}$$

However, using only (2.6)–(2.10) nothing is said about what will happen if *head* and *tail* are applied to *empty*. Since selectors essentially denote *partial* mappings defined only on arguments built with their constructors, we are free to associate *head*(*empty*) (and *tail*(*empty*)) with any constructor ground term of sort *number* (and *list* respectively). But we also need to make such assignments because otherwise *head*(*empty*) and *tail*(*empty*) cannot be evaluated to *constructor* ground terms. For avoiding this situation we associate with each data structure s a so-called *witness term* ∇s. The witness term ∇s is obtained as the application of the leftmost *irreflexive* constructor in the definition of the data structure s to the witness terms of its argument sorts.[3] Using this notion, we define $sel(cons(\ldots)) = \nabla s$ for each selector *sel* and each constructor *cons* such that *sel* does not belong to the constructor *cons*. Consequently we demand that each data structure s provides one irreflexive constructor at least. But this must be demanded anyway because otherwise the set $\mathcal{T}(\Sigma^c)_s$ of constructor ground terms of sort s would be empty. Here we have 0 as witness term of *number*, i.e. $\nabla number = 0$, and consequently

$$head(empty) = 0 \tag{2.11}$$

is obtained. We also find $\nabla list = empty$ and therefore

$$tail(empty) = empty\ . \tag{2.12}$$

The formulas (2.6)–(2.12) constitute the set REP_{list} of representation formulas for *list*, and we define $\mathrm{REP}_{list} \subset \Phi(S)$.

Also algorithms are translated into axioms: Each algorithm

[3] A function symbol is called *irreflexive* iff its rangesort is different from all its domainsorts, and otherwise it is called *reflexive*. For instance, each constant symbol, e.g. 0 or *empty*, is irreflexive (simply because there are no domainsorts for it) and *head* is an irreflexive (non-constant) function symbol. The function symbols *pred*, *succ*, *add*, *tail* and *plus* all are reflexive. The choice of the *left-most* irreflexive constructor is completely arbitrary, i.e. the rightmost irreflexive constructor or any other would do as well.

function $f(x^*:w):s \Leftarrow$
  if φ_1 **then** r_1 ,
  $\vdots$
  if φ_k **then** r_k (2.13)

is translated into a set

$$\mathrm{DEF}_f = \{\forall x^*:w\ \varphi_1 \rightarrow f(x^*) = r_1, \ldots, \forall x^*:w\ \varphi_k \rightarrow f(x^*) = r_k\} \tag{2.14}$$

of *definition formulas* for f, and we demand $\mathrm{DEF}_f \subset \Phi(S)$ for each algorithm f in the specification S. For instance, the algorithm *plus* is translated into the set of definition formulas

$$\begin{aligned}\mathrm{DEF}_{plus} = \{&\forall x, y:number\ x = 0 \rightarrow plus(x\ y) = y\ ,\\ &\forall x, y:number\\ &\quad x \neq 0 \rightarrow plus(x\ y) = succ(plus(pred(x)\ y))\}\ .\end{aligned} \tag{2.15}$$

It can be verified that all axioms obtained from the data structures and algorithms are satisfied by the standard interpretation, i.e. $\mathcal{I} \models \Phi(S)$, and therefore $\Phi(S) \subset Th(S)$, cf. [22]. Hence $\mathcal{I}$ is called the *standard model* of the axioms $\Phi(S)$ and it is also called the standard model of the *specification* S.

Our termination proof procedure generates formulas φ from an algorithm f such that $\varphi \in Th(S)$ entails the termination of f. Therefore some machine support for testing $\varphi \in Th(S)$ is required if termination of algorithms has to be verified by machine. This support is provided by an *induction theorem proving system* (see e.g. [24] for a survey and further references). Roughly speaking, such a system tries to verify $\Phi(S) \cup I(S) \vdash \varphi$ where $\vdash$ denotes derivability in some sound first-order calculus K and $I(S) \subset Th(S)$ is some decidable set of first-order formulas associated with S, called the *induction axioms* of S. If successful, then $\varphi \in Th(S)$ because all models of $\Phi(S) \cup I(S)$ are also models of φ (with K being sound), and then in particular $\mathcal{I} \models \varphi$ because $\mathcal{I} \models \Phi(S) \cup I(S)$ (with $\Phi(S) \cup I(S) \subset Th(S)$). Consequently $\Phi(S) \cup I(S) \vdash \varphi$ is sufficient for $\varphi \in Th(S)$ to hold. So for an implementation of our method each (semantical) requirement of the form "$\varphi \in Th(S)$" which is stipulated subsequently must be replaced by a call of an induction theorem prover to test for "$\Phi(S) \cup I(S) \vdash \varphi$".

3. Terminating algorithms

Let **function** $f(x_1:s_1 \ldots x_n:s_n):s \Leftarrow \ldots$ be some deterministic and case-complete algorithm which is well-formed for an admissible specification S, i.e. all function symbols (except f) used in the cases of f are defined by S. Since S is admissible it has a standard model $\mathcal{I}$. The standard model gives a meaning to each function symbol (except f) in the definition of f because f is well-formed for S. Here we are concerned with the *termination* of the algorithm f and we have to define what this notion formally means.

If the interpreter has to evaluate a term $f(q_{1,0} \dots q_{n,0})$ for some input $(q_{1,0} \dots q_{n,0}) \in \mathcal{T}(\Sigma^c)_{s_1 \dots s_n}$, some other term $f(q_{1,1} \dots q_{n,1})$ stemming from a recursive call in f has to be evaluated, which in turn necessitates an evaluation of some further term $f(q_{1,2} \dots q_{n,2})$, etc. This means that the evaluation of each term $t = f(q_{1,0} \dots q_{n,0})$ defines a so-called *trace* $(q_{1,0} \dots q_{n,0}), \dots, (q_{1,i} \dots q_{n,i}), (q_{1,i+1} \dots q_{n,i+1}), \dots$, of n-tuples of constructor ground terms which arise as evaluated arguments of f on the evaluation of t. Intuitively, the algorithm f terminates iff *each* trace is *finite*, i.e. during the evaluation of $f(q_{1,0} \dots q_{n,0})$ some term $f(q_{1,k} \dots q_{n,k})$ is created which can be evaluated without a further recursion and therefore the evaluation of $f(q_{1,0} \dots q_{n,0})$ must halt after finitely many steps. The finiteness of a trace is formally captured by the notion of a well-founded relation: For a set M, a binary relation $<_M$ on M is *well-founded* iff there is no infinite sequence $m_1, m_2, m_3, \dots$ of elements in M such that $\dots <_M m_3 <_M m_2 <_M m_1$. Now termination of f can be expressed by demanding the existence of some well-founded relation $<_{\mathcal{R}}$ on $\mathcal{T}(\Sigma^c)_{s_1 \dots s_n}$ such that $(q_{1,i+1} \dots q_{n,i+1}) <_{\mathcal{R}} (q_{1,i} \dots q_{n,i})$ for each pair of adjacent terms in each trace which stems from the evaluation of any term $f(q_{1,0} \dots q_{n,0})$.

The elements of a trace (except the first) arise from the evaluation of the arguments of some recursive calls in the algorithm f. Since an algorithm may use *nested recursions*, i.e. terms of the form $f(\dots f(\dots) \dots)$, we cannot use the standard model $\mathcal{I}$ of S to evaluate the arguments of $f(t_1 \dots t_n)$ if some argument t_i contains a further call of f. Also the condition φ of a recursive case may contain recursive calls and then the truth of φ cannot be determined by $\mathcal{I}$. Therefore the standard model is extended to an interpretation which can also be applied to f-terms:

Let $\mathcal{I}_f$ be any standard interpretation for $\Sigma \cup \{f\}$ which coincides with $\mathcal{I}$ for all function symbols in Σ, i.e. $\mathcal{I}_f \models \Phi(S)$, where Σ is the set of function symbols in S and $\Phi(S)$ are the axioms of S. This means that the universe of $\mathcal{I}_f$ is exactly the set of constructor ground terms because "$\mathcal{I}_f$ is standard" is demanded, $\mathcal{I}_f$ "behaves" like $\mathcal{I}$ for each function symbol in S because $\mathcal{I}_f \models \Phi(S)$ is demanded, and $\mathcal{I}_f$ maps the function symbol f to some function defined on constructor ground terms, where arities etc. are respected of course. Each such interpretation is called an *f-expansion* of $\mathcal{I}$, and termination means that (i) the standard model $\mathcal{I}$ can be extended to an f-expansion $\mathcal{I}_f$ which satisfies the definition of f, i.e. $\mathcal{I}_f \models \mathrm{DEF}_f$, and (ii) each trace which arises from an evaluation of a term $f(\dots)$ by $\mathcal{I}_f$ is finite:

Definition 3.1. A case-complete and deterministic algorithm **function** $f(x_1 : s_1 \dots x_n : s_n) : s \Leftarrow \cdots$ terminates in an admissible specification S with standard model $\mathcal{I}$ iff f is well-formed for S and there exists an f-expansion $\mathcal{I}_f$ of $\mathcal{I}$ and some well-founded relation $<_{\mathcal{R}}$ on $\mathcal{T}(\Sigma^c)_{s_1 \dots s_n}$ such that $\mathcal{I}_f \models \mathrm{DEF}_f$ and for each $(q_1 \dots q_n) \in \mathcal{T}(\Sigma^c)_{s_1 \dots s_n}$, for each recursive case "**if** φ **then** r" of f and for each term $f(t_1 \dots t_n)$ in φ or in r

$$\mathcal{I}_f[\![x_1 \dots x_n / q_1 \dots q_n]\!] \models \varphi \text{ implies}$$
$$\mathcal{I}_f[\![x_1 \dots x_n / q_1 \dots q_n]\!](t_1 \dots t_n) <_{\mathcal{R}} (q_1 \dots q_n)\,. \tag{3.1}$$

Requirement (3.1) demands that for each n-tuple $(q_1 \ldots q_n)$ of constructor ground terms which satisfies the condition φ of a recursive case, i.e. $\mathcal{I}_f[\![x_1 \ldots x_n / q_1 \ldots q_n]\!] \models \varphi$, the arguments $(t_1 \ldots t_n)$ of a recursive call $f(t_1 \ldots t_n)$ are evaluated to some n-tuple $\mathcal{I}_f[\![x_1 \ldots x_n / q_1 \ldots q_n]\!](t_1 \ldots t_n)$ of constructor ground terms which is smaller than the n-tuple $(q_1 \ldots q_n)$ with respect to the well-founded relation $<_{\mathcal{R}}$. So f decreases each input $(q_1 \ldots q_n)$ in each recursive call with respect to a well-founded relation, and by the well-foundedness of $<_{\mathcal{R}}$ each trace is finite and each evaluation must halt after finitely many steps. The following theorem is easily proved by noetherian induction upon $<_{\mathcal{R}}$:

Theorem 3.2. *If a case-complete and deterministic algorithm* **function** $f(x_1 : s_1 \ldots x_n : s_n) : s \Leftarrow \cdots$ *is well-formed for an admissible specification S with standard model* $\mathcal{I}$, *then f terminates in S iff there exists* exactly one *f-expansion* $\mathcal{I}_f$ *of* $\mathcal{I}$ *such that* $\mathcal{I}_f \models DEF_f$.

So termination means that there is exactly one total function which satisfies the definition formulas of f. We illustrate Definition 3.1 by some examples. Suppose that S contains the data structure *number* and consider the algorithm

function $f1(n : number) : number \Leftarrow$
  if $n = 0$ **then** 1
  if $n \neq 0$ **then** $f1(f1(n-1))$.[4]

We find that $\mathcal{I}_{f1}(f1(0)) = 1$ for each $f1$-expansion which satisfies DEF_{f1}. Hence requirement (3.1) demands in particular $\mathcal{I}_{f1}(f1(0)) = 1 <_{\mathcal{R}} 1$ and $<_{\mathcal{R}}$ cannot be well-founded because there is no well-founded relation $<_M$ such that $m <_M m$ for some $m \in M$. Consequently $f1$ does not terminate in S. The algorithm

function $f2(n : number) : number \Leftarrow$
  if $n = 0$ **then** 0
  if $n \neq 0$ **then** $2 + f2(f2(n-1))$

does not terminate in S either: If $\mathcal{I}_{f2}$ is an $f2$-expansion such that $\mathcal{I}_{f2} \models \mathrm{DEF}_{f2}$, then in particular $\mathcal{I}_{f2} \models [f2(2) = 2 + f2(2)]$. But $q \neq 2 + q$ for all numbers q, i.e. there is no standard interpretation which satisfies DEF_{f2} and consequently the algorithm $f2$ does not terminate in S.[5] But the algorithm

function $f3(n : number) : number \Leftarrow$
  if $n = 0$ **then** 0
  if $n \neq 0$ **then** $f3(f3(n-1))$

[4] Here we write e.g. 1 instead of *succ*(0), $1 + \cdots$ instead of *succ*(...), ... -1 instead of *pred*(...) etc. to ease readability.

[5] Note that the non-termination of both algorithms has different reasons: *more than one* total function satisfies the definition of the algorithm $f1$ whereas *no* total function satisfies the definition formulas of the algorithm $f2$.

terminates in S because the $f3$-expansion $\mathcal{I}_{f3}$ which assigns $f3$ the constant mapping 0 satisfies DEF_{f3} and with $\mathcal{I}_{f3}(q) = q <_{\mathcal{R}} 1 + q$ as well as $\mathcal{I}_{f3}(f3(q)) = 0 <_{\mathcal{R}} 1 + q$ for each number q, where $<_{\mathcal{R}}$ is the usual (well-founded) "less-than" relation on numbers, requirement (3.1) is satisfied.

Proving termination according to Definition 3.1 requires some intuition and skill because the required f-expansion $\mathcal{I}_f$ has to be found such that requirement (3.1) can be verified. Since $\mathcal{I}_f \models \mathrm{DEF}_f$ is demanded, proving termination necessitates that the operation which is computed by the algorithm f (or relevant properties of this operation at least) must be known to verify the termination of f. This means that generally reasoning about an algorithm's *semantics* is required for proving termination and *correctness* and *termination* have to be shown *simultaneously*[6] (see e.g. the termination proofs for McCarthy's 91-function and for Ashcroft's algorithm for list reversal [11, 12], for Takeuchi's function [16] or for the algorithm *norm*2 in [17]).

Since reasoning about an algorithm's semantics can be very difficult (and in particular for a machine) it seems worthwhile to look for some stronger termination requirement. Consider the following definition:

Definition 3.3. A case-complete and deterministic algorithm **function** $f(x_1 : s_1 \ldots x_n : s_n) : s \Leftarrow \cdots$ *strongly* terminates in an admissible specification S with standard model $\mathcal{I}$ iff f is well-formed for S and there is some well-founded relation $<_{\mathcal{R}}$ on $\mathcal{T}(\Sigma^c)_{s_1 \ldots s_n}$ such that for each $(q_1 \ldots q_n) \in \mathcal{T}(\Sigma^c)_{s_1 \ldots s_n}$, for each recursive case "**if** φ **then** r" of f, for each term $f(t_1 \ldots t_n)$ in φ or in r, and for *each* f-expansion $\mathcal{I}_f$

$$\mathcal{I}_f[\![x_1 \ldots x_n / q_1 \ldots q_n]\!] \models \varphi \text{ implies}$$
$$\mathcal{I}_f[\![x_1 \ldots x_n / q_1 \ldots q_n]\!](t_1 \ldots t_n) <_{\mathcal{R}} (q_1 \ldots q_n)\,. \tag{3.2}$$

For instance, the algorithms *plus* and *lt* from Section 2 strongly terminate. The difference between termination and strong termination is that Definition 3.3 does not demand the satisfiability of DEF_f by a standard interpretation, but instead requirement (3.2) must hold for *all* f-expansions of $\mathcal{I}$. Consequently strong termination is independent of an algorithm's semantics.

It can be shown that strong termination entails termination (cf. [22, Lemma B.3.3]). But strong termination is not *necessary* for termination, i.e. there are terminating algorithms like $f3$ which do not *strongly* terminate. Consider the algorithms $f1$, $f2$ and $f3$ above. Since the recursive calls in these algorithms coincide, strong termination of $f3$ would entail strong termination of $f1$ and also of $f2$ which in turn entails the termination of $f1$ and $f2$, i.e. a contradiction. Since requirement (3.2) must hold for *all* f-expansions, it must in particular also hold for those which do *not* satisfy DEF_f and then the termination test may fail. Here there is no well-founded relation $<_{\mathcal{R}}$ such that (3.2) holds for the $f3$-expansion

[6] This also holds if other formal frameworks (which may even use *partial* functions) are used for proving termination, cf. [3, 11, 12, 17].

$\mathcal{I}_{f3}$ which assigns $f3$ the successor function because then $\mathcal{I}_{f3}(f3(q)) = 1 + q <_{\mathcal{R}} 1 + q$ for each number q.

Strong termination is easier to verify than general termination because the satisfiability of DEF_f is not demanded. Instead only the recursive calls in an algorithm are considered, which means in particular that the *result* of a non-recursive case and the *context* of a recursive call are irrelevant for strong termination. So strong termination is not destroyed if an algorithm is modified by altering the result in a non-recursive case or the context of a recursive call. This does not hold for termination in general. E.g. algorithm $f1$ differs from $f3$ only in the result of the non-recursive case and $f2$ differs from $f3$ only in the context of the recursive call.

Strong termination provides the formal base on which our termination proof procedure rests. But since strong termination is only sufficient for termination it seems worthwhile to consider which terminating algorithms fail to terminate strongly. The following lemma gives some insight into the situation:

Lemma 3.4. *If a case-complete and deterministic algorithm* **function** $f(x_1 : s_1 \ldots x_n : s_n) : s \Leftarrow \cdots$ *without recursive calls in the conditions of the cases and without nested recursions terminates in an admissible specification S, then f strongly terminates in S.*

Proof. Requirement (3.1) holds for *some* f-expansion of $\mathcal{I}$ because f terminates. Since neither φ nor the argument t_i of a recursive call contain an f-term, (3.1) also holds for *each* f-expansion of $\mathcal{I}$. Hence f strongly terminates in S. □

We call an algorithm f *normal* iff f has no recursive calls in the conditions of the cases and has no nested recursions, and Lemma 3.4 shows that for normal algorithms strong termination is also necessary for termination. In other words, we generally fail in proving termination at most for algorithms with nested recursions or with recursions in conditions if we only test for *strong* termination. But this restriction seems not too strong, because non-normal algorithms—although of theoretical interest—seem not to be very relevant in practice.[7]

In this paper only *normal* algorithms are considered because this eases the required formalism. Strictly speaking this provides a further restriction of the class of algorithms which are considered for termination, because there are also non-normal but strongly terminating algorithms, e.g.

function $f4(n, m : number) : number \Leftarrow$
  if $n = 0$ **then** $1 + m$
  if $n \neq 0$ **then** $f4(n - 1\ f4(n - 1\ m))$.

[7] The above algorithm $f3$, McCarthy's 91-function, Takeuchi's function, Ashcroft's algorithm and Paulson's algorithm *norm2* are examples of terminating non-normal algorithms which do not strongly terminate.

Other examples for such algorithms are the usual algorithm for *Ackermann's function* (cf. [11, 12]), the algorithm *value* in [2] or the algorithm *normif* in [17]. The restriction to normal algorithms is only for the sake of the presentation and not necessitated by our proposal for proving termination (cf. [22]).

Subsequently we develop a termination proof procedure which generates a so-called *termination hypothesis* τ for each recursive call in a normal algorithm f such that $\tau \in Th(S)$ entails requirement (3.2) of Definition 3.3. Each termination hypothesis is given to an induction theorem prover for verification, and if successful strong termination of f is proved.

4. The estimation calculus

4.1. The size order

A frequently used well-founded order for termination proofs compares the *size* of the data objects under consideration. For instance, stacks are compared by their *depth*, lists by their *length*, trees by their *number of nodes*, etc. We obtain an abstract notion of size by counting the number of occurrences of *reflexive* constructors in a constructor ground term (where substructures are ignored). Formally the size of a constructor ground term is given by the *s-size measure* $\#_s : \mathcal{T}(\Sigma^c) \to \mathbb{N}$ which is defined as

$$\#_s(q) = \begin{cases} 0\,, & \text{if } q \notin \mathcal{T}(\Sigma^c)_s\,, \\ 0\,, & \text{if } q = ircons(\ldots)\,, \\ 1 + \sum_{i=1}^{n} \#_s(q_i)\,, & \text{if } q = rcons(q_1 \ldots q_n)\,, \end{cases}$$

where *ircons* is some irreflexive and *rcons* is some reflexive constructor of s. Now comparing the s-sizes of a pair of constructor ground terms of sort s with the usual $<_{\mathbb{N}}$ relation on the natural numbers $\mathbb{N}$, we obtain the *size order* $<_{\#}$ which is a well-founded relation for the constructor ground terms of the data structure s.

Since we only count occurrences of s-constructors, substructures of the data structure s are ignored. Hence we do not count the occurrences of *succ* if we compare elements of the data structure *list*, and therefore

$$add(succ(succ(0))\ empty) <_{\#} add(0\ add(0\ empty))$$

because

$$\#_{list}(add(succ(succ(0))\ empty)) = 1 <_{\mathbb{N}} 2 = \#_{list}(add(0\ add(0\ empty)))\,.$$

All terms built with irreflexive constructors are minimal elements of the size order because only reflexive constructors are considered. Therefore *empty* and 0 are $<_{\#}$-minimal constructor ground terms. For the *S-expressions* of LISP [15], which are defined in our notation by

structure *atom*(*index* : *number*)

nil cons (*car* : *sexpr cdr* : *sexpr*) : *sexpr* ,

we obtain *nil* and all terms of the form *atom*(...) as $<_{\#}$-minimal elements in $\mathcal{T}(\Sigma^{c})_{sexpr}$.

Since we intend to use the size order to formulate and to verify the termination of algorithms, we extend our many-sorted language $\mathcal{F}(\Sigma, \mathcal{V})$ by two new binary predicate symbols $<_{\#}$ and $\leqslant_{\#}$. The semantics of these predicate symbols are defined by

$$\mathcal{I}[\![x^*/q^*]\!] \models t <_{\#} r \quad \text{iff}$$
$$\#_s(\mathcal{I}[\![x^*/q^*]\!](t)) <_{\mathbb{N}} \#_s(\mathcal{I}[\![x^*/q^*]\!](r)) , \tag{4.1}$$
$$\mathcal{I}[\![x^*/q^*]\!] \models t \leqslant_{\#} r \quad \text{iff}$$
$$\#_s(\mathcal{I}[\![x^*/q^*]\!](t)) \leqslant_{\mathbb{N}} \#_s(\mathcal{I}[\![x^*/q^*]\!](r)) , \tag{4.2}$$

where $\mathcal{I}$ is the standard model of an admissible specification S, t and r are terms from $\mathcal{T}(\Sigma, \mathcal{V})_s$ and x^* are the variables in t and r which are replaced by the constructor ground terms in q^*. Now assume that *mod* computes the remainder operation (cf. Section 7) and consider the following algorithm for the computation of the greatest common divisor:

function *gcd*(*n*, *m* : *number*) : *number* $\Leftarrow$
if $n = m$ **then** n
if $n \neq m \wedge n = 0$ **then** m
if $n \neq m \wedge m = 0$ **then** n
if $n \neq m \wedge n \neq 0 \wedge m \neq 0$ **then** *gcd*(*mod*(*n m*) *mod*(*m n*)) .

A termination requirement for *gcd* can be formulated in the extended many-sorted language as

$$[\forall n, m : number \;\; mod(n\, m) \leqslant_{\#} n] \in Th(S) , \tag{4.3}$$
$$[\forall n, m : number \;\; mod(m\, n) \leqslant_{\#} m] \in Th(S) , \tag{4.4}$$
$$[\forall n, m : number \;\; n \neq m \wedge n \neq 0 \wedge m \neq 0$$
$$\rightarrow mod(n\, m) <_{\#} n \vee mod(m\, n) <_{\#} m] \in Th(S) \tag{4.5}$$

and our general problem is to verify statements of the form

$$[\forall x^* : w \;\; t \leqslant_{\#} r] \in Th(S) , \tag{4.6}$$
$$[\forall x^* : w \;\; \varphi \rightarrow \cdots \vee t <_{\#} r \vee \cdots] \in Th(S) . \tag{4.7}$$

A straightforward idea for proving those statements is to invent some set of axioms describing $<_{\#}$ as well as $\leqslant_{\#}$ and then to call an induction theorem prover for verification of (4.6) and (4.7). However, we do not follow this idea. Instead a calculus is developed, called the *estimation calculus* or the *E-calculus* for short,

which eases to prove those statements. The formulas of the E-calculus are called *estimation formulas*, where we define:

Definition 4.1. An expression of the form $\langle t \leqslant_{\#} r, \Delta \rangle$ is an *estimation formula* iff t and r are terms from $\mathcal{T}(\Sigma, \mathcal{V})_s$ and Δ is a quantifier-free formula.

An estimation formula $\langle t \leqslant_{\#} r, \Delta \rangle$ is *true*, abbreviated $\mathit{TRUE}\langle t \leqslant_{\#} r, \Delta \rangle$, iff $[\forall x^* : w\, t \leqslant_{\#} r] \in Th(S)$ and $[\forall x^* : w\, \Delta \leftrightarrow t <_{\#} r] \in Th(S)$, where x^* are the variables in t, r and Δ.

We let $\vdash_\Gamma \langle t \leqslant_{\#} r, \Delta \rangle$ denote that $\langle t \leqslant_{\#} r, \Delta \rangle$ can be derived in the E-calculus and we demand that the E-calculus is *sound*, i.e. $\vdash_\Gamma \langle t \leqslant_{\#} r, \Delta \rangle$ implies $\mathit{TRUE}\langle t \leqslant_{\#} r, \Delta \rangle$. Then statements like (4.6) and (4.7) can be proved by showing

$$\vdash_\Gamma \langle t \leqslant_{\#} r, \Delta \rangle\,, \tag{4.8}$$

$$\Phi(S) \cup I(S) \vdash [\forall x^* : w\, \varphi \rightarrow \cdots \vee \Delta \vee \cdots]\,, \tag{4.9}$$

using an induction theorem prover for (4.9). Since the E-calculus is sound, requirement (4.8) implies (4.6) as well as $[\forall x^* : w\, \Delta \leftrightarrow t <_{\#} r] \in Th(S)$ and with requirement (4.9) statement (4.7) then also holds.

Of course, our approach seems somewhat indirect as compared to the method of using axioms to prove (4.6) and (4.7) directly. But as will become obvious later on, our method has certain advantages because $\vdash_\Gamma$ is *decidable* and trivial to compute and verification problems like (4.9) are usually much easier solved than verification problems like (4.7).

4.2. Argument-bounded functions

Suppose that $[\forall x^* : w\, t \leqslant_{\#} r] \in Th(S)$ has to be verified for a pair of terms t and r with variables x^*. A common verification technique is to use estimations for proving those inequalities. Assume, for instance, we have to verify

$$\forall x, y, z : \mathit{number} \quad (x - y)/z \leqslant_{\#} x \tag{4.10}$$

where $\leqslant_{\#}$ coincides with $\leqslant_{\mathbb{N}}$ here. Assume further that we know certain *estimation laws* about the operations, viz. $n/m \leqslant_{\#} n$ for the truncated quotient and $(n - m) \leqslant_{\#} n$ for subtraction.[8] Then we may conclude with the first law that $(x - y)/z \leqslant_{\#} (x - y)$, with the second law we find that $(x - y) \leqslant_{\#} x$, and with the transitivity of $\leqslant_{\#}$ inequality (4.10) is proved. All we used in the proof were both estimation laws and the transitivity of $\leqslant_{\#}$.

We formalize this technique for proving inequalities by defining a relation on terms which mirrors the *semantical* $\leqslant_{\#}$ relation on the *syntactical level*. Therefore let us look closer at the estimation laws which were used in the example: Obviously both are very similar by stating that the size of a function's result is

[8] We assume for the sake of the example that $n/0 = 0$ and $0 - m = 0$.

always less than or equal to the size of its first argument. We call such functions 1-*bounded*. Of course, we may use other estimation laws stating that the result of a function is bounded by one of its arguments other than the first. We formalize this property with the notion of an *argument-bounded* function:

Definition 4.2. A function symbol $f : s_1 \times \cdots \times s_n \to s$ is *p-bounded* iff $1 \leq p \leq n$ and

$$[\forall x_1 : s_1 \ldots x_n : s_n \, f(x_1 \ldots x_n) \leq_\# x_p] \in Th(S) .$$

A function symbol f is *argument-bounded* iff f is *p-bounded* for some argument position p of f.

For each $p \in \mathbb{N}$, $\Gamma_p(S)$ denotes a set of p-bounded function symbols in an admissible specification S and $\Gamma(S)$ is the family of all these sets. We write Γ_p and Γ if S is known from the context.

Argument-bounded functions are frequently used in computer science to define algorithms recursively. For instance, all *reflexive* selectors are 1-bounded provided they return an appropriate result if applied to a constructor to whom they do not belong (cf. Section 2.3). Hence *car*, *cdr*, *pred* and *tail* are 1-bounded if we assume e.g. $car(nil) = cdr(nil) = car(atom(\ldots)) = cdr(atom(\ldots)) = nil$, $tail(empty) = empty$ and $pred(0) = 0$ because then $\{\forall x : number \; pred(x) \leq_\# x, \forall x : list \; tail(x) \leq_\# x, \forall x : sexpr \; car(x) \leq_\# x, \forall x : sexpr \; cdr(x) \leq_\# x\} \subset Th(S)$.

If we define, for instance,

function *minus*$(x, y : number) : number \Leftarrow$

if $x = 0 \vee y = 0$ **then** x

if $x \neq 0 \wedge y \neq 0$ **then** *minus*$(pred(x) \; pred(y))$

and

function *half*$(x : number) : number \Leftarrow$

if $pred(x) = 0$ **then** 0

if $pred(x) \neq 0$ **then** $succ(half(pred(pred(x))))$,

then *minus* and *half* are 1-bounded because $[\forall x, y : number \; minus(x \; y) \leq_\# x] \in Th(S)$ and $[\forall x : number \; half(x) \leq_\# x] \in Th(S)$. Also the (truncated) *quotient* and the *remainder* operation are 1-bounded if we use definitions which guarantee $[\forall x : number \; quotient(x \; 0) \leq_\# x] \in Th(S)$ and $[\forall x : number \; mod(x \; 0) \leq_\# x] \in Th(S)$. The remainder is also 2-bounded if $[\forall x : number \; mod(x \; 0) \leq_\# 0] \in Th(S)$ is satisfied.[9]

In COMMONLISP [20] for instance *nthcdr*, *member*, *intersection* and *remove*

[9] We need such additional requirements only for exceptional arguments, e.g. *car*(*nil*) or division by zero, which would normally yield an undefined result. We would not need these requirements if we could use *partial* operations. But since the semantics of first-order logic demand *total* operations, we feel free for these cases to stipulate the values which are convenient for us. (Cf. Section 7.)

are 2-bounded, and *intersection* is also 1-bounded. Our algorithm *remove* from Section 1 also computes a 2-bounded function, and one can observe that all these functions are frequently used in recursive definitions of algorithms.

The notion of an argument-bounded function is the key concept for our formalization of estimation proofs: Given a family Γ of argument-bounded function symbols, a decidable *estimation relation* $\leqslant_\Gamma$ on $\mathcal{T}(\Sigma, \mathcal{V})_s$ can be defined such that

$$t_1 \leqslant_\Gamma t_n \quad \text{implies} \quad [\forall x^* : w\, t_1 \leqslant_\# t_n] \in Th(S) \tag{4.11}$$

for all terms t_1 and t_n, where x^* is a list of all variables in t_1 and t_n. The estimation relation $\leqslant_\Gamma$ is based on the knowledge about data structures and the argument-bounded functions in Γ. It provides a *deductive requirement* for the $\leqslant_\#$ relation, and the proof technique of estimation is mirrored by $\leqslant_\Gamma$.

Suppose that $[\forall x, y : number\ minus(half(pred(x))\ succ(y)) \leqslant_\# x] \in Th(S)$ has to be verified, where it is known that *minus*, *half* and *pred* are 1-bounded, i.e. $\{minus, half, pred\} \subset \Gamma_1$. Then we conclude that $minus(half(pred(x))\ succ(y)) \leqslant_\Gamma half(pred(x))$ because *minus* is 1-bounded, we conclude that $half(pred(x)) \leqslant_\Gamma pred(x)$ because *half* is 1-bounded, and we finally conclude that $pred(x) \leqslant_\Gamma x$ because *pred* is 1-bounded. Therefore $minus(half(pred(x))\ succ(y)) \leqslant_\Gamma x$ is established as it was demanded. So the general idea to establish $t_1 \leqslant_\Gamma t_n$ is to test whether t_n is a subterm of t_1, as x is a subterm of $minus(half(pred(x))\ succ(y))$, where however only subterms in positions p are inspected for which a function is p-bounded.

4.3. Difference functions for argument-bounded functions

The estimation relation $\leqslant_\Gamma$ only provides a deductive requirement for the (semantical) $\leqslant_\#$ relation, cf. requirement (4.11). But we need a deductive means for the strict $<_\#$ relation because this is the well-founded relation our termination proofs are based on. We therefore introduce the notion of a *p-difference function*:

Definition 4.3. A function symbol $d : s_1 \times \cdots \times s_n \to bool$ is a *p-difference function* for a *p-bounded* function $f : s_1 \times \cdots \times s_n \to s$ iff

$$[\forall x_1 : s_1 \ldots x_n : s_n\ d(x_1 \ldots x_n) \leftrightarrow f(x_1 \ldots x_n) <_\# x_p] \in Th(S)\,.$$

A function d is a *difference function* for f iff d is a p-difference function for the p-bounded function f.

Subsequently we assume the existence of a difference function for each argument-bounded function in Γ and we let $\Delta^p f$ denote the p-difference function for $f \in \Gamma_p$.

A function may be p-bounded for more than one argument position p, as the function $min : number \times number \to number$ for the minimum of a pair of numbers

is 1- and also 2-bounded. For those functions a difference function is required for each such argument position p, as $\Delta^1 min$ and $\Delta^2 min$ exist for min.[10]

If a p-bounded function f is applied to some input $q_1 \ldots q_n$ such that it returns something $<_\#$-smaller than q_p, then the associated difference function $\Delta^p f$ applied to the same input yields *true*, and it returns *false* if f returns something of the same size as the input on argument position p. For instance, we may define

function $\Delta^1 minus(x, y : number) : bool \Leftarrow$

if $x = 0 \vee y = 0$ **then** *false*

if $x \neq 0 \wedge y \neq 0$ **then** *true*

function $\Delta^1 pred(x : number) : bool \Leftarrow$

if $x = 0$ **then** *false*

if $x \neq 0$ **then** *true*

function $\Delta^1 half(x : number) : bool \Leftarrow$

if $x = 0$ **then** *false*

if $x \neq 0$ **then** *true*

because then

$$[\forall x, y : number\ \Delta^1 minus(x\ y) \leftrightarrow minus(x\ y) <_\# x] \in Th(S)\,,$$

$$[\forall x : number\ \Delta^1 pred(x) \leftrightarrow pred(x) <_\# x] \in Th(S)\,,$$

$$[\forall x : number\ \Delta^1 half(x) \leftrightarrow half(x) <_\# x] \in Th(S)\,.$$

Using difference functions, a syntactical requirement for the strict $<_\#$ relation can be formulated. Consider again the estimation from Section 4.2

$$minus(half(pred(x))\ succ(y)) \leqslant_\Gamma half(pred(x)) \leqslant_\Gamma pred(x) \leqslant_\Gamma x\,. \tag{4.12}$$

If we can establish that at least one of the three inequalities is strict, i.e. (i) $minus(half(pred(x))\ succ(y)) <_\# half(pred(x))$ or (ii) $half(pred(x)) <_\# pred(x)$ or (iii) $pred(x) <_\# x$, then (iv) $minus(half(pred(x))\ succ(y)) <_\# x$ is obviously proved.

The strictness of each estimation step can be expressed by a literal which is built with a difference function. We find e.g. $\Delta^1 minus(half(pred(x))\ succ(y))$ as an equivalent requirement for (i), we find $\Delta^1 half(pred(x))$ as an equivalent requirement for (ii) and we find $\Delta^1 pred(x)$ as an equivalent requirement for the truth of (iii). Consequently

$$\Delta^1 minus(half(pred(x))\ succ(y)) \vee \Delta^1 half(pred(x)) \vee \Delta^1 pred(x) \tag{4.13}$$

is an equivalent requirement for the truth of (iv). Restated in a more readable

[10] We may define e.g. $\Delta^1 min(x\ y) = lt(y\ x)$ and $\Delta^2 min(x\ y) = lt(x\ y)$.

notation, it is inferred that $(x-1)/2-(y+1)<x \leftrightarrow (x-1)/2 \neq 0 \wedge y+1 \neq 0 \vee x-1 \neq 0 \vee x \neq 0$ for all numbers x and y.

So the general idea here is to scan an estimation $t_1 \leqslant_\Gamma t_2 \leqslant_\Gamma \cdots \leqslant_\Gamma t_{n-1} \leqslant_\Gamma t_n$ step by step. For each estimation step $t_i \leqslant_\Gamma t_{i+1}$, where $t_i = f_i(\ldots t_{i+1} \ldots)$ and t_{i+1} stands in position p_i of the p_i-bounded function f_i, the corresponding "call" of the p_i-difference function $\Delta^{p_i} f_i(\ldots t_{i+1} \ldots)$ is collected. From all "calls" in the collection the disjunction

$$\Delta^{p_1} f_1(\ldots t_2 \ldots) \vee \Delta^{p_2} f_2(\ldots t_3 \ldots) \vee \cdots \vee \Delta^{p_{n-1}} f_{n-1}(\ldots t_n \ldots),$$

called the *difference equivalent* $\Delta_\Gamma(t_1, t_n)$ of t_1 and t_n, is computed. Since each literal of $\Delta_\Gamma(t_1, t_n)$ is built with a difference function, an equivalent requirement for $t_1 <_\# t_n$ is generated, and we have

$$t_1 \leqslant_\Gamma t_n \quad \text{implies} \quad [\forall x^* : w\ \Delta_\Gamma(t_1, t_n) \leftrightarrow t_1 <_\# t_n] \in Th(S) \tag{4.14}$$

where x^* are the variables in t_1 and t_n.

4.4. Estimation rules based on argument-bounded functions

Knowing a family Γ of argument-bounded functions and the difference functions for them, our idea of proving inequalities by estimations can be formalized further: We define the axioms and the inference rules (both also called *estimation rules*) of the E-calculus such that inequalities (represented as estimation formulas) can be formally derived. The estimation rules combine the test for the estimation relation $\leqslant_\Gamma$ and the computation of the difference equivalent Δ_Γ such that

$$t \leqslant_\Gamma r \quad \text{iff} \quad \vdash_\Gamma \langle t \leqslant_\# r, \Delta_\Gamma(t, r)\rangle \tag{4.15}$$

holds. The E-calculus is "parameterized" with Γ, i.e. $\vdash_\Gamma \langle t \leqslant_\# r, \Delta\rangle$ denotes that we derive an estimation formula by using only the information about the argument-boundedness of the functions given by Γ.

On defining the estimation rules we have to care about the soundness of the calculus, i.e. that $\vdash_\Gamma \langle t \leqslant_\# r, \Delta\rangle$ implies $\mathit{TRUE}\langle t \leqslant_\# r, \Delta\rangle$ (cf. Definition 4.1). Since $[\forall x^* : w\ \ t \leqslant_\# t] \in Th(S)$ and $[\forall x^* : w\ \ \text{FALSE} \leftrightarrow t <_\# t] \in Th(S)$ obviously holds for all terms $t \in \mathcal{T}(\Sigma, \mathcal{V})$, $\langle t \leqslant_\# t, \text{FALSE}\rangle$ is a true estimation formula and we may use

$$\frac{}{\langle t \leqslant_\# t, \text{FALSE}\rangle} \quad \text{for all } t$$

as an axiom of the calculus, called the *identity* rule.

Now suppose that $f : s_1 \times \cdots \times s_n \to s$ is a p-bounded function with p-difference function $\Delta^p f : s_1 \times \cdots \times s_n \to \mathit{bool}$. Then for all terms $t_1, \ldots, t_n$

$$\mathit{TRUE}\langle f(t_1 \ldots t_n) \leqslant_\# t_p, \Delta^p f(t_1 \ldots t_n)\rangle \tag{4.16}$$

by Definitions 4.1, 4.2 and 4.3. Assume further that $\vdash_\Gamma \langle t_p \leqslant_\# r, \Delta\rangle$. Then by the soundness of the E-calculus

$$\mathit{TRUE}\langle t_p \leqslant_\# r, \Delta\rangle\ . \tag{4.17}$$

From (4.16) and (4.17) we conclude

$$\mathit{TRUE}\langle f(t_1 \ldots t_n) \leqslant_\# r, \Delta^p f(t_1 \ldots t_n) \vee \Delta\rangle\ . \tag{4.18}$$

We formalize this reasoning step by an inference rule, called the *argument estimation* rule:

$$\frac{\langle t_p \leqslant_\# r, \Delta\rangle}{\langle f(t_1 \ldots t_n) \leqslant_\# r, \Delta^p f(t_1 \ldots t_n) \vee \Delta\rangle} \quad \text{for all } f \in \Gamma_p(S), t_1, \ldots, t_n, r, \Delta\ .$$

For instance, the 1-bounded function *minus* with 1-difference function $\Delta^1 minus$ yields the argument estimation rule

$$\frac{\langle t_1 \leqslant_\# r, \Delta\rangle}{\langle minus(t_1\ t_2) \leqslant_\# r, \Delta^1 minus(t_1\ t_2) \vee \Delta\rangle} \quad \text{for all } t_1, t_2, r, \Delta\ .$$

With similar rules for *half* and *pred* the formal derivation in the E-calculus

$$\begin{aligned}
&\langle x \leqslant_\# x\ , && \mathrm{FALSE}\rangle\\
&\langle pred(x) \leqslant_\# x\ , && \Delta^1 pred(x)\rangle\ ^{11}\\
&\langle half(pred(x)) \leqslant_\# x\ , && \Delta^1 half(pred(x)) \vee \Delta^1 pred(x)\rangle\\
&\langle minus(half(pred(x))\ succ(y)) \leqslant_\# x\ , && \Delta^1 minus(half(pred(x))\ succ(y)) \vee\\
& && \Delta^1 half(pred(x)) \vee \Delta^1 pred(x)\rangle
\end{aligned}$$

is obtained for our example from Sections 4.2 and 4.3, where we start with the identity rule and then use the argument estimation rule three times, viz. for *pred*, *half* and *minus*.

4.5. Estimation rules based on data structures

We define some additional estimation rules to increase the performance of the E-calculus in proving inequalities. However, we will not obtain a *complete* calculus because we do not provide rules such that e.g. $\vdash_\Gamma \langle x \leqslant_\# plus(x\ y), y \neq 0\rangle$. But we do not bother with this incompleteness because we use the calculus only for the special estimation problems necessitated by termination proofs and the computation of difference functions (cf. Sections 5 and 6). As will be demonstrated subsequently, the E-calculus is "complete enough" for our purposes.

All estimation rules introduced in this section depend only on the definition of

[11] We sometimes simplify $\varphi \vee \mathrm{FALSE}$ to φ and $\varphi \vee \mathrm{TRUE}$ to TRUE in our examples.

the data structures, i.e. they can be uniformly obtained from the data structures in a specification. Hence no specific knowledge is involved with these rules as opposed to the argument estimation rule which presupposes that p-bounded functions and also the p-difference functions for them are known.

We start with the axioms of the calculus, i.e. the estimation rules with an empty list of premises: Consider a pair of terms $ircons_1(\ldots)$ and $ircons_2(\ldots)$ with irreflexive constructors as leading function symbols, i.e. the s-size of both terms is 0. Then $[\forall \ldots ircons_1(\ldots) \leq_\# ircons_2(\ldots)] \in Th(S)$ and $[\forall \ldots \text{FALSE} \leftrightarrow ircons_1(\ldots) <_\# ircons_2(\ldots)] \in Th(S)$ obviously holds, and it is sound to use

$$\frac{-}{\langle ircons_1(t_1 \ldots t_n) \leq_\# ircons_2(r_1 \ldots r_m), \text{FALSE}\rangle} \quad \text{for all } t_i, r_j$$

as an axiom, called the *equivalence* rule. Now e.g. $\vdash_\Gamma \langle atom(n) \leq_\# nil, \text{FALSE}\rangle$ holds.

Next we compare a term $ircons(\ldots)$ with a term $rcons(\ldots)$ where $ircons$ is an irreflexive constructor and $rcons$ is a reflexive constructor, i.e. the s-size of $ircons(\ldots)$ is 0 and the s-size of $rcons(\ldots)$ is 1 at least. Then $[\forall \ldots ircons(\ldots) \leq_\# rcons(\ldots)] \in Th(S)$ and $[\forall \ldots \text{TRUE} \leftrightarrow ircons(\ldots) <_\# rcons(\ldots)] \in Th(S)$ obviously holds, and it is sound to use

$$\frac{-}{\langle ircons(t_1 \ldots t_n) \leq_\# rcons(r_1 \ldots r_m), \text{TRUE}\rangle} \quad \text{for all } t_i, r_j$$

as an axiom, called the *strong estimation* rule. Now e.g. $\vdash_\Gamma \langle atom(n) \leq_\# cons(x\ y), \text{TRUE}\rangle$ holds.

Now we compare a pair of terms $ircons(\ldots)$ and r of sort s such that $ircons$ is an irreflexive constructor, i.e. the s-size of $ircons(\ldots)$ is 0 and therefore $[\forall \ldots ircons(\ldots) \leq_\# r] \in Th(S)$. The s-size of r is 1 at least iff $r = rcons_j(sel_{j_1}(r) \ldots sel_{j_{h(j)}}(r))$ for some reflexive constructor $rcons_j$ with selectors $sel_{j_1}, \ldots, sel_{j_{h(j)}}$. Hence if $rcons_1, \ldots, rcons_n$ are all reflexive constructors of s, then $[\forall \ldots (r = rcons_1(sel_{1_1}(r) \ldots sel_{1_{h(1)}}(r)) \vee \cdots \vee r = rcons_n(sel_{n_1}(r) \ldots sel_{n_{h(n)}}(r))) \leftrightarrow ircons(\ldots) <_\# r] \in Th(S)$, and it is sound to use

$$\frac{-}{\langle ircons(t_1 \ldots t_n) \leq_\# r, r = rcons_1(sel_{1_1}(r) \ldots sel_{1_{h(1)}}(r)) \vee \cdots \vee r = rcons_n(sel_{n_1}(r) \ldots sel_{n_{h(n)}}(r))\rangle} \quad \text{for all } t_i, r$$

as an axiom, called the *minimum* rule. Now e.g. $\vdash_\Gamma \langle atom(n) \leq_\# x,\ x = cons(car(x)\ cdr(x))\rangle$ holds.

We continue with the non-axiom rules: Suppose that some term t is compared with $rcons(r_1 \ldots r_k \ldots r_n)$, where k is a reflexive argument position of the reflexive constructor $rcons$ and therefore $[\forall \ldots r_k <_\# rcons(r_1 \ldots r_k \ldots r_n)] \in Th(S)$. Suppose further that $\vdash_\Gamma \langle t \leq_\# r_k, \Delta\rangle$. Then $[\forall \ldots t \leq_\# r_k] \in Th(S)$ and therefore $[\forall \ldots \text{TRUE} \leftrightarrow t <_\# rcons(r_1 \ldots r_k \ldots r_n)] \in Th(S)$, and it is sound to use

$$\frac{\langle t \leq_{\#} r_k, \Delta\rangle}{\langle t \leq_{\#} rcons(r_1 \ldots r_k \ldots r_n), \text{TRUE}\rangle} \quad \text{for all } t, r_j, \Delta$$

as an inference rule, called the *strong embedding* rule. Now e.g. $\vdash_\Gamma \langle car(y) \leq_{\#} cons(x\ y), \text{TRUE}\rangle$ holds.

Finally we consider a pair of terms $rcons(t_1 \ldots t_n)$ and $rcons(r_1 \ldots r_n)$ where *rcons* is a reflexive constructor. If $\vdash_\Gamma \langle t_j \leq_{\#} r_j, \Delta_j\rangle$ for all reflexive argument positions $j \in \{j_1, \ldots, j_h\}$ of *rcons*, then $[\forall \ldots rcons(t_1 \ldots t_n) \leq_{\#} rcons(r_1 \ldots r_n)] \in Th(S)$ and $[\forall \ldots (\Delta_{j_1} \vee \cdots \vee \Delta_{j_h}) \leftrightarrow rcons(t_1 \ldots t_n) <_{\#} rcons(r_1 \ldots r_n)] \in Th(S)$ obviously holds. Therefore it is sound to use

$$\frac{\langle t_{j_1} \leq_{\#} r_{j_1}, \Delta_{j_1}\rangle, \ldots, \langle t_{j_h} \leq_{\#} r_{jh}, \Delta_{jh}\rangle}{\langle rcons(t_1 \ldots t_n) \leq_{\#} rcons(r_1 \ldots r_n), \Delta_{j_1} \vee \cdots \vee \Delta_{j_h}\rangle} \quad \text{for all } t_i, r_i, \Delta_i$$

as an inference rule, called the *weak embedding* rule. Now. e.g. $\vdash_\Gamma \langle cons(car(x)\ cdr(y)) \leq_{\#} cons(x\ y), \Delta^1 car(x) \vee \Delta^1 cdr(y)\rangle$ holds.

A *deduction* of $\langle t_n \leq_{\#} r_n, \Delta_n\rangle$ in the E-calculus, called an *E-deduction* for short, is a sequence of estimation formulas $\langle t_1 \leq_{\#} r_1, \Delta_1\rangle, \ldots, \langle t_n \leq_{\#} r_n, \Delta_n\rangle$ such that each estimation formula $\langle t_k \leq_{\#} r_k, \Delta_k\rangle$ in the sequence either is an axiom or can be inferred by one of the inference rules from some estimation formulas preceding $\langle t_k \leq_{\#} r_k, \Delta_k\rangle$ in the sequence.

$\vdash_{\Gamma(S)} \langle t_n \leq_{\#} r_n, \Delta_n\rangle$ denotes the existence of such an E-deduction. The *estimation relation* $t \leq_{\Gamma(S)} r$ abbreviates that $\vdash_{\Gamma(S)} \langle t \leq_{\#} r, \Delta\rangle$ for some formula Δ, and the *difference equivalent* $\Delta_{\Gamma(S)}(t, r)$ of t and r is some formula Δ for which $\vdash_{\Gamma(S)} \langle t \leq_{\#} r, \Delta\rangle$ holds.

It remains to verify the soundness of the estimation calculus:

Theorem 4.4. If $\vdash_\Gamma \langle t \leq_{\#} r, \Delta\rangle$, then $\mathit{TRUE}\langle t \leq_{\#} r, \Delta\rangle$.

Proof. Each axiom provides a true estimation formula and true estimation formulas are obtained from true estimation formulas by each inference rule as it was verified in Sections 4.4 and 4.5. Using these facts, the statement is easily proved by induction upon the length of the E-deduction. □

4.6. *A proof procedure for the estimation calculus*

A proof procedure for the E-calculus is obtained by using the estimation rules in reverse direction. Each estimation rule can be written as

$$\frac{\langle t_1 \leq_{\#} r_1, \Delta_1\rangle, \ldots, \langle t_k \leq_{\#} r_k, \Delta_k\rangle}{\langle t \leq_{\#} r, \gamma \vee \Delta_1 \vee \cdots \vee \Delta_k\rangle} \qquad \text{(ER)}$$

where FALSE must be substituted for the formula in γ in case of weak embedding to obtain the general form (ER) and the antecedent of the rule is empty in case of an axiom. From each estimation rule (ER), a *production rule*

$$(E \cup \{t \leqslant_{\#} r\}, \Delta) \Rightarrow (E \cup \{t_1 \leqslant_{\#} r_1, \ldots, t_k \leqslant_{\#} r_k\}, \gamma \vee \Delta) \qquad \text{(PR)}$$

is obtained and we have

$$\vdash_\Gamma \langle t_1 \leqslant_{\#} t_2, \Delta\rangle \quad \text{iff} \quad (\{t_1 \leqslant_{\#} t_2\}, \text{FALSE}) \Rightarrow^+ (\emptyset, \Delta)\,, \qquad (4.19)$$

where $\Rightarrow^+$ is the reflexive and transitive closure of $\Rightarrow$.

It can be easily observed from the definitions of the estimation rules that $\{t_1 \leqslant_{\#} r_1, \ldots, t_k \leqslant_{\#} r_k\}$ has a strictly smaller number of variable and function symbols than $\{t \leqslant_{\#} r\}$ for each production rule (PR). Therefore $\Rightarrow$ is *noetherian*, i.e. there is no infinite derivation with respect to $\Rightarrow$. Also only finitely many production rules can be applied to $(\{t \leqslant_{\#} r\}, \Delta)$, which means that $\Rightarrow$ is *locally finite*. Both properties of $\Rightarrow$ imply that $\vdash_\Gamma \langle t \leqslant_{\#} r, \Delta\rangle$ is *decidable*.

A proof procedure for the estimation calculus can be implemented by a production rule interpreter. This system requires some search control because $\Rightarrow$ is not *confluent*, i.e. there are terms t_1 and t_2 such that

$$(E_1, \Delta_1) \;{}^+\!\Leftarrow (E \cup \{t_1 \leqslant_{\#} t_2\}, \Delta) \Rightarrow^+ (E_2, \Delta_2)$$

for some (E_1, Δ_1) and (E_2, Δ_2) but

$$(E_1, \Delta_1) \Rightarrow^* (E', \Delta') \;{}^*\!\Leftarrow (E_2, \Delta_2)$$

for no (E', Δ'), where $\Rightarrow^*$ is the reflexive closure of $\Rightarrow^+$. So it may happen that for some $\Delta_1 \neq \Delta_2$

$$(\emptyset, \Delta_1) \;{}^+\!\Leftarrow (\{t_1 \leqslant_{\#} t_2\}, \Delta) \Rightarrow^+ (\emptyset, \Delta_2)\,,$$

as $(\{0 \leqslant_{\#} succ(x)\},\ \Delta) \Rightarrow (\emptyset,\ \text{TRUE})$ by strong estimation and $(\{0 \leqslant_{\#} succ(x)\},\ \Delta) \Rightarrow (\emptyset,\ succ(x) = succ(pred(succ(x))) \vee \Delta)$ by the minimum rule. This means that the computed difference equivalent depends upon the selection of the estimation rules. However this problem can be tolerated because Δ_1 and Δ_2 are equivalent by (4.19) and Theorem 4.4. A more serious problem arises if for all Δ'

$$(\emptyset, \Delta_1) \;{}^+\!\Leftarrow (\{t_1 \leqslant_{\#} t_2\}, \Delta) \Rightarrow^+ (E, \Delta_2) \not\Rightarrow^* (\emptyset, \Delta')\,.$$

For instance, $(\{pred(x) \leqslant_{\#} pred(x)\},\ \Delta) \Rightarrow (\emptyset,\ \Delta)$ by identity and $(\{pred(x) \leqslant_{\#} pred(x)\},\ \Delta) \Rightarrow (x \leqslant_{\#} pred(x),\ \Delta^1 pred(x) \vee \Delta) \not\Rightarrow^* (\emptyset,\ \Delta')$ if argument estimation is used. This shows that a dead end can be entered if the wrong estimation rule is selected.

We solve this problem by an appropriate interaction of the rules which minimizes backtracking (see [25] for details).

5. Generating termination hypotheses

5.1. Computing termination hypotheses with the E-calculus

Using the estimation relation $\leqslant_\Gamma$ and the difference equivalent Δ_Γ, the generation of termination hypotheses is straightforward. If the algorithm tested for termination is given as **function** $f(x : s') : s \Leftarrow \cdots$ and $f(t)$ is a recursive call in a

case "**if** φ **then** r" of f, then it is tested whether $t \leqslant_\Gamma x$ and if successful the *termination hypothesis* $[\forall x : s'\ \varphi \rightarrow \Delta_\Gamma(t, x)]$ is generated for this recursion. This is performed for each recursive call and each termination hypothesis is given to an induction theorem prover for verification. If a proof can be found, then $[\forall x : s'\ \varphi \rightarrow t <_\# x] \in Th(S)$ holds for each recursion and consequently the algorithm f terminates.

E.g. for proving the termination of the algorithm *half* from Section 4.2, $\vdash_\Gamma \langle pred(pred(x)) \leqslant_\# x,\ \Delta^1 pred(pred(x)) \vee \Delta^1 pred(x) \rangle$ is deduced in the E-calculus and therefore the termination hypothesis

$$\forall x : number \quad pred(x) \neq 0 \rightarrow \Delta^1 pred(pred(x)) \vee \Delta^1 pred(x)\,, \tag{5.1}$$

i.e. $x - 1 \neq 0 \rightarrow x - 1 \neq 0 \vee x \neq 0$, is generated for *half*. For the algorithm *sort* from the introduction $\vdash_\Gamma \langle remove(minimum(x)\ x) \leqslant_\# x,\ member(minimum(x)\ x) \rangle$ is computed, where *remove* is assumed as a 2-bounded function with 2-difference function *member*. Hence the induction theorem prover is called with the termination hypothesis

$$\forall x : list \quad x \neq empty \rightarrow member(minimum(x)\ x)\,. \tag{5.2}$$

Our method is easily generalized for proving termination of algorithms with more than one argument: for an algorithm **function** $f(x_1 : s_1 \ldots x_n : s_n) : s \Leftarrow \cdots$ a set of *termination positions* $P \subset \{1, \ldots, n\}$ is computed such that $i \in P$ iff i is an argument position with $t_i \leqslant_\Gamma x_i$ for *each* recursive call $f(t_1 \ldots t_n)$ in the algorithm. The termination test fails if $P = \emptyset$. Otherwise a termination hypothesis $[\forall x_1 : s_1 \ldots x_n : s_n\ \ \varphi \rightarrow \bigvee_{i \in P} \Delta_\Gamma(t_i,\ x_i)]$ is generated for each recursive call $f(t_1 \ldots t_n)$ in a case "**if** φ **then** r" of f. If each termination hypothesis can be proven, then in each recursive call *at least one* argument t_j is $<_\#$-smaller than its corresponding initial input x_j. By the definition of P, no input x_i (with $i \in P$) is $<_\#$-smaller than its corresponding argument t_i in any recursive call, and therefore the algorithm f must terminate.

Consider for instance the algorithm

function $gcd2(x, y : number) : number \Leftarrow$

if $x = 0 \vee y = 0$ **then** $max(x\ y)$

if $x \neq 0 \wedge y \geqslant x$ **then** $gcd2(x\ minus(y\ x))$

if $y \neq 0 \wedge y < x$ **then** $gcd2(minus(x\ y)\ y)$.

Since $x \leqslant_\Gamma x$ as well as $minus(y\ x) \leqslant_\Gamma y$ in the first recursion and $minus(x\ y) \leqslant_\Gamma x$ as well as $y \leqslant_\Gamma y$ in the second one, $P = \{1, 2\}$ is computed (where *minus* is a 1-bounded function with difference function $\Delta^1 minus$, cf. Section 4.3). Therefore

$$\forall x, y : number \quad x \neq 0 \wedge y \geqslant x \rightarrow \text{FALSE} \vee \Delta^1 minus(y\ x)\,, \tag{5.3}$$

i.e. $x \neq 0 \wedge y \geqslant x \rightarrow y \neq 0 \wedge x \neq 0$, is the computed termination hypothesis for the first recursion, and

$$\forall x, y : number \quad y \neq 0 \wedge y < x \rightarrow \Delta^1 minus(x\ y) \vee \text{FALSE}\,, \tag{5.4}$$

i.e. $y \neq 0 \wedge y < x \rightarrow x \neq 0 \wedge y \neq 0$, is computed for the second one. We formalize this approach with the following termination criterion (see Appendix B for the proof):

Theorem 5.1. *A normal algorithm* **function** $f(x_1 : s_1 \ldots x_n : s_n) : s \Leftarrow \cdots$ *strongly terminates in an admissible specification S if there exists some non-empty set* $P \subset \{1, \ldots, n\}$ *such that for each recursive call* $f(t_1 \ldots t_n)$ *in a case "***if** φ **then** ..." *of f*

(1) $t_i \leqslant_{\Gamma(S)} x_i$ *for all* $i \in P$, *and*

(2) $[\forall x_1 : s_1 \ldots x_n : s_n\ \varphi \rightarrow \bigvee_{i \in P} \Delta_{\Gamma(S)}(t_i, x_i)] \in Th(S)$.

The formulas of requirement (2) in Theorem 5.1 are called the *termination hypotheses* for *f*. The termination hypotheses are the *weakest* requirements for the termination of an algorithm with respect to each syntactical termination criterion based on the size order, because with Theorem 4.4 the difference equivalent is sufficient and *necessary* for a strict decrease of an argument in a recursive call.[12]

The set P in Theorem 5.1, called a *measured subset* in [2], denotes the argument positions which are relevant for the termination of an algorithm. If requirements (1) and (2) of Theorem 5.1 are satisfied for an algorithm *f*, then $\tau(x_1 \ldots x_n) = \Sigma_{i \in P} \#_{s_i}(x_i)$ is a *termination function* for *f* as it can be seen from the proof of Theorem 5.1.

We may weaken the requirements of Theorem 5.1 by comparing the argument list of a recursive call with the list of initial arguments by some *lexicographical order* based on the size order. This modification of the termination criterion is required for proving termination of algorithms like Ackermann's function, cf. [22] for the generalized termination criterion.[13] Also termination requirements based on other tuple-orders built with the size order, e.g. the *multiset order* [6], may be formulated if required.

For an implementation the set P in Theorem 5.1 should be chosen as the *maximal* set of indices which satisfies requirement (1) of Theorem 5.1 as the following example reveals. Consider the algorithm *gcd* from Section 4.1, where *mod* (cf. Section 7) is a 1-bounded function with 1-difference function

function $\Delta^1 mod(n, m : number) : bool \Leftarrow$

if $n = 0 \wedge m = 0$ **then** *false*

if $n \neq 0 \wedge m = 0$ **then** *true*

if $n < m$ **then** *false*

[12] This corresponds to the requirement in [2], that a human user states each induction lemma with the "cleanest hypotheses available" in order to obtain a termination hypothesis as weak as possible. See also Section 8.

[13] We found no frequent need for lexicographical comparisons. For instance, Ackermann's function is the only algorithm in [2] which requires a lexicographical argumentation for proving termination.

if $n \geq m \wedge m \neq 0$ **then** *true* .

Since $mod(n\ m) \leqslant_\Gamma n$ we may define $P = \{1\}$, and then the false termination hypothesis

$$\forall n, m : number \quad n \neq m \wedge n \neq 0 \wedge m \neq 0 \rightarrow \Delta^1 mod(n\ m)\,, \tag{5.5}$$

i.e. $n \neq m \wedge n \neq 0 \wedge m \neq 0 \rightarrow n \geq m \wedge m \neq 0$, is obtained. Alternatively $P = \{2\}$ can be used because $mod(m\ n) \leqslant_\Gamma m$. However then the false termination hypothesis

$$\forall n, m : number \quad n \neq m \wedge n \neq 0 \wedge m \neq 0 \rightarrow \Delta^1 mod(m\ n)\,, \tag{5.6}$$

i.e. $n \neq m \wedge n \neq 0 \wedge m \neq 0 \rightarrow m \geq n \wedge n \neq 0$, is computed. But using $P = \{1, 2\}$ the true termination hypothesis

$$\forall n, m : number$$
$$n \neq m \wedge n \neq 0 \wedge m \neq 0 \rightarrow \Delta^1 mod(n\ m) \vee \Delta^1 mod(m\ n)\,, \tag{5.7}$$

i.e. $n \neq m \wedge n \neq 0 \wedge m \neq 0 \rightarrow n \geq m \wedge m \neq 0 \vee m \geq n \wedge n \neq 0$, is obtained for *gcd*.

5.2. *Improvements of the termination test*

The central idea for proving termination is to deduce $\vdash_\Gamma \langle t \leqslant_\# x,\ \Delta(t,\ x)\rangle$, where x is some formal parameter of an algorithm and t is the corresponding actual parameter in a recursive call under some condition φ, and then to prove a termination hypothesis of the form $[\forall \cdots \varphi \rightarrow \cdots \vee \Delta_\Gamma(t,\ x) \vee \cdots]$. However it may happen that $[\forall \cdots \varphi \rightarrow \cdots \vee t <_\# x \vee \cdots] \in Th(S)$ but $\not\vdash_\Gamma \langle t \leqslant_\# x, \ldots\rangle$. Sometimes the condition φ has to be considered not only for proving the termination hypothesis but also for *generating* it. The following algorithm (computing the minimal element of a non-empty list) provides an example:

function *minimum*$(x : list) : number \Leftarrow$
if $x = empty$ **then** 0
if $x = add(head(x)\ tail(x)) \wedge tail(x) = empty$ **then** $head(x)$
if $x = add(head(x)\ tail(x)) \wedge tail(x) = add(head(tail(x))\ tail(tail(x)))$
$\wedge\ head(x) > head(tail(x))$ **then** $minimum(tail(x))$
if $x = add(head(x)\ tail(x)) \wedge tail(x) = add(head(tail(x))\ tail(tail(x)))$
$\wedge\ head(x) \leqslant head(tail(x))$ **then**
$minimum(add(head(x)\ tail(tail(x))))$.

Obviously, *minimum* terminates but this cannot be verified by our method. This is because the estimation

$$add(head(x)\ tail(tail(x))) \leqslant_\Gamma x \tag{5.8}$$

does not hold and therefore the termination test fails for the second recursion of *minimum*. But since

$$x = add(head(x)\ tail(x)) \wedge tail(x) = add(head(tail(x))\ tail(tail(x)))$$

is demanded for this case we may test for

$$add(head(x)\ tail(tail(x))) \leqslant_\Gamma add(head(x)\ add(head(tail(x))\ tail(tail(x)))) \quad (5.9)$$

instead. Now with $\vdash_\Gamma \langle add(head(x)\ tail(tail(x))) \leqslant_\# add(head(x)\ add(head(tail(x))\ tail(tail(x)))), \text{TRUE}\rangle$ the trivial termination hypothesis $[\forall x : list \cdots \rightarrow \text{TRUE}]$ is generated for the second recursion.

The general problem is that we ignore the bindings imposed by the condition φ of a recursive case on the parameter x when we test for $t \leqslant_\Gamma x$. It may happen that such a condition expresses the shape of a formal parameter in terms of constructors and this information is sometimes required to compute a termination hypothesis. Consequently the termination test fails if this information is ignored, as it did for the algorithm *minimum*.

Our solution for this problem is to replace an actual parameter t and a formal parameter x by their so-called *representatives* $\lceil t \rceil_\varphi$ and $\lceil x \rceil_\varphi$, where

$$[\forall \cdots \varphi \rightarrow \lceil r \rceil_\varphi = r] \in Th(S) \quad (5.10)$$

for each term r and its representative $\lceil r \rceil_\varphi$. Now it is tested for $\lceil t \rceil_\varphi \leqslant_\Gamma \lceil x \rceil_\varphi$ and if successful the termination hypothesis

$$\forall \cdots \varphi \rightarrow \cdots \vee \Delta(\lceil t \rceil_\varphi, \lceil x \rceil_\varphi) \vee \cdots \quad (5.11)$$

is computed. This approach is sound because a proof of (5.11) entails $[\forall \cdots \varphi \rightarrow \cdots \vee \lceil t \rceil_\varphi <_\# \lceil x \rceil_\varphi \vee \cdots] \in Th(S)$ and (5.10) then implies $[\forall \cdots \varphi \rightarrow \cdots \vee t <_\# x \vee \cdots] \in Th(S)$.

For computing representatives we demand that each condition of a case is given as a conjunction of literals and certain inconsistencies are forbidden to guarantee the uniqueness of each representative. For instance, 0 and $succ(pred(x))$ both could be computed as representatives of x if (inconsistent) conditions like $x = 0 \wedge x = succ(pred(x))$ would be allowed.

Definition 5.2. A case "**if** φ **then** ..." of an algorithm is in *conjunctive normal form* (*cnf* for short) iff φ is a *conjunction* of *literals*.

A case "**if** φ **then** ..." in cnf is *structural consistent* iff φ does not contain a pair of unnegated equations $q = c_1(\ldots)$ and $q = c_2(\ldots)$ such that c_1 and c_2 are constructor functions.

An algorithm f is in *conjunctive normal form* (*cnf*) iff each case of f is in cnf, and f is in *structure normal form* (*snf*) iff f is in cnf and each case of f is structural consistent.

Each algorithm can be transformed into an equivalent snf algorithm (cf. [22]). As we have seen in the above example, the representative of x is obtained by applying equations of the condition of the algorithm's case: Using $x = add(head(x)$

tail(*x*)) we first replace *x* with *add*(*head*(*x*) *tail*(*x*)), and then obtain *add*(*head*(*x*) *add*(*head*(*tail*(*x*)) *tail*(*tail*(*x*)))) by using *tail*(*x*) = *add*(*head*(*tail*(*x*)) *tail*(*tail*(*x*))). But we use only specific equations, called *structure equations* subsequently, which express the shape of a term in terms of constructors. We do not need arbitrary equations but only structure equations for our purposes, because only structure equations support the estimations necessary for the termination test.

Definition 5.3. An equation of the form $t = cons(sel_1(t) \ldots sel_n(t))$ is a *structure equation* iff *cons* is a constructor function and $sel_1, \ldots, sel_n$ are the selectors of *cons*.

The equations $x = empty$, $x = add(head(x)\ tail(x))$, $tail(x) = add(head(tail(x))\ tail(tail(x)))$, $n = succ(pred(n))$, $pred(n) = succ(pred(pred(n)))$ and $y = cons(car(y)\ cdr(y))$ are examples for structure equations. Structure equations $t = cons(sel_1(t) \ldots sel_n(t))$ are used to compute the representative of a term *r* by replacing an occurrence of *t* in *r* by the right-hand side $cons(sel_1(t) \ldots sel_n(t))$ of the structure equation. Hence the right-hand side of a structure equation designates the term *which replaces* another term for the computation of a representative.

However, structure equations must be applied in a restricted way for obtaining useful results. For instance, we may replace *x* by *add*(*head*(*x*) *tail*(*x*)) using the structure equation $x = add(head(x)\ tail(x))$, then replace the new term by *add*(*head*(*add*(*head*(*x*) *tail*(*x*))) *tail*(*add*(*head*(*x*) *tail*(*x*)))) using the same structure equation again and so on. To avoid such replacements we also have to designate the terms which we want to replace. We call these candidates for a replacement *parameter components*, i.e. a parameter component is a term *which may be replaced* by another term when a representative is computed.

As a starting point we declare each formal parameter of an algorithm as a parameter component. Hence we may replace *x* with *add*(*head*(*x*) *tail*(*x*)) using the structure equation $x = add(head(x)\ tail(x))$. Now the shape of *add*(*head*(*x*) *tail*(*x*)) can be refined in terms of constructors if *head*(*x*) or *tail*(*x*) can be replaced by the right-hand side of a structure equation. Consequently *head*(*x*) and *tail*(*x*) are declared as parameter components, and since $tail(x) = add(head(tail(x))\ tail(tail(x)))$ is a structure equation in our example, we obtain *add*(*head*(*x*) *add*(*head*(*tail*(*x*)) *tail*(*tail*(*x*))))). We continue with *head*(*tail*(*x*)) and *tail*(*tail*(*x*)) as a parameter components. But since there are no structure equations of the form $head(tail(x))) = \cdots$ and $tail(tail(x)) = \cdots$, the computation stops with *add*(*head*(*x*) *add*(*head*(*tail*(*x*)) *tail*(*tail*(*x*)))) as a representative of *x*. This technique is formalized by the following two definitions:

Definition 5.4. Let "**if** φ **then**..." be a case of an algorithm **function** $f(x_1 : s_1 \ldots x_n : s_n) : s \Leftarrow \cdots$ in snf. Then each formal parameter x_i is a parameter component of φ, and if *t* is a parameter component of φ and φ contains an unnegated structure equation $t = cons(sel_1(t) \ldots sel_n(t))$, then $sel_1(t), \ldots, sel_n(t)$ are also parameter components of φ.

Definition 5.5. Let "**if** φ **then**..." be a case of an algorithm **function** $f(x_1 : s_1 \ldots x_n : s_n) : s \Leftarrow \cdots$ in snf. Then the *representative* of a term t with respect to φ, abbreviated $\lceil t \rceil_\varphi$, is defined as

$$\lceil t \rceil_\varphi = \begin{cases} cons(\lceil sel_1(t) \rceil_\varphi \ldots \lceil sel_n(t) \rceil_\varphi), & \text{if } t = cons(sel_1(t) \ldots sel_n(t)) \text{ is a structure equation in } \varphi \text{ and } t \text{ is a parameter component of } \varphi, \\ g(\lceil t_1 \rceil_\varphi \ldots \lceil t_n \rceil_\varphi), & \text{if } t \text{ is } g(t_1 \ldots t_n), \text{ where } g \text{ is any function symbol, and } t \text{ is not a parameter component of } \varphi, \\ t, & \text{otherwise.} \end{cases}$$

Since we define representatives only with algorithms in structure normal form, each case is structural consistent and therefore at most one structure equation $t = cons(sel_1(t) \ldots sel_n(t))$ exists for each parameter component t. The requirements in Definition 5.5 also exclude each other. Consequently each representative $\lceil t \rceil_\varphi$ is uniquely determined. Subsequently we omit subscripts and write $\lceil t \rceil$ instead of $\lceil t \rceil_\varphi$ if φ is obvious.

The computation of representatives can be viewed as a special *equational reasoning procedure* tailored for our purposes. Since we only use (structure) equations when computing the representative of a term, each term must denote the same object as its representative if the condition of the case is satisfied:

Lemma 5.6. *If* "**if** φ **then**..." *is a case of an algorithm in snf, then* $[\forall \cdots \varphi \rightarrow \lceil r \rceil_\varphi = r] \in Th(S)$ *for all terms* r.

Using term representatives we now improve our termination test by the following corollary which is an obvious consequence of Theorem 5.1 and Lemma 5.6:

Corollary 5.7. *A normal algorithm* **function** $f(x_1 : s_1 \ldots x_n : s_n) : s \Leftarrow \cdots$ *in snf strongly terminates in an admissible specification* S *if there exists some non-empty set* $P \subset \{1, \ldots, n\}$ *such that for each recursive call* $f(t_1 \ldots t_n)$ *in a case* "**if** φ **then**..." *of* f

(1) $\lceil t_i \rceil_\varphi \leqslant_{\Gamma(S)} \lceil x_i \rceil_\varphi$ *for all* $i \in P$, *and*

(2) $[\forall x_1 : s_1 \ldots x_n : s_n \varphi \rightarrow \bigvee_{i \in P} \Delta_{\Gamma(S)}(\lceil t_i \rceil_\varphi, \lceil x_i \rceil_\varphi)] \in Th(S)$.

Note that we must also use the representatives $\lceil t_i \rceil_\varphi$ of the *actual* parameters in a recursive call, because if only the representatives $\lceil x_i \rceil_\varphi$ of the formal parameters are used, termination cannot be verified for certain algorithms. Consider for instance the algorithm

function $log(n : number) : number \Leftarrow$
  if $n = 0$ **then** 0
  if $n = succ(pred(n)) \wedge pred(n) = 0$ **then** 0
  if $n = succ(pred(n)) \wedge pred(n) = succ(pred(pred(n)))$ **then**
    $succ(log(half(n)))$.

Since $half(n) \not\leq_\Gamma succ(succ(pred(pred(n)))) = \lceil n \rceil$, we cannot compute a termination hypothesis if we only use the representative of the formal parameter n. But if the representative is also computed for the actual parameter, then $\lceil half(n) \rceil = half(\lceil n \rceil) = \cdots$ is obtained (cf. Definition 5.5) and with $\vdash_\Gamma \langle half(\lceil n \rceil) \leq_\# \lceil n \rceil, \Delta^1 half(\lceil n \rceil) \rangle$ the termination hypothesis

$$\forall n : number \quad n = succ(pred(n)) \wedge pred(n) = succ(pred(pred(n))) \rightarrow \Delta^1 half(succ(succ(pred(pred(n))))) , \tag{5.12}$$

i.e. $n \neq 0 \wedge n - 1 \neq 0 \rightarrow 2 + (\ldots) \neq 0$, is computed for *log*.

5.3. Algorithms in positive structure normal form

The success of the termination criterion given by Corollary 5.7 depends on the form of the algorithm which is tested for termination. If we avoid structure equations in the condition of an algorithm, the method may fail as the following examples reveal: If the condition in the second recursive case of the algorithm *minimum* is given as

$$tail(x) = add(head(tail(x))\ tail(tail(x))) \wedge head(x) \leq head(tail(x)) \tag{5.13}$$

or is given as

$$tail(x) \neq empty \wedge head(x) \leq head(tail(x)) \tag{5.14}$$

then $x = \lceil x \rceil$ and the termination test fails. The reason for this failure is that the required structure equations are not given in the conditions, but are expressed *implicitly* instead.

For instance, $tail(x) = add(head(tail(x))\ \ tail(tail(x)))$ entails $x = add(head(x)\ tail(x))$ and with this additional structure equation

$$add(head(x)\ tail(tail(x))) \leq_\Gamma add(head(x)\ add(head(tail(x))\ tail(tail(x)))) = \lceil x \rceil$$

holds for (5.13). Also $tail(x) \neq empty$ entails $x = add(head(x)\ \ tail(x))$ as well as $tail(x) = add(head(tail(x))\ \ tail(tail(x)))$, and using these structure equations the required representative for x can be computed also for (5.14). To avoid an implicit representation of structure equations a further normal form for algorithms is defined:

Definition 5.8. An algorithm **function** $f(x_1 : s_1 \ldots x_n : s_n) : s \Leftarrow \cdots$ is in *positive structure normal form* (*psnf*) iff f is in snf and for each case "**if** φ **then**..." in f:

(1) φ does not contain a subformula of the form $t \neq cons(\ldots)$, and
(2) each equation $t = cons(\ldots)$ in φ is a structure equation such that
(3) t is a parameter component of φ,

where *cons* is any constructor function.

For instance, the algorithms *minimum* and *log* are in psnf, but *half* is not a psnf algorithm.

By requirement (1), a case like "**if** $x \neq add(y\ z)$ **then** r" is not allowed and must be replaced by the sequence of cases

if $x = empty$ **then** r ,

if $x = add(head(x)\ tail(x)) \wedge head(x) \neq y$ **then** r ,

if $x = add(head(x)\ tail(x)) \wedge head(x) = y \wedge tail(x) \neq z$ **then** r

to obtain an equivalent psnf algorithm.

A case like "**if** $x = add(y\ z)$ **then** r" is forbidden by requirement (2) and must be replaced by

if $x = add(head(x)\ tail(x)) \wedge head(x) = y \wedge tail(x) = z$ **then** r .

Requirement (3) does not allow cases like

if $tail(x) = add(head(tail(x))\ tail(tail(x)))$ **then** r ,

which are modified to

if $x = add(head(x)\ tail(x)) \wedge tail(x) = add(head(tail(x))\ tail(tail(x)))$ **then** r

for obtaining an equivalent psnf algorithm. Also cases like "**if** $tail(x) = empty$ **then** r" are not allowed and must be replaced by a sequence of cases:

if $x = empty$ **then** r ,

if $x = add(head(x)\ tail(x)) \wedge tail(x) = empty$ **then** r .

These examples illustrate how snf algorithms are transformed into equivalent psnf algorithms for increasing the success of the termination test.

6. Argument-bounded algorithms

6.1. Introduction

The generation of termination hypotheses is based on the argument-bounded functions in Γ and on their difference functions and one may wonder where the essential information actually comes from?

Often an argument-bounded function g can be recognized and a difference function for g can be found by machine. The starting point is trivial: Since each reflexive selector *rsel* of a data structure s is 1-bounded (cf. Sections 2.3 and 4.2), *rsel* is inserted into Γ_1 and a so-called 1-difference algorithm **function** $\Delta^1 rsel(x:s) : bool \Leftarrow \cdots$ is synthesized, which contains a case

if $x = ircons(sel_1(x) \ldots sel_n(x))$ **then** *false*

for each irreflexive constructor *ircons* of s, and contains a case

if $x = rcons(sel_1(x) \ldots sel_n(x))$ **then** *true*

for each reflexive constructor *rcons* of s, where $sel_1, \ldots, sel_n$ are the selectors of *ircons* or *rcons* respectively. We obtain for instance for the data structures *list* and *sexpr*

function $\Delta^1 tail(x : list) : bool \Leftarrow$
  if $x = empty$ **then** *false*
  if $x = add(head(x)\ tail(x))$ **then** *true*
function $\Delta^1 cdr(x : sexpr) : bool \Leftarrow$
  if $x = atom(index(x))$ **then** *false*
  if $x = nil$ **then** *false*
  if $x = cons(car(x)\ cdr(x))$ **then** *true*

and $\Delta^1 car$ is defined like $\Delta^1 cdr$.

Since **function** $\Delta^1 rsel(x : s) : bool \Leftarrow \cdots$ is deterministic, case-complete and (vacuously) terminates, each admissible specification remains admissible when extended by the 1-difference algorithm of a reflexive selector. So we we may assume that each admissible specification S contains the 1-difference algorithms for all reflexive selectors of the data structures in S, and then $\Delta^1 rsel$ is a 1-difference function for *rsel*.

To recognize other argument-bounded functions than reflexive selectors we define a decidable *algorithm schema* such that each instance of this schema (called a *p-bounded algorithm*) computes a p-bounded function. So for each algorithm **function** $g(x^* : w) : s \Leftarrow \cdots$ in an admissible specification it can be decided whether g is an instance of this algorithm schema. If the instance test succeeds, g must be a p-bounded function and is inserted into Γ_p.

Then a recursive *synthesis schema* is used to synthesize a so-called p-difference algorithm **function** $\Delta^p g(x^* : w) : bool \Leftarrow \cdots$ for the algorithm just recognized as p-bounded. Since each p-difference algorithm is deterministic, case-complete and terminates "by construction", each admissible specification remains admissible when extended by a p-difference algorithm. So we may assume that each admissible specification S contains a p-difference algorithm for each p-bounded algorithm g in S, and then $\Delta^p g$ is a p-difference function for g as guaranteed by the synthesis procedure.

The main idea embodied in the algorithm schema is to construct a meta-induction proof for verification that the function computed by the considered algorithm is argument-bounded. The cases of the difference algorithm are generated in parallel to the proof steps of the meta-induction. We illustrate the instance test and the synthesis of the difference algorithm with the algorithm **function** $remove(n : number\ x : list) : list \Leftarrow \cdots$ from Section 1.

For performing the instance test, i.e. to verify whether *remove* computes a 2-bounded function, the representative $\lceil r \rceil$ of the result term of each case "**if** φ **then** r" is compared with the representative $\lceil x \rceil$ of the second formal parameter,

i.e. $\vdash_\Gamma \langle \lceil r \rceil \leqslant_\# \lceil x \rceil, \Delta \rangle$ is computed. We find $\vdash_\Gamma \langle empty \leqslant_\# empty, \text{FALSE} \rangle$ for the non-recursive case "**if** $x = empty$ **then** x" which means that *remove* is 2-bounded at least for this case. Since the difference equivalent is computed as FALSE, the result in the non-recursive case never is $<_\#$-smaller than the input argument x. Therefore the case "**if** $x = empty$ **then** *false*" is inserted into the definition of the 2-difference algorithm **function** $\Delta^2 remove(n : number\ x : list) : bool \Leftarrow \cdots$ under construction. This completes the base case of the meta-induction.

In each recursive case we assume

$$\text{TRUE}\langle remove(n\ tail(x)) \leqslant_\# tail(x), \Delta^2 remove(n\ tail(x)) \rangle$$

as the induction hypothesis. Now if $\vdash_\Gamma \langle tail(x) \leqslant_\# t, \Delta \rangle$ for a term t, then $\text{TRUE}\langle tail(x) \leqslant_\# t, \Delta \rangle$ by the soundness of the E-calculus. Hence $\text{TRUE}\langle remove(n\ tail(x)) \leqslant_\# t, \Delta^2 remove(n\ tail(x)) \vee \Delta \rangle$ by the induction hypothesis. Therefore the E-calculus remains sound if we add an estimation rule

$$\frac{\langle tail(x) \leqslant_\# t, \Delta \rangle}{\langle remove(n\ tail(x)) \leqslant_\# t, \Delta^2 remove(n\ tail(x)) \vee \Delta \rangle} \quad \text{for all } t, \Delta \tag{6.1}$$

to the rules of the E-calculus. The E-calculus also remains decidable by the same argumentation as used in Section 4.6. The additional rule (6.1) is an *instance* of the argument estimation rule for *remove*, viz.

$$\frac{\langle t_2 \leqslant_\# t, \Delta \rangle}{\langle remove(t_1\ t_2) \leqslant_\# t, \Delta^2 remove(t_1\ t_2) \vee \Delta \rangle} \quad \text{for all } t_1, t_2, t, \Delta \tag{6.2}$$

because (6.1) can be obtained from (6.2) by substituting n for $t_1$and $tail(x)$ for t_2. Since we are in a proof for verification that *remove* is 2-bounded, i.e. that (6.2) is a sound estimation rule, (6.2) cannot be used in the proof. However the additional rule (6.1) can be used by an inductive argument.

We let $\vdash_{\Gamma+}$ denote a derivation in the E-calculus which is extended by (6.1), and then $\vdash_{\Gamma+} \langle remove(n\ tail(x)) \leqslant_\# add(head(x)\ tail(x)), \text{TRUE} \rangle$ is computed for the first recursive case "**if** $x \neq empty \wedge head(x) = n$ **then** $remove(n\ tail(x))$".[14] This means that *remove* is 2-bounded also for this case. Since the difference equivalent is computed as TRUE, the result in the first recursive case always is $<_\#$-smaller than the input argument x. Therefore the case "**if** $x \neq empty \wedge head(x) = n$ **then** *true*" is inserted into the definition of the 2-difference algorithm **function** $\Delta^2 remove$ under construction. This completes the first step case of the meta-induction.

[14] For sake of readability we do not present each algorithm in psnf (cf. Section 5.3). Instead we "treat" all algorithms as if they were given in psnf. For instance, $x \neq empty$ has to be read as $x = add(head(x)\ tail(x))$ and then a nontrivial representative for x can be computed.

Finally we compute $\vdash_{\Gamma+} \langle add(head(x)\ remove(n\ tail(x))) \leqslant_{\#} add(head(x)\ tail(x)),\ \Delta^2 remove(n\ tail(x))\rangle$ for the second recursive case "**if** $x \neq empty \wedge head(x) \neq n$ **then** $add(head(x)\ remove(n\ tail(x)))$", where also here (6.1) is used as an additional estimation rule which soundness is guaranteed by the induction hypothesis. Hence *remove* is 2-bounded also for this case. Since the difference equivalent is computed as $\Delta^2 remove(n\ tail(x))$, the result in the second recursive case is $<_{\#}$-smaller than the input argument x iff $\Delta^2 remove(n\ tail(x))$ is satisfied. Therefore the case "**if** $x \neq empty \wedge head(x) \neq n$ **then** $\Delta^2 remove(n\ tail(x))$" is inserted into the definition of the 2-difference algorithm **function** $\Delta^2 remove$ under construction. This completes the second step case of the meta-induction and the whole proof.

Summing up, *remove* is recognized as a 2-bounded function with 2-difference function $\Delta^2 remove$ (usually known as *member*), where this difference function is computed by a 2-difference algorithm synthesized as

function $\Delta^2 remove(n : number\ x : list) : bool \Leftarrow$

if $x = empty$ **then** *false*

if $x \neq empty \wedge head(x) = n$ **then** *true*

if $x \neq empty \wedge head(x) \neq n$ **then** $\Delta^2 remove(n\ tail(x))$.

6.2. Recognizing argument-bounded algorithms

Analyzing the above deduction for establishing the 2-boundedness of *remove* we see that all we needed was induction (based on the recursions in the algorithm) and the estimation relation (and recursion and the difference equivalent for the synthesis of the difference algorithm). Therefore we can formulate decidable syntactical requirements for an algorithm which imply that the computed function is argument-bounded.

This approach generally works because algorithms which compute p-bounded functions often have a *similar shape and structure* as the algorithm *remove* and the subsequently presented algorithms for *minus*2, *half*2 and *log*2 have. This is an empirical observation which was obtained by the analysis of several algorithms (cf. [23]). The similarity of these algorithms can be formally expressed by an algorithm schema for p-bounded algorithms such that each instance of this schema computes a p-bounded function:

Definition 6.1. An algorithm **function** $g(x_1 : s_1 \ldots x_n : s_n) : s \Leftarrow \cdots$ in an admissible specification S is p-bounded for some $p \in \{1, \ldots, n\}$ iff for each case "**if** φ **then** r" of the algorithm

$$\vdash_{\Gamma(S)+} \langle \lceil r \rceil_{\varphi} \leqslant_{\#} \lceil x_p \rceil_{\varphi}, \gamma \rangle \quad \text{for some formula } \gamma\ , \tag{6.3}$$

where $\vdash_{\Gamma(S)+}$ denotes derivability in the E-calculus which is extended by additional estimation rules

$$\frac{\langle \lceil t_p \rceil_\varphi \leqslant_\# t, \Delta \rangle}{\langle g(\lceil t_1 \rceil_\varphi \ldots \lceil t_n \rceil_\varphi) \leqslant_\# t, \Delta^p g(t_1 \ldots t_n) \vee \Delta \rangle} \quad \text{for all } t, \Delta \tag{6.4}$$

for each recursive call $g(t_1 \ldots t_n)$ in r. An algorithm is *argument-bounded* iff it is p-bounded for some argument position p.

The p-difference algorithm **function** $\Delta^p g(x_1 : s_1 \ldots x_n : s_n) : bool \Leftarrow \cdots$ for a p-bounded algorithm **function** $g(x_1 : s_1 \ldots x_n : s_n) : s \Leftarrow \cdots$ contains exactly a pair of cases

if $\varphi \wedge \gamma$ **then** *true* ,

if $\varphi \wedge \neg\gamma$ **then** *false* ,

for each case "**if** φ **then** r" in g, where γ is given as in (6.3).

Derivability $\vdash_{\Gamma(S)+}$ in the extended E-calculus depends on the cases of the algorithm which is tested for p-boundedness (cf. (6.4)). $\vdash_{\Gamma(S)+}$ coincides with $\vdash_{\Gamma(S)}$ for all non-recursive cases because no additional estimation rules exist, and $\vdash_{\Gamma(S)+}$ varies with the recursive calls in each recursive case.

The algorithm schema of Definition 6.1 is *decidable*, because (6.3) is the only requirement and $\vdash_{\Gamma(S)+}$ is decidable (cf. Section 4.6). Consequently the synthesis of a difference algorithm is *recursive*. Each p-difference algorithm is deterministic and case-complete "by construction", because each p-bounded algorithm is. Since only a terminating algorithm is tested for p-boundedness, its associated p-difference algorithm also terminates "by construction" because it uses a recursion $\Delta^p g(t_1 \ldots t_n)$ as a subformula of γ iff the p-bounded algorithm uses a recursion $g(t_1 \ldots t_n)$ in the corresponding case. Consequently an admissible specification remains admissible if it is extended by a p-difference algorithm.

The syntactical requirement demanded for an argument-bounded algorithm **function** g and the cases stipulated for its difference algorithm **function** $\Delta^p g$ allow to prove that g is a p-bounded function and $\Delta^p g$ is a p-difference function for g as expressed by the following theorem (see Appendix B for the proof):

Theorem 6.2. *If S is an admissible specification containing a p-bounded algorithm* **function** $g(x_1 : s_1 \ldots x_n : s_n) : s \Leftarrow \cdots$ *and its p-difference algorithm* **function** $\Delta^p g(x_1 : s_1 \ldots x_n : s_n) : bool \Leftarrow \cdots$, *then* $\text{TRUE} \langle g(x_1 \ldots x_n) \leqslant_\# x_p, \Delta^p g(x_1 \ldots x_n) \rangle$.

The above algorithm *remove* satisfies the requirements of Definition 6.1 as illustrated in Section 6.1 and its 2-difference algorithm is formally synthesized as

function $\Delta^2 remove(n : number\ x : list) : bool \Leftarrow$

if $x = empty \wedge$ FALSE **then** *true*

if $x = empty \wedge \neg$FALSE **then** *false*

if $x \neq empty \wedge head(x) = n \wedge$ TRUE **then** *true*

if $x \neq empty \wedge head(x) = n \wedge \neg \text{TRUE}$ **then** *false*

if $x \neq empty \wedge head(x) \neq n \wedge \Delta^2 remove(n\ tail(x))$ **then** *true*

if $x \neq empty \wedge head(x) \neq n \wedge \neg\Delta^2 remove(n\ tail(x))$ **then** *false* .

This algorithm differs from the 2-difference algorithm presented in Section 6.1, but it should be obvious how trivial transformations and simplifications yield the version of **function** $\Delta^2 remove$ as presented before. To ease readability we subsequently present examples of difference algorithms only in their simplified form (see Section 6.3 for some simplification techniques).

We give some examples for p-bounded algorithms. The following algorithm

function $minus2(n, m : number) : number \Leftarrow$

if $m = 0$ **then** n

if $m \neq 0$ **then** $minus2(pred(n)\ pred(m))$

is 1-bounded because

$$\vdash_\Gamma \langle n \leqslant_\# n, \text{FALSE} \rangle,$$

$$\vdash_{\Gamma+} \langle minus2(pred(n)\ pred(m)) \leqslant_\# n\ ,\ \Delta^1 minus2(pred(n)\ pred(m)) \vee n \neq 0 \rangle\ ,$$

where $n \neq 0$ abbreviates $\Delta^1 pred(n)$ and

$$\frac{\langle pred(n) \leqslant_\# t, \Delta \rangle}{\langle minus2(pred(n)\ pred(m)) \leqslant_\# t\ ,\ \Delta^1 minus2(pred(n)\ pred(m)) \vee \Delta \rangle}$$

for all t, Δ

is used as an additional estimation rule. Therefore the (simplified) 1-difference algorithm for *minus*2 is synthesized as

function $\Delta^1 minus2(n, m : number) : bool \Leftarrow$

if $m = 0$ **then** *false*

if $m \neq 0 \wedge n \neq 0$ **then** *true*

if $m \neq 0 \wedge n = 0$ **then** $\Delta^1 minus2(pred(n)\ pred(m))$.

The following algorithm

function $half2(n : number) : number \Leftarrow$

if $n = 0$ **then** 0

if $n \neq 0 \wedge pred(n) = 0$ **then** 0

if $n \neq 0 \wedge pred(n) \neq 0$ **then** $succ(half2(pred(pred(n))))$

is 1-bounded because

$$\vdash_{\Gamma} \langle 0 \leqslant_{\#} 0, \text{FALSE} \rangle ,$$
$$\vdash_{\Gamma} \langle 0 \leqslant_{\#} succ(0), \text{TRUE} \rangle ,$$
$$\vdash_{\Gamma+} \langle succ(half2(pred(pred(n)))) \leqslant_{\#} succ(succ(pred(pred(n)))), \text{TRUE} \rangle ,$$

where

$$\frac{\langle pred(pred(n)) \leqslant_{\#} t, \Delta \rangle}{\langle half2(pred(pred(n))) \leqslant_{\#} t, \Delta^1 half2(pred(pred(n))) \vee \Delta \rangle} \quad \text{for all } t, \Delta$$

is used as an additional estimation rule. Therefore the (simplified) 1-difference algorithm for $half2$ is synthesized as

function $\Delta^1 half2(n : number) : bool \Leftarrow$
  if $n = 0$ **then** *false*
  if $n \neq 0 \wedge pred(n) = 0$ **then** *true*
  if $n \neq 0 \wedge pred(n) \neq 0$ **then** *true* .

The following algorithm

function $log2(n : number) : number \Leftarrow$
  if $n = 0$ **then** 0
  if $n \neq 0 \wedge pred(n) = 0$ **then** 0
  if $n \neq 0 \wedge pred(n) \neq 0$ **then**
    $succ(log2(succ(half2(pred(pred(n))))))$

is 1-bounded because

$$\vdash_{\Gamma} \langle 0 \leqslant_{\#} 0, \text{FALSE} \rangle ,$$
$$\vdash_{\Gamma} \langle 0 \leqslant_{\#} succ(0), \text{TRUE} \rangle ,$$
$$\vdash_{\Gamma+} \langle succ(log2(succ(half2(pred(pred(n)))))) \leqslant_{\#} succ(succ(pred(pred(n)))) , \Delta^1 log2(succ(half2(pred(pred(n))))) \vee pred(pred(n)) \neq 0 \rangle ,$$

where the 1-boundedness of $half2$ is required to verify the 1-boundedness of $log2$, i.e. we assume $half2 \in \Gamma_1$, $pred(pred(n)) \neq 0$ abbreviates $\Delta^1 half2(pred(pred(n)))$ and

$$\frac{\langle succ(half2(pred(pred(n)))) \leqslant_{\#} t, \Delta \rangle}{\langle log2(succ(half2(pred(pred(n))))) \leqslant_{\#} t , \Delta^1 log2(succ(half2(pred(pred(n))))) \vee \Delta \rangle} \quad \text{for all } t, \Delta$$

is used as an additional estimation rule. Therefore the (simplified) 1-difference algorithm for *log*2 is synthesized as

function $\Delta^1 log2(n : number) : bool \Leftarrow$
  if $n = 0$ **then** *false*
  if $n \neq 0 \wedge pred(n) = 0$ **then** *true*
  if $n \neq 0 \wedge pred(n) \neq 0 \wedge pred(pred(n)) \neq 0$ **then** *true*
  if $n \neq 0 \wedge pred(n) \neq 0 \wedge pred(pred(n)) = 0$ **then**
    $\Delta^1 log2(succ(half2(pred(pred(n)))))$.

6.3. Optimization of difference algorithms

Since the definition formulas of a difference algorithm are used in termination proofs (cf. Section 2.3), we are interested in obtaining a difference algorithm as simple as possible. This is of proof-technical relevance because the simpler the difference algorithms are, the easier the termination hypotheses are to prove.

The synthesized difference algorithms often contain redundant case-conditions, superfluous cases and unnecessary recursions. For instance, the 2-difference algorithm $\Delta^2 remove$ given in Section 6.2 has redundant case-conditions of the form $\varphi \wedge \neg$FALSE and $\varphi \wedge$ TRUE which can be simplified. It also contains superfluous cases of the form "**if** ... $\wedge$ FALSE **then** ..." and "**if** ... $\wedge \neg$TRUE **then** ..." which can be eliminated. Such modifications of a difference algorithm are performed by *condition subsumption* which uses an induction theorem prover to verify $[\forall \cdots \varphi \rightarrow \gamma] \in Th(S)$ for each pair of cases

if $\varphi \wedge \gamma$ **then** *true* ,
if $\varphi \wedge \neg\gamma$ **then** *false* ,

in a difference algorithm. If successful both cases are replaced by "**if** φ **then** *true*". Otherwise it is tested for $[\forall \cdots \varphi \rightarrow \neg\gamma] \in Th(S)$, and "**if** φ **then** *false*" replaces both cases if this test succeeds.

Difference algorithms are also modified by *case merging*, which means that each pair of cases

if $\varphi \wedge L$ **then** r ,
if $\varphi \wedge \neg L$ **then** r ,

in a difference algorithm is replaced by "**if** φ **then** r". For instance, case merging simplifies the 1-difference algorithm for *half*2 from Section 6.2 to

function $\Delta^1 half2(n : number) : bool \Leftarrow$
  if $n = 0$ **then** *false*
  if $n \neq 0$ **then** *true* .

The costs of a termination proof (measured in the number of inductive

argumentations) directly depends on the number of recursive calls in a difference algorithm and therefore it is useful to eliminate recursions in a difference algorithm if possible.[15] For instance, the difference algorithm $\Delta^1 log2$ from Section 6.2 contains a recursive case

if $n \neq 0 \wedge pred(n) \neq 0 \wedge pred(pred(n)) = 0$ **then**

$$\Delta^1 log2(succ(half2(pred(pred(n))))) \tag{6.5}$$

and consequently the proof of a termination hypothesis as $[\forall n : number\ n \neq 0 \rightarrow \Delta^1 log2(n)]$ requires induction. After replacement of the recursive call by *true*, case merging can be applied which yields the simplified difference algorithm

function $\Delta^1 log2(n : number) : bool \Leftarrow$

if $n = 0$ **then** *false*

if $n \neq 0$ **then** *true* .

Now $[\forall n : number\ n \neq 0 \rightarrow \Delta^1 log2(n)]$ can be proved by propositional reasoning only. We therefore always attempt to eliminate recursions in a difference algorithm **function** $\Delta^p g(x_1 : s_1 \ldots x_n : s_n) : bool \Leftarrow \cdots$ by replacing some recursive cases with non-recursive cases.

Let Ψ be some non-empty set of formulas such that "**if** ψ_i **then** $\Delta^p g(t_{i,1} \ldots t_{i,n})$" is a case in **function** $\Delta^p g$ for each $\psi_i \in \Psi$ and let Φ_b be the set of all formulas φ_j such that "**if** φ_j **then** b" is a case in **function** $\Delta^p g$ where $b \in \{true, false\}$. Now if

$$[\forall x^* : w\ \psi_i \rightarrow \bigvee_{\psi \in \Psi} \delta_i(\psi) \vee \bigvee_{\varphi \in \Phi_b} \delta_i(\varphi)] \in Th(S) \tag{6.6}$$

can be verified for each $\psi_i \in \Psi$, where $\delta_i = \{x_1/t_{i,1}, \ldots, x_n/t_{i,n}\}$, then each recursive case "**if** ψ_i **then** $\Delta^p g(t_{i,1} \ldots t_{i,n})$" can be replaced by "**if** ψ_i **then** b" without altering the operation computed by **function** $\Delta^p g$. The formula in (6.6) is a so-called *recursion elimination formula* and the truth of all these formulas guarantees that an equivalent difference algorithm is obtained if recursive calls are eliminated.

Assume that some $\psi_h \in \Psi$ is satisfied for an input q_0^* of **function** $\Delta^p g$. Since **function** $\Delta^p g$ terminates, q_0^* necessitates finitely many, say k, successive recursive calls with the cases given by Ψ. Hence some ψ_i holds on the kth call for the actual input q_k^* but no $\delta_i(\psi)$ holds for q_k^*, because otherwise we would have $k+1$ successive recursive calls with the cases given by Ψ. Hence with (6.6) some $\delta_i(\varphi)$ is satisfied for q_k^* and **function** $\Delta^p g$ returns $\delta_i(b) = b$, i.e. the same result which is computed by the modified difference algorithm. If no $\psi_h \in \Psi$ is satisfied for q_0^*, then some other condition γ of a case "**if** γ **then** r" is satisfied for q_0^*. Since $\gamma \notin \Psi$ this case is also contained in the modified algorithm, and therefore both algorithms return the same result.

For the difference algorithm $\Delta^1 log2$ from Section 6.2 we find $\Psi = \{n \neq 0 \wedge pred(n) \neq 0 \wedge pred(pred(n)) = 0\}$ and with $\Phi_{true} = \{n \neq 0 \wedge pred(n) = 0,\ n \neq 0 \wedge pred(n) \neq 0 \wedge pred(pred(n)) \neq 0\}$ the recursion elimination formula

[15] **function** $\Delta^2 remove$ is an example of a difference algorithm which cannot be defined without recursion.

$$\begin{aligned}&\forall n : number\\ &\quad n \neq 0 \wedge pred(n) \neq 0 \wedge pred(pred(n)) = 0 \rightarrow\\ &\qquad half2(pred(pred(n))) \neq 0 \wedge\\ &\qquad pred(half2(pred(pred(n)))) = 0 \vee\\ &\qquad half2(pred(pred(n))) = 0 \vee\\ &\qquad half2(pred(pred(n))) \neq 0 \wedge\\ &\qquad pred(half2(pred(pred(n)))) \neq 0\end{aligned} \tag{6.7}$$

is obtained, where we have already performed trivial simplifications like replacement of $succ(\ldots) \neq 0$ by TRUE and $pred(succ(t))$ by t to ease readability. This recursion elimination formula can be easily proved by propositional reasoning and this justifies the soundness of the recursion elimination in **function** $\Delta^1 log2$.[16]

Also the recursion in the difference algorithm $\Delta^1 minus2$ of Section 6.2 can be replaced by *false* after verification of the recursion elimination formula

$$\begin{aligned}&\forall n, m : number\\ &\quad m \neq 0 \wedge n = 0 \rightarrow pred(m) \neq 0 \wedge pred(n) = 0 \vee pred(m) = 0\,.\end{aligned} \tag{6.8}$$

7. When termination cannot be proved

Of course there are cases for which our method fails: Termination cannot be verified for non-strongly terminating algorithms as McCarthy's 91-function (cf. Section 3) and we did not consider mutual recursion. Also termination cannot be verified if a well-founded relation which is not based on the s-size measure (cf. Section 4.1) is required for the termination proof. For instance, our method is unable to prove termination of the algorithm

function $subtract1(n, m : number) : number \Leftarrow$

if $n \leq m$ **then** 0

if $n > m$ **then** $succ(subtract1(n\ succ(m)))$.

Using $\tau(n, m) := n - m$ as a termination function, termination of $subtract1$ is easily verified but this termination argument cannot be expressed by estimation formulas.

Our method also fails if an argument-bounded function g is defined by an algorithm which is not argument-bounded because then a termination hypothesis cannot be generated for an algorithm which "uses" g in a recursive call. For instance,

[16] Sometimes, as for $\Delta^1 log2$, recursions can also be eliminated by symbolic evaluation. But this technique is too weak in general, because if fails e.g. for the recursion elimination in $\Delta^1 minus2$.

function *reverse*(x : *list*) : *list* $\Leftarrow$

if x = *empty* **then** x

if $x \neq$ *empty* **then** *append*(*reverse*(*tail*(x)) *add*(*head*(x) *empty*))

defines a 1-bounded function because $\mathit{TRUE}\langle \mathit{reverse}(x) \leqslant_{\#} x, \text{ FALSE}\rangle$, but the algorithm is not 1-bounded because *append*(*reverse*(*tail*(x)) *add*(*head*(x) *empty*)) $\not\leqslant_{\Gamma+}$ *add*(*head*(x) *tail*(x)). Therefore *reverse* $\notin \Gamma_1$ and the termination hypothesis $[\forall x : \mathit{list}\ x \neq \mathit{empty} \rightarrow \text{TRUE}]$ cannot be generated for

function *shuffle*(x : *list*) : *list* $\Leftarrow$

if x = *empty* **then** x

if $x \neq$ *empty* **then** *add*(*head*(x) *shuffle*(*reverse*(*tail*(x)))) .

This problem is caused by the incompleteness of the E-calculus but (by our experience) such failures rarely occur in practice because algorithms like *reverse* are rarely used in the recursive calls of other algorithms.

Another example is the algorithm

function *mod*(n, m : *number*) : *number* $\Leftarrow$

if $m = 0$ **then** m

if $n < m$ **then** n

if $n \geqslant m \wedge m \neq 0$ **then** *mod*(*minus*(n m) m) ,

which defines a 2-bounded function because $\mathit{TRUE}\langle \mathit{mod}(n\ m) \leqslant_{\#} m, m \neq 0\rangle$. This algorithm is 1-bounded but it is not 2-bounded because $n \not\leqslant_{\Gamma} m$. Here a remedy is to replace the result n in the second case of *mod* by *min*(n m), where *min* (computing the minimum of its inputs) is defined by a 1- and 2-bounded algorithm. Since $\mathit{min}(n\ m) \leqslant_{\Gamma} n$ the modified algorithm still is 1-bounded and with $\mathit{min}(n\ m) \leqslant_{\Gamma} m$ the algorithm now is also 2-bounded. This solution seems quite ad hoc but (besides *mod*) we only know of one (non-artificial) algorithm, viz. *greatest.factor* in [2], which requires such a modification.

Sometimes argument-boundedness cannot be verified because a result term in an algorithm is not "maximally evaluated". For instance, the 1-boundedness of *log* (cf. Section 5.2) cannot be verified because *half*(*succ*(*succ*(*pred*(*pred*(n))))) $\not\leqslant_{\Gamma}$ *succ*(*pred*(*pred*(n))). Such problems are avoided if terms are evaluated before they are compared by the estimation relation. Here *half*(*succ*(*succ*(*pred*-(*pred*(n))))) must be replaced by *succ*(*half*(*pred*(*pred*(n)))) and then 1-boundedness can be verified, cf. *log*2 in Section 6.2.

The test for argument-boundedness also fails if a required term representative cannot be computed because the necessary structure equation is only implicitly given. We have already demanded that algorithms are in positive structure normal form for avoiding these problems (cf. Section 5.3), but this is not a general remedy. For instance,

function *subtract2*$(n, m : number) : number \Leftarrow$

if $n \leqslant m$ **then** 0

if $n > m$ **then** *succ*(*subtract2*(*pred*(n) m))

defines a 1-bounded function because $\mathit{TRUE}\langle \mathit{subtract2}(n\ m) \leqslant_{\#} n,\ n \neq 0 \wedge m \neq 0\rangle$, but the algorithm is not 1-bounded since $\mathit{succ}(\mathit{subtract2}(\mathit{pred}(n)\ m)) \not\leqslant_{\Gamma^+} n = \lceil n \rceil$. Here the condition in the recursive case must be replaced by $n > m \wedge n = \mathit{succ}(\mathit{pred}(n))$ and then 1-boundedness can be verified since now $\lceil n \rceil = \mathit{succ}(\mathit{pred}(n))$.

Finally, the method fails if results are stipulated for "don't care" inputs such that the function defined by an algorithm is not argument-bounded only for this reason. For instance, the algorithm *gcd* from Section 4.1 still terminates if the first case in *mod* is given as "**if** $m = 0$ **then** *succ*(0)". But then *mod* is not 1-bounded because $\mathit{succ}(0) \not\leqslant_{\Gamma} n$ and consequently no termination hypothesis can be generated for *gcd*. Therefore "helpful" results, as 0, should be stipulated for irrelevant function applications, as *mod*$(n\ 0)$, such that argument-boundedness is supported (cf. Section 4.2).

8. Related work

Research on proving termination is closely related to research on program verification in general. The idea of using termination functions and properties of well-founded sets for proving termination of flow chart programs is suggested by Floyd [7].

Manna [10] presents a procedure which synthesizes a first-order formula φ_P from a flow chart program P such that P terminates under an interpretation I (which assigns operations over particular domains to the operation symbols in P) if and only if φ_P is valid under the same interpretation. For proving the formula φ_P axioms Φ_P which are satisfied by I and also axioms modeling the involved operations are needed.[17] The axioms Φ_P define a well-founded relation by which P terminates or else contradict φ_P. Consequently the problem of finding a termination argument for P is mapped into the problem of finding the adequate axiom set Φ_P. Viewed in this context, our method computes an axiom set Φ_P such that Φ_P is satisfied by I and a proof of φ_P from Φ_P can be uniformly obtained. For the introductory example from Section 1, for instance, Φ_P contains only formula (1.5). Our procedure fails in the computation of Φ_P if P does not terminate, but it may fail also for programs which do terminate.

A system for automated program verification is presented in [5], where proving termination is essentially based on Floyd's ideas. The paper describes a semi-automatic facility for the generation and verification of *convergence conditions* which are sufficient for the termination of the loops in a program.

[17] Note that φ_P as well as Φ_P contain "new" relation symbols not in P.

Based on Hoare's method, Manna and Pnueli [13] present an axiomatic approach which allows to prove both correctness and termination of while-programs by one unified formalism. Here termination of a while-statement is proved by defining a termination function (called *convergence function* in the paper) and then to infer with the proposed inference rules that the value of the termination function decreases by the execution of the loop. Examples are given, including an algorithm for the greatest common divisor and the partition problem from Hoare's FIND algorithm [8]. The example proofs in the paper give an impression of the complexity and difficulty of verifying termination with a rigorous formal statement.[18]

Katz and Manna [9] compare four methods for proving termination. *Floyd's* technique [7] and the *loop approach* use termination functions, where in the latter approach an upper bound for each increasing counter of a loop has to be established. In the *exit approach* termination is proved by showing that some so-called *exit conditions*, which have to be generated for a loop, are satisfied at some stage of the computation. Unlike the former methods (and our proposal), also *non-termination* can be handled with this method. Finally *Burstall's* method [4] which proves termination and correctness simultaneously by structural induction is discussed. The success of this method depends on the invention of the "right" well-founded order for the given verification problem. *Burstall's* method seems especially suited to show the termination of algorithms using nested recursions and therefore being difficult to understand and to verify (cf. Section 3). Because one needs to know what these algorithms do when proving termination, correctness and termination have to be shown simultaneously. This observation is also well recognized in the papers of Boyer and Moore [3] and of Paulson [17] who compares three methods for proving termination, viz. using measurement functions, in domain theory using LCF and showing that the relation defined by the recursive calls is well-founded. Burstall's approach is further developed with the *intermittent-assertion method* [14].

A system for proving termination of PROLOG programs is presented in [1] where termination properties are proved by reasoning with specific equations which have to be associated with a program. Proving termination of PROLOG programs by inferring certain inequations is considered in [21] and in [18].

Our proposal for proving termination has similarities with the method implemented in the NQTHM system of Boyer and Moore [2]. There, an initial specification S_0 is assumed which contains the data structures *bool* and *number* and the algorithms *plus* and *lt* (cf. Section 2). Each data structure s implicitly

[18] The recursive versions of the algorithms for the greatest common divisor and the partition problem can be proved to be terminating with our method, cf. [23]. It must be admitted, however, that proving termination is much more complicated for algorithms defined in an imperative language like ALGOL 60 with assignments, goto- and while-statements than for algorithms given in a pure functional notation.

defines an algorithm **function** $count\_s(x:s):number \Leftarrow \cdots$ which provides an algorithmic definition of the s-size measure $\#_s$ as defined in Section 4.1. Therefore each estimation formula $\langle t \leqslant_{\#} r,\ \Delta\rangle$ could be represented in the NQTHM system by the first-order formulas $[\forall \cdots lt(count\_s(t)\ count\_s(r)) \vee count\_s(t) = count\_s(r)]$ and $[\forall \cdots \Delta \leftrightarrow lt(count\_s(t)\ count\_s(r))]$. For proving termination so-called *induction lemmata* are formulated by the user of the system. These formulas must be proved by the system and if successful they are associated with a certain label such that the system remembers them when proving termination. The formula

$$\forall n : number\ \forall x : list$$
$$member(n\ x) \rightarrow count\_list(remove(n\ x)) < count\_list(x) \tag{8.1}$$

is an example of an induction lemma for *remove*, where $t < r$ abbreviates $lt(t\ r)$, and an induction lemma

$$\forall x_1 : s_1 \ldots x_n : s_n \quad \Delta^p g(x_1 \ldots x_n) \rightarrow count\_s(g(x_1 \ldots x_n)) < count\_s(x_p) \tag{8.2}$$

can be formulated for each p-bounded algorithm g with difference algorithm $\Delta^p g$ by the user of the NQTHM system. Now if termination of an algorithm like *sort* from Section 1 has to be proven, the system recognizes the call of *remove* in the recursion of *sort*, remembers the induction lemma for *remove* and generates the same termination hypothesis for *sort*, viz. (1.4), as our method does.

The induction lemma technique is more powerful than our method because measurement algorithms other than *count_s* may be used in the formulation of an induction lemma. For instance, a termination hypothesis for the algorithm *subtract*1 from Section 7 can be computed with the induction lemma

$$\forall x, y : number \quad x < y \rightarrow minus(y\ succ(x)) < minus(y\ x)\,. \tag{8.3}$$

This means that strong termination can be proved also for algorithms for which our method fails.

The induction lemma technique provides less automatization than our proposal because the measurement algorithms and the induction lemmata have to be formulated by the user of the system. This means that the idea why an algorithm terminates has to be found by a human. Our method also requires less proof obligations. For instance, for proving the termination of *sort* in the NQTHM system the user has to define the algorithm *member* and to formulate the induction lemma (8.1), which is performed automatically by the synthesis of a difference algorithm in our method. Then the system has to prove the termination of *member* and the truth of the induction lemma, which is guaranteed "by construction" in our method. The remaining steps, viz. the generation and the proof of the termination hypothesis for *sort*, are similar in both methods.

9. Conclusion

Our method has been implemented in the induction theorem prover INKA.[19] After a new data structure is defined, the systems labels each reflexive selector as 1-bounded and computes the corresponding 1-difference algorithm as described in Section 6.1. If a new algorithm is defined, termination hypotheses are generated for the algorithm as discussed in Section 5, and the system tries to prove these conjectures. If successful, the algorithm is inserted into the data base. Then the system decides for each argument position p whether the new algorithm is p-bounded and a p-difference algorithm is synthesized for each such argument position (cf. Section 6.2). After each p-difference algorithm has been optimized by condition subsumption, case merging and recursion elimination (cf. Section 6.3), it is also inserted into the data base and the system is ready for the next input.

Since this approach may also produce false conjectures some precautions must be taken to avoid wasting theorem proving resources. If an algorithm does not terminate, our method computes a false termination hypothesis (if it computes a termination hypothesis at all). Also an optimization may not be applicable and then false formulas are generated by condition subsumption or by recursion elimination. Therefore the system tries to falsify each system-computed conjecture before a proof attempt is made (cf. [19]).

The INKA system performs successfully on many algorithms. A collection of 50 algorithms is presented in [23], including classical sorting algorithms and algorithms for standard arithmetical operations, and it is shown which termination hypotheses are generated for the algorithms in the collection. Generally termination proofs require induction, but—as it can be seen from the examples—the generated termination hypotheses often can be proved by cases and propositional reasoning only (cf. Appendix A).

The data base from [2] was also used as a benchmark. In this data base 73 algorithms terminate with selector functions and 6 terminate with other argument-bounded algorithms. Our system proves the termination of all these algorithms automatically (where the algorithm for *greatest.factor* must be modified as described in Section 8). This means that our system implicitly synthesizes all the induction lemmata which have to be submitted to the NQTHM system by the system user. But three algorithms in this data base terminate with a well-founded relation which is not based on the s-size measure, viz. *normalize*, *gopher* and *samefringe*, and consequently our method fails for these algorithms.

[19] Induction theorem proving systems have to solve their own termination problems (and the work presented here was inspired by the development of such a system). Verification of termination is a central problem in automated induction because knowing the reason(s) why an algorithm terminates is the most important key to find a useful induction axiom for a formula (cf. [2, 24]).

Acknowledgements

I am indebted to Norbert Eisinger and to my colleagues from the former INKA-group, Susanne Biundo, Birgit Hummel and Dieter Hutter, for their comments on a first draft of this work. I am grateful to Maritta Heisel, who accompanied several iterations of the material with constructive criticism, helpful suggestions and encouragement, and last but not least to Jürgen Giesl and two anonymous referees for further constructive comments.

Appendix A

We illustrate our method by six sorting algorithms which are standard material in computer science ground courses. For each algorithm the termination hypotheses which are computed according to the termination criterion of Corollary 5.7 are presented. As may be observed from the presentation, all termination hypotheses can be proved by propositional reasoning and cases only.

We also present the difference algorithms (synthesized according to Definition 6.1) for the argument-bounded algorithms (recognized according to Definition 6.1) which are used in the sorting algorithms. All difference algorithms are given in their optimized form (cf. Section 6.3), and it is easily seen that all formulas justifying the soundness of the optimizations can be proved by propositional reasoning and cases only.

For saving space, auxiliary algorithms as **function** $last(x : list) : number \Leftarrow \cdots$ (for the computation of the rightmost element of a list) etc. are omitted if they do not illustrate our method.

A.1. Bubblesort

function $bubble(x : list) : list \Leftarrow$
  if $x = empty$ **then** x
  if $x \neq empty \wedge tail(x) = empty$ **then** x
  if $x \neq empty \wedge tail(x) \neq empty \wedge head(x) \leqslant head(tail(x))$ **then**
    $add(head(tail(x))\ bubble(add(head(x)\ tail(tail(x)))))$
  if $x \neq empty \wedge tail(x) \neq empty \wedge head(x) > head(tail(x))$ **then**
    $add(head(x)\ bubble(tail(x)))$

computes a permutation of x such that the last list element is a minimal element of the list. *bubble* is a 1-bounded algorithm with 1-difference algorithm (obtained after recursion elimination)

function $\Delta^1 bubble(x : list) : bool \Leftarrow$
  if TRUE **then** *false* .

function *but.last*(x : *list*) : *list* $\Leftarrow$

if $x = empty$ **then** *empty*

if $x \neq empty \wedge tail(x) = empty$ **then** *empty*

if $x \neq empty \wedge tail(x) \neq empty$ **then** $add(head(x)\ but.last(tail(x)))$

computes a copy of x with the last list element removed. *but.last* is a 1-bounded algorithm with 1-difference algorithm (obtained after recursion elimination)

function $\Delta^1 but.last(x : list) : bool \Leftarrow$

if $x = empty$ **then** *false*

if $x \neq empty$ **then** *true* .

function *bubblesort*(x : *list*) : *list* $\Leftarrow$

if $x = empty$ **then** *empty*

if $x \neq empty$ **then**

$add(last(bubble(x))\ bubblesort(but.last(bubble(x))))$.

The termination hypothesis for *bubblesort* is computed as

$\forall x : list \quad x \neq empty \rightarrow \Delta^1 but.last(bubble(\boldsymbol{x})) \vee \Delta^1 bubble(\boldsymbol{x})$.[20]

A.2. Selection sort

function *replace*(n, m : *number* x : *list*) : *list* $\Leftarrow$

if $x = empty$ **then** *empty*

if $x \neq empty \wedge head(x) = n$ **then** $add(m\ tail(x))$

if $x \neq empty \wedge head(x) \neq n$ **then** $add(head(x)\ replace(n\ m\ tail(x)))$

replaces the leftmost occurrence of n in x with m. *replace* is a 3-bounded algorithm with 3-difference algorithm (obtained after recursion elimination)

function $\Delta^3 replace(n, m : number\ x : list) : bool \Leftarrow$ **if** TRUE **then** *false* .

function *selectsort*(x : *list*) : *list* $\Leftarrow$

if $x = empty$ **then** *empty*

if $x \neq empty \wedge head(x) = minimum(x)$ **then**

$add(head(x)\ selectsort(tail(x)))$

if $x \neq empty \wedge head(x) \neq minimum(x)$ **then**

$add(minimum(x)$

$selectsort(replace(minimum(x)\ head(x)\ tail(x))))$.

[20] Bold italic symbols denote term representatives as here $\boldsymbol{x}$ abbreviates $add(head(x)\ tail(x))$.

The termination hypotheses for *selectsort* are computed as

$$\forall x : list \quad x \neq empty \wedge head(x) = minimum(x) \rightarrow \text{TRUE} ,$$
$$\forall x : list \quad x \neq empty \wedge head(x) \neq minimum(x) \rightarrow$$
$$\Delta^3 replace(minimum(x)\ head(x)\ tail(x)) \vee \text{TRUE} .$$

A.3. Minsort

function *delete.minimum*(x : *list*) : *list* $\Leftarrow$
  if $x = empty$ **then** *empty*
  if $x \neq empty \wedge tail(x) = empty$ **then** *empty*
  if $x \neq empty \wedge tail(x) \neq empty \wedge head(x) \leq head(tail(x))$ **then**
    add(*head*(*tail*(x)) *delete.minimum*(*add*(*head*(x) *tail*(*tail*(x)))))
  if $x \neq empty \wedge tail(x) \neq empty \wedge head(x) > head(tail(x))$ **then**
    add(*head*(x) *delete.minimum*(*tail*(x)))

deletes a minimal element from x and returns a permutation of the remaining list. *delete.minimum* is a 1-bounded algorithm with 1-difference algorithm (obtained after recursion elimination)

function Δ^1*delete.minimum*(x : *list*) : *bool* $\Leftarrow$
  if $x = empty$ **then** *false*
  if $x \neq empty$ **then** *true* .

function *minsort*(x : *list*) : *list* $\Leftarrow$
  if $x = empty$ **then** *empty*
  if $x \neq empty$ **then**
    add(*minimum*(x) *minsort*(*delete.minimum*(x))) .

The termination hypothesis for *minsort* is computed as

$$\forall x : list \quad x \neq empty \rightarrow \Delta^1 delete.minimum(x) .$$

A.4. Quicksort

function *smaller*(n : *number* x : *list*) : *list* $\Leftarrow$
  if $x = empty$ **then** *empty*
  if $x \neq empty \wedge n < head(x)$ **then** *smaller*(n *tail*(x))
  if $x \neq empty \wedge n \geq head(x)$ **then** *add*(*head*(x) *smaller*(n *tail*(x)))

computes a copy of x with all list elements greater than n removed. *smaller* is a 2-bounded algorithm with 2-difference algorithm

function $\Delta^2 smaller(n : number\ x : list) : bool \Leftarrow$

if $x = empty$ **then** *false*

if $x \neq empty \wedge n < head(x)$ **then** *true*

if $x \neq empty \wedge n \geqslant head(x)$ **then** $\Delta^2 smaller(n\ tail(x))$,

i.e. $\Delta^2 smaller(n\ x)$ returns *true* iff some element of x is greater than n.

function $larger(n : number\ x : list) : list \Leftarrow$

if $x = empty$ **then** *empty*

if $x \neq empty \wedge n < head(x)$ **then** $add(head(x)\ larger(n\ tail(x)))$

if $x \neq empty \wedge n \geqslant head(x)$ **then** $larger(n\ tail(x))$

computes a copy of x with all list elements less than or equal n removed. *larger* is a 2-bounded algorithm with 2-difference algorithm

function $\Delta^2 larger(n : number\ x : list) : bool \Leftarrow$

if $x = empty$ **then** *false*

if $x \neq empty \wedge n < head(x)$ **then** $\Delta^2 larger(n\ tail(x))$

if $x \neq empty \wedge n \geqslant head(x)$ **then** *true* ,

i.e. $\Delta^2 larger(n\ x)$ returns *true* iff some list element of x is less than or equal n.

function $quicksort(x : list) : list \Leftarrow$

if $x = empty$ **then** *empty*

if $x \neq empty$ **then**

$append(quicksort(smaller(head(x)\ tail(x))$

$add(head(x)\ quicksort(larger(head(x)\ tail(x)))))$.

The termination hypotheses for *quicksort* are computed as

$\forall x : list \quad x \neq empty \rightarrow \Delta^2 smaller(head(x)\ tail(x)) \vee \text{TRUE}$,

$\forall x : list \quad x \neq empty \rightarrow \Delta^2 larger(head(x)\ tail(x)) \vee \text{TRUE}$.

A.5. Mergesort

function $distribute.odd(x : list) : list \Leftarrow$

if $x = empty$ **then** *empty*

if $x \neq empty \wedge tail(x) = empty$ **then** x

if $x \neq empty \wedge tail(x) \neq empty$ **then**

$add(head(x)\ distribute.odd(tail(tail(x))))$

computes a copy of x with all elements on even positions removed. *distribute.odd* is a 1-bounded algorithm with 1-difference algorithm

function $\Delta^1 distribute.odd(x : list) : bool \Leftarrow$

if $x = empty$ **then** *false*

if $x \neq empty \wedge tail(x) = empty$ **then** *false*

if $x \neq empty \wedge tail(x) \neq empty$ **then** *true* ,

i.e. $\Delta^1 distribute.odd(x)$ returns *true* iff x has at least two elements.

function $distribute.even(x : list) : list \Leftarrow$

if $x = empty$ **then** *empty*

if $x \neq empty \wedge tail(x) = empty$ **then** *empty*

if $x \neq empty \wedge tail(x) \neq empty$ **then**

$add(head(tail(x))\ distribute.even(tail(tail(x))))$

computes a copy of x with all elements on odd positions removed. *distribute.even* is a 1-bounded algorithm with 1-difference algorithm

function $\Delta^1 distribute.even(x : list) : bool \Leftarrow$

if $x = empty$ **then** *false*

if $x \neq empty$ **then** *true* ,

i.e. $\Delta^1 distribute.even(x)$ returns *true* iff x has at least one element.

function $mergesort(x : list) : list \Leftarrow$

if $x = empty$ **then** *empty*

if $x \neq empty \wedge tail(x) = empty$ **then** x

if $x \neq empty \wedge tail(x) \neq empty$ **then**

$merge(mergesort(distribute.odd(x))$

$mergesort(distribute.even(x)))$.

The termination hypotheses for *mergesort* are computed as

$\forall x : list \quad x \neq empty \wedge tail(x) \neq empty \rightarrow \Delta^1 distribute.odd(x)$,

$\forall x : list \quad x \neq empty \wedge tail(x) \neq empty \rightarrow \Delta^1 distribute.even(x)$.

A.6. Heapsort

structure $tip\ node(left : tree\ key : number\ right : tree) : tree$,

function $pop(h : tree) : tree \Leftarrow$

if $h = tip$ **then** *tip*

if $h \neq tip \wedge left(h) = tip$ **then** *tip*

if $h \neq tip \wedge left(h) \neq tip \wedge depth(left(h)) > depth(right(h))$ **then**

$node(pop(left(h))\ key(h)\ right(h))$

if $h \neq tip \wedge left(h) \neq tip \wedge depth(left(h)) \leqslant depth(right(h))$ **then**
$node(left(h)\ key(h)\ pop(right(h)))$

computes a copy of h with the bottom node removed (if h is a heap). *pop* is a 1-bounded algorithm with 1-difference algorithm (after recursion elimination)

function $\Delta^1 pop(h : tree) : bool \Leftarrow$
if $h = tip$ **then** *false*
if $h \neq tip$ **then** *true* .

function $swap(h : tree) : tree \Leftarrow$
if $h = tip$ **then** tip
if $h \neq tip \wedge left(h) = tip$ **then** tip
if $h \neq tip \wedge left(h) \neq tip \wedge depth(left(h)) > depth(right(h))$ **then**
$node(pop(left(h))\ bottom(h)\ right(h))$
if $h \neq tip \wedge left(h) \neq tip \wedge depth(left(h)) \leqslant depth(right(h))$ **then**
$node(left(h)\ bottom(h)\ pop(right(h)))$

replaces the key in the root of h by the bottom key of h and removes the bottom node from h (if h is a heap). *swap* is a 1-bounded algorithm with 1-difference algorithm

function $\Delta^1 swap(h : tree) : bool \Leftarrow$
if $h = tip$ **then** *false*
if $h \neq tip \wedge left(h) = tip$ **then** *true*
if $h \neq tip \wedge left(h) \neq tip$
$\wedge\ depth(left(h)) > depth(right(h))$ **then**
$\Delta^1 pop(\boldsymbol{left(h)})$
if $h \neq tip \wedge left(h) \neq tip$
$\wedge\ depth(left(h)) \leqslant depth(right(h))$ **then**
$\Delta^1 pop(right(h))$,

i.e. $\Delta^1 swap(h)$ returns *true* iff h has at least one node.

function $sift(h : tree) : tree \Leftarrow$
if $h = tip$ **then** tip
if $h \neq tip \wedge left(h) = tip$ **then** h
if $h \neq tip \wedge left(h) \neq tip$
$\wedge\ key(h) = min.key(h\ left(h)\ right(h))$ **then** h
if $h \neq tip \wedge left(h) \neq tip$
$\wedge\ key(left(h)) = min.key(h\ left(h)\ right(h))$ **then**

$$node(sift(node(left(left(h))\ key(h)\ right(left(h))))$$
$$key(left(h))\ right(h))$$
$$\textbf{if } h \neq tip \wedge left(h) \neq tip$$
$$\wedge\ key(right(h)) = min.key(h\ left(h)\ right(h))\ \textbf{then}$$
$$node(left(h)\ key(right(h))$$
$$sift(node(left(right(h))\ key(h)\ right(right(h)))))$$

computes a heap consisting of all keys in h provided $left(h)$ and $right(h)$ are heaps and h has heap structure. *sift* is a 1-bounded algorithm with 1-difference algorithm (obtained after recursion elimination)

$$\textbf{function } \Delta^1 sift(h : tree) : bool \Leftarrow$$
$$\textbf{if } \text{TRUE } \textbf{then } false\ .$$
$$\textbf{function } heapsort(h : tree) : list \Leftarrow$$
$$\textbf{if } h = tip \textbf{ then } empty$$
$$\textbf{if } h \neq tip \textbf{ then } add(key(h)\ heapsort(sift(swap(h))))$$

computes a sorted list consisting of all keys in h if h is a heap. The termination hypothesis for *heapsort* is computed as

$$\forall h : tree \quad h \neq tip \rightarrow \Delta^1 sift(swap(h)) \vee \Delta^1 swap(h)\ .$$

Appendix B

We summarize the development of the estimation calculus presented in Section 4 with the following definition:

Definition B.1 (*Estimation calculus*). For an admissible specification S and a family $\Gamma(S)$ of argument-bounded functions in S, the estimation calculus is given as:

(1) *Language*: Estimation formulas, i.e. expressions for the form $\langle t \leqslant_{\#} r, \Delta \rangle$ such that t and r are terms from $\mathcal{T}(\Sigma, \mathcal{V})_s$ and Δ is a quantifier-free formula.

(2) *Axioms and inference rules* (estimation rules):

(a) *identity*:

$$\frac{}{\langle t \leqslant_{\#} t, \text{FALSE} \rangle} \quad \text{for all } t\ ;$$

(b) *equivalence*: for all irreflexive constructors $ircons_1$ and $ircons_2$

$$\frac{}{\langle ircons_1(t_1 \ldots t_n) \leqslant_{\#} ircons_2(r_1 \ldots r_m), \text{FALSE} \rangle} \quad \text{for all } t_i, r_j\ ;$$

(c) *strong estimation*: for all irreflexive constructors *ircons* and all reflexive constructors *rcons*

$$\frac{}{\langle ircons(t_1 \ldots t_n) \leqslant_{\#} rcons(r_1 \ldots r_m), \text{TRUE}\rangle} \quad \text{for all } t_i, r_j ;$$

(d) *strong embedding*: for all reflexive constructors *rcons*

$$\frac{\langle t \leqslant_{\#} r_k, \Delta\rangle}{\langle t \leqslant_{\#} rcons(r_1 \ldots r_k \ldots r_m), \text{TRUE}\rangle} \quad \text{for all } t, r_j, \Delta ;$$

(e) *argument estimation*: for all $f \in \Gamma_p(S)$

$$\frac{\langle t_p \leqslant_{\#} r, \Delta\rangle}{\langle f(t_1 \ldots t_n) \leqslant_{\#} r, \Delta^p f(t_1 \ldots t_n) \vee \Delta\rangle} \quad \text{for all } t_i, r, \Delta ;$$

(f) *weak embedding*: for all reflexive constructors *rcons* with reflexive argument positions $j_1, \ldots, j_h$

$$\frac{\langle t_{j_1} \leqslant_{\#} r_{j_1}, \Delta_{j_1}\rangle, \ldots, \langle t_{j_h} \leqslant_{\#} r_{j_h}, \Delta_{j_h}\rangle}{\langle rcons(t_1 \ldots t_n) \leqslant_{\#} rcons(r_1 \ldots r_n), \Delta_{j_1} \vee \cdots \vee \Delta_{j_h}\rangle} \quad \text{for all } t_i, r_i, \Delta_i ;$$

(g) *minimum*: for all irreflexive constructors *ircons* and for all reflexive constructors $rcons_j$ with selectors $sel_{j_1}, \ldots, sel_{j_{h(j)}}$

$$\frac{}{\begin{array}{l}\langle ircons(t_1 \ldots t_n) \leqslant_{\#} r , \\ \quad r = rcons_1(sel_{1_1}(r) \ldots sel_{1_{h(1)}}(r)) \vee \cdots \vee \\ \quad r = rcons_n(sel_{n_1}(r) \ldots sel_{n_{h(n)}}(r))\rangle\end{array}} \quad \text{for all } t_i, r .$$

(3) A *deduction* of $\langle t_n \leqslant_{\#} r_n, \Delta_n\rangle$ in the E-calculus is a sequence of estimation formulas $\langle t_1 \leqslant_{\#} r_1, \Delta_1\rangle, \ldots, \langle t_n \leqslant_{\#} r_n, \Delta_n\rangle$ such that each estimation formula $\langle t_k \leqslant_{\#} r_k, \Delta_k\rangle$ in the sequence either is an axiom or can be inferred by one of the inference rules from some estimation formulas preceding $\langle t_k \leqslant_{\#} r_k, \Delta_k\rangle$ in the sequence.

Theorem 5.1. *A normal algorithm* **function** $f(x_1 : s_1 \ldots x_n : s_n) : s \Leftarrow \cdots$ *strongly terminates in an admissible specification S if there exists some non-empty set* $P \subset \{1, \ldots, n\}$ *such that for each recursive call* $f(t_1 \ldots t_n)$ *in a case* "**if** φ **then** ..." *of f*

(1) $t_i \leqslant_{\Gamma(S)} x_i$ *for all* $i \in P$, *and*
(2) $[\forall x_1 : s_1 \ldots x_n : s_n \ \varphi \rightarrow \bigvee_{i \in P} \Delta_{\Gamma(S)}(t_i, x_i)] \in Th(S)$.

Proof. Let $<_f$ be the binary relation defined on $\mathcal{T}(\Sigma^c)_{s_1 \ldots s_n}$ as $(r_1 \ldots r_n) <_f (q_1 \ldots q_n)$ iff $\Sigma_{i \in P} \#_{s_i}(r_i) <_{\mathbb{N}} \Sigma_{i \in P} \#_{s_i}(q_i)$ (cf. Section 4.1). Obviously $<_f$ is well-founded and we prove that f strongly terminates with respect to $<_f$ (cf.

Definition 3.3). Since only normal algorithms are considered here we have to verify that for each $(q_1 \dots q_n) \in \mathcal{T}(\Sigma^c)_{s_1 \dots s_n}$ and for each recursive call $f(t_1 \dots t_n)$ in a case "**if** φ **then** ..." of f

$$\mathcal{I}[\![x_1 \dots x_n / q_1 \dots q_n]\!] \models \varphi \text{ implies}$$
$$\mathcal{I}[\![x_1 \dots x_n / q_1 \dots q_n]\!](t_1 \dots t_n) <_f (q_1 \dots q_n)\,, \tag{B.1}$$

where $\mathcal{I}$ is the standard model of S. Assume that $(q_1 \dots q_n) \in \mathcal{T}(\Sigma^c)_{s_1 \dots s_n}$ such that $\mathcal{I}[\![x_1 \dots x_n / q_1 \dots q_n]\!] \models \varphi$. Then we conclude from requirement (1) and Theorem 4.4 that

$$\#_{s_i}(\mathcal{I}[\![x_1 \dots x_n / q_1 \dots q_n]\!](t_i)) \leqslant_{\mathbb{N}} \#_{s_i}(q_i) \quad \text{for each } i \in P\,. \tag{B.2}$$

We obtain $\mathcal{I}[\![x_1 \dots x_n / q_1 \dots q_n]\!] \models \bigvee_{i \in P} \Delta_{\Gamma(S)}(t_i, x_i)$ by requirement (2), therefore $\mathcal{I}[\![x_1 \dots x_n / q_1 \dots q_n]\!] \models \Delta_{\Gamma(S)}(t_j, x_j)$ for *some* $j \in P$, and with Theorem 4.4

$$\#_{s_j}(\mathcal{I}[\![x_1 \dots x_n / q_1 \dots q_n]\!](t_j)) <_{\mathbb{N}} \#_{s_j}(q_j) \quad \text{for some } j \in P \tag{B.3}$$

must hold. We infer from (B.2) and (B.3)

$$\sum_{i \in P} \#_{s_i}(\mathcal{I}[\![x_1 \dots x_n / q_1 \dots q_n]\!](t_i)) <_{\mathbb{N}} \sum_{i \in P} \#_{s_i}(q_i) \tag{B.4}$$

and $\mathcal{I}[\![x_1 \dots x_n / q_1 \dots q_n]\!](t_1 \dots t_n) <_f (q_1 \dots q_n)$ is verified. □

Theorem 6.2. *If S is an admissible specification containing a p-bounded algorithm* **function** $g(x_1 : s_1 \dots x_n : s_n) : s \Leftarrow \cdots$ *and its p-difference algorithm* **function** $\Delta^p g(x_1 : s_1 \dots x_n : s_n) : bool \Leftarrow \cdots$, *then* $\mathit{TRUE}\langle g(x_1 \dots x_n) \leqslant_\# x_p,\ \Delta^p g(x_1 \dots x_n)\rangle$.

Proof. The proof is by induction according to the recursions in the p-bounded algorithm. We find for each non-recursive case "**if** φ_i **then** r_i" of g that $\vdash_{\Gamma(S)} \langle \lceil r_i \rceil \leqslant_\# \lceil x_p \rceil,\ \gamma_i \rangle$, hence by Theorem 4.4, Lemma 5.6 and the definitions of g and $\Delta^p g$

$$[\forall x_1 : s_1 \dots x_n : s_n\ \varphi_i \rightarrow g(x_1 \dots x_n) \leqslant_\# x_p] \in Th(S)\,, \tag{B.5}$$
$$[\forall x_1 : s_1 \dots x_n : s_n\ \varphi_i \rightarrow$$
$$(\Delta^p g(x_1 \dots x_n) \leftrightarrow g(x_1 \dots x_n) <_\# x_p)] \in Th(S)\,. \tag{B.6}$$

For each recursive case "**if** φ_i **then** r_i" and for each recursive call $g(t_1 \dots t_n)$ in r_i we use

$$\mathit{TRUE}\langle g(\lceil t_1 \rceil \dots \lceil t_n \rceil) \leqslant_\# \lceil t_p \rceil, \Delta^p g(t_1 \dots t_n)\rangle \tag{B.7}$$

as an induction hypothesis. For each term t and each formula Δ we find that (B.7) and $\mathit{TRUE}\langle \lceil t_p \rceil \leqslant_\# t,\ \Delta\rangle$ implies

$$\mathit{TRUE}\langle g(\lceil t_1 \rceil \dots \lceil t_n \rceil) \leqslant_\# t,\ \Delta^p g(t_1 \dots t_n) \vee \Delta\rangle\,. \tag{B.8}$$

Hence we have proved that

$$\frac{\langle \lceil t_p \rceil \leqslant_{\#} t, \Delta \rangle}{\langle g(\lceil t_1 \rceil \ldots \lceil t_n \rceil) \leqslant_{\#} t, \Delta^{p} g(t_1 \ldots t_n) \vee \Delta \rangle} \quad \text{for all } t, \Delta$$

is a sound estimation rule (relative to the recursive case under consideration). Consequently $\vdash_{\Gamma(S)+} \langle \lceil r_i \rceil \leqslant_{\#} \lceil x_p \rceil, \gamma_i \rangle$ implies $\mathit{TRUE}\langle \lceil r_i \rceil \leqslant_{\#} \lceil x_p \rceil, \gamma_i \rangle$ (cf. Theorem 4.4), and we conclude with Lemma 5.6 that (B.5) and (B.6) also hold for the recursive cases of g.

Using the case-completeness of g, i.e. $[\forall x_1 : s_1 \ldots x_n : s_n \; \varphi_1 \vee \cdots \vee \varphi_k] \in Th(S)$, the statement of the theorem is obtained from (B.5) and (B.6). □

References

[1] M. Baudinet, Proving termination properties of PROLOG programs: a semantic approach, in: *Proceedings Third Symposium on Logic in Computer Science* (1988) 336–347.

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

[3] R.S. Boyer and J S. Moore, The addition of bounded quantification and partial functions to a computational logic and its theorem prover, *J. Automated Reasoning* **4** (2) (1988) 117–172.

[4] R.M. Burstall, Program proving as hand simulation with a little induction, in: *Proceedings IFIP Congress 1974 Stockholm* (North-Holland, Amsterdam, 1974) 308–312.

[5] D.C. Cooper, Programs for mechanical program verification, in: B. Meltzer and D. Michie, eds., *Machine Intelligence* **6** (Edinburgh University Press, Edinburgh, 1971) 43–59.

[6] N. Dershowitz and Z. Manna, Proving termination with multiset orderings, *Commun. ACM* **22** (8) (1979) 465–476.

[7] R.W. Floyd, Assigning meaning to programs, in: J.T. Schwartz, ed., *Mathematical Aspects of Computer Science*, Proceedings Symposia in Applied Mathematics **19** (American Mathematical Society, Providence, RI, 1967) 19–32.

[8] C.A.R. Hoare, Algorithm 65—Find, *Commun. ACM* **12** (1961) 321.

[9] S. Katz and Z. Manna, A closer look at termination, *Acta Inf.* **5** (1975) 333–352.

[10] Z. Manna, Termination of algorithms, Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA (1968).

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

[12] Z. Manna, S. Ness and J. Vuillemin, Inductive methods for proving properties of programs, *Commun. ACM* **16** (8) (1973) 491–502.

[13] Z. Manna and P. Pnueli, Axiomatic approach to total correctness of programs, *Acta. Inf.* **3** (1974) 243–263.

[14] Z. Manna and R. Waldinger, Is "sometimes" sometimes better than "always"?, *Commun. ACM* **21** (2) (1978) 159–172.

[15] J. McCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart and M.I. Levin, *LISP 1.5 Programmer's Manual* (MIT Press, Cambridge, MA, 1962).

[16] J S. Moore, A mechanical proof of the termination of Takeuchi's function, *Inf. Process. Lett.* **9** (4) (1979) 176–181.

[17] L.C. Paulson, Proving termination of normalization functions for conditional expressions, *J. Automated Reasoning* **2** (1) (1986) 63–74.

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

[19] M. Protzen, Disproving conjectures, in: *Proceedings 11th International Conference on Automated Deduction, Saratoga Springs*, Lecture Notes in Artificial Intelligence **607** (Springer-Verlag, Berlin, 1992) 340–354.

[20] G.L. Steele, *Common Lisp—The Language* (Digital Press, Bedford, MA, 1984).
[21] J.D. Ullman and A. van Gelder, Efficient tests for top-down termination of logical rules, *J. ACM* **35** (2) (1988) 345–373.
[22] C. Walther, Automated termination proofs, Interner Bericht 17/88, Fakultät für Informatik, Universität Karlsruhe (1988).
[23] C. Walther, A digest of argument-bounded algorithms, Interner Bericht 18/88, Fakultät für Informatik, Universität Karlsruhe (1988).
[24] 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, Oxford, 1994).
[25] C. Walther, *Automatisierung von Terminierungsbeweisen* (Vieweg, Braunschweig, 1991).