# Automated Termination Analysis for Incompletely Defined Programs

Christoph Walther and Stephan Schweitzer

Fachgebiet Programmiermethodik
Technische Universität Darmstadt
`www.informatik.tu-darmstadt.de/pm/`
{`chr.walther,schweitz`}`@informatik.tu-darmstadt.de`

**Abstract.** Incompletely defined programs provide an elegant and easy way to write and to reason about programs which may halt with a run time error by throwing an exception or printing an error message, e.g. when attempting to divide by zero. Due to the presence of stuck computations, which arise when calling incompletely defined procedures with invalid arguments, we cannot use the method of argument bounded algorithms for proving termination by machine. We analyze the problem and present a solution to improve this termination analysis method so that it works for incompletely defined programs as well. Our technique of proving the termination of incompletely defined programs maintains performance as well as simplicity of the original method and proved successful by an implementation in the verification tool √`eriFun`.

## 1 Introduction

A central problem in the development of correct software is to verify that algorithms terminate. A non-terminating algorithm results in looping computations, hence machine resources are wasted if a given input is not in the domain of the function computed by the algorithm. 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.

Termination analysis is concerned with the synthesis and verification of *termination hypotheses*, i.e. proof obligations the truth of which entail the termination of the algorithm under consideration. In this paper, we are concerned with functional programs where several proposals exist for proving termination, e.g. [1],[2],[3],[4],[5],[6],[7],[8],[9],[10].

In the √`eriFun` system [11],[12],[13], a semi-automated verifier for functional programs, the method of *argument bounded algorithms* [10] is used and proved successful for verifying the termination of procedures by machine. Recently, √`eriFun` was upgraded to work for *incompletely defined* programs as well [14]. Those programs compute *partially determined* functions, i.e. functions which may yield (defined but) "unknown" results for some of their input arguments. However, unsound inferences will result if our termination proof procedure is

```
structure bool  <=  true, false
structure nat   <=  0, succ(pred:nat)
structure list  <=  empty, add(hd:nat,tl:list)
```

```
function minus(x,y:nat):nat <=          function remainder(x,y:nat):nat <=
if y=0                                  if y=0
  then x                                  then * || 0
  else if x=0                             else if y>x
         then * || 0                             then x
         else minus(pred(x),pred(y))             else remainder(minus(x,y),y)
       fi                                      fi
 fi                                     fi
```

**Fig. 1.** Data structures, incompletely and completely defined procedures

applied to incompletely defined procedures, because the method requires that only *totally determined* functions are computed by the procedures of a program. We therefore use *domain procedures* [14] to modify the method so that it can be soundly applied to prove the termination of incompletely defined programs too.

## 2 Completely Defined Programs

**Syntax** We use a programming language in which data structures are defined in the spirit of (free) algebraic data types. A data structure $s$ is defined by stipulating the *constructors* of the data structure as well as a *selector* for each argument position of a constructor. The set of all *constructor ground terms* built with the constructors of $s$ then defines the elements of the data structure $s$.

For example, truth values are represented by the set $\mathcal{T}(\{true, false\}) = \{true, false\}$ and the set of natural numbers is represented by the set $\mathcal{T}(\{0, succ\}) = \{0\ , succ(0), succ(succ(0)), \ldots\}$, both given by data structures `bool` and `nat` of Fig. 1.[1] Likewise, the data structure `list` of Fig. 1 represents the set of linear lists of natural numbers, with e.g. $add(succ(0), add(0, empty)) \in \mathcal{T}(\{0,\ succ, empty, add\})$. The *selectors* act as inverses to their constructors, since e.g. $hd(add(n, k)) = n$ and $tl(add(n, k)) = k$ is demanded. Each definition of a data structure $s$ implicitly introduces an equality symbol $\texttt{=}_s : s \times s \to bool$ (where $s \neq bool$) and a function symbol $if_s : bool \times s \times s \to s$ for conditionals.

A procedure, which operates on these data structures, is defined by giving the procedure name, say $f$, the formal parameters and the result type in the *procedure head*. The *procedure body* is given as a first-order term over the set of formal parameters, the function symbols already introduced by some data structures and other procedures plus the function symbol $f$ to allow recursive definitions, cf. Fig. 1 where "$*$ $\|$" in the procedure bodies should be ignored.

A finite list $P$ of data structure and procedure definitions—always beginning with the data structure definitions of `bool` and `nat` as given in Fig. 1—is called

[1] $\mathcal{T}(\Sigma, \mathcal{V})_s$ is the set of *terms* of type $s$, $\mathcal{T}(\Sigma)_s = \mathcal{T}(\Sigma, \emptyset)_s$, and $\mathcal{CL}(\Sigma, \mathcal{V})$ is the set of *clauses* over a signature $\Sigma$ for function symbols and a set $\mathcal{V}$ of variable symbols.

a *completely defined functional program.* $\Sigma(P)$ is the set of all function symbols introduced by the data structures and procedures of $P$, and $\Sigma(P)^c \subset \Sigma(P)$ is the set of all constructor function symbols given by the data structures of $P$.

**Semantics and Termination** Given a (completely defined functional) program $P$, an interpreter $eval_P$ for $P$ evaluates terms of $\mathcal{T}(\Sigma(P))$ to "values", i.e. terms of $\mathcal{T}(\Sigma(P)^c)$. The interpreter computes calls $f(t_1, \ldots, t_n)$ of a procedure `function` $f(x_1{:}s_1, \ldots, x_n{:}s_n){:}s$ `<=` $R_f$ *call-by-value*, i.e. by replacing each formal parameter $x_i$ in the procedure body $R_f$ by the computation $t'_i$ of the actual parameter $t_i$, and then continuing with the computation of the instantiated procedure body obtained. The interpreter also respects the definitions of the data structures by computing, for instance, *false* for $0$=$succ(t)$ and $q$ for $pred(succ(t))$, provided $eval_P(t) = q$ for some $q \in \mathcal{T}(\Sigma(P)^c)$. For selectors $sel : s \to s'$ applied to constructors *cons* to which they do not belong, so-called *witness terms* $\omega_{sel}[x] \in \mathcal{T}(\Sigma(P), \{x\})_{s'}$ with $x \in \mathcal{V}_s$ are assigned in $P$ to *sel*, and we define $eval_P(sel(cons(q_1, \ldots, q_n)) := eval_P(\omega_{sel}[cons(q_1, \ldots, q_n)])$. Hence e.g. $eval_P(tl(empty)) = empty$ and $eval_P(hd(empty)) = 0$ if $\omega_{\mathtt{tl}}[x] := x$ and $\omega_{\mathtt{hd}}[x] := 0$ for the selectors of data structure `list`, cf. Fig. 1. By these definitions, our programming language is provided with an *eager* semantics.

Since $P$ may contain non-terminating procedures, $eval_P$ is a *partial* mapping only, i.e. $eval_P : \mathcal{T}(\Sigma(P)) \mapsto \mathcal{T}(\Sigma(P)^c)$, and we define [14]:

**Definition 1.** (Termination) *A procedure* `function` $f(x_1{:}s_1, \ldots, x_n{:}s_n){:}s$ `<=` $\ldots$ *of a completely defined program $P$* terminates in *$P$ iff $eval_P(f(q_1, \ldots, q_n))$ $\in \mathcal{T}(\Sigma(P)^c)$ for all $q_i \in \mathcal{T}(\Sigma(P)^c)_{s_i}$. $P$ terminates iff (i) each procedure of $P$ terminates in $P$ and (ii) $eval_P(\omega_{sel}[q]) \in \mathcal{T}(\Sigma(P)^c)$ for each selector $sel : s \to s'$ and for all $q \in \mathcal{T}(\Sigma(P)^c)_s$.*

## 3 Incompletely Defined Programs

**Motivation** Incompletely defined programs provide an elegant and easy way to specify and to verify statements about (recursive) *partial* functions with *decidable* domain [14]. Incompletely defined programs compute *partially determined* functions, also called *loosely specified* or *underspecified* functions in the literature.

A *total* function is called *partially determined* iff the result of a function application is *indetermined* for some arguments, called *stuck* arguments. *Partial* functions $\phi : M \mapsto N$ with domain $dom_\phi$ can be represented by *partially determined* but *total* functions $\widehat{\phi} : M \to N$ by stipulating $\widehat{\phi}(m) := \phi(m)$ for each $m \in dom_\phi$ but demanding $\widehat{\phi}(m) \in N$ for each $m \in M \setminus dom_\phi$ only, being silent about which $n \in N$ exactly is assigned to $\widehat{\phi}(m)$. The elements of $M \setminus dom_\phi$ are the *stuck arguments* of $\widehat{\phi}$, and $\widehat{\phi}(m)$ is *indetermined* iff $m$ is a stuck argument. Examples of partially determined functions are *quotient* and *remainder* with stuck arguments of form $(m, 0)$, list processing functions, like *head*, *last* and *minimum* with the empty list as stuck argument, and so on. If $dom_\phi$ is *decidable* and the completion $\widehat{\phi}$ of $\phi$ is *recursive*, $\widehat{\phi}$ can be computed by an incompletely defined procedure $\wp_{\widehat{\phi}}$ so that properties of $\phi$ can be verified by reasoning about $\wp_{\widehat{\phi}}$, using some verifier based on a logic of *total* functions.

An incompletely defined program is obtained by giving an incomplete case analysis in a procedure or using a specific symbol, say `*`, to denote an indetermined result. Such programs can be implemented by causing a *runtime error* or throwing an *exception* when called with a stuck argument, e.g. upon the attempt to divide by zero. When focussing on functional programs—as we do here—the interpreter of the programming language responds by returning a ground term $r \notin \mathcal{T}(\Sigma(P)^{c})$ when called with a stuck argument, and we call such a result $r$ a *stuck computation.*

**Syntax** A data structure $s$ is incompletely defined by not stipulating witness terms for the selectors of $s$. For defining a procedure $f$ incompletely, we allow the use of a wildcard `*` to stipulate the result when calling $f$ with a stuck argument. E.g., procedure `minus` of Fig. 1 is incompletely defined if "|| `0`" is ignored in the procedure body, and the value of $minus(n, m)$ is only determined if $n \geq m$. Also procedure `remainder` of Fig. 1 is incompletely defined when ignoring "|| `0`", and the value of $remainder(n, m)$ is determined iff $m \neq 0$.

Formally, we assume a constant symbol $*_s \notin \Sigma(P)$ for each data structure $s$ in a functional program $P$, and we demand upon the extension of $P$ by a new procedure `function` $f(x_1{:}s_1, \ldots, x_n{:}s_n){:}s$ `<=` $R_f$, that $R_f \in \mathcal{T}(\Sigma(P) \cup \{f, *_s\}, \{x_1, \ldots, x_n\})$ be $*$-*correct*, i.e. $R_f = *$ or `*` is only used as a (direct) argument in the *alternatives* of an *if*-conditional.

**Termination and Semantics** For defining the termination and in turn the semantics of an incompletely defined program $P$, the notion of a *fair completion* $P'$ of $P$ is needed [14]: $\widehat{P}$ is the set of all fair completions of $P$, where each $P' \in \widehat{P}$ is a completely defined program containing each data structure $s$ which is given in $P$ plus the witness terms for the selectors of $s$. $P'$ also contains a procedure `function` $f(x_1{:}s_1, \ldots, x_n{:}s_n){:}s$ `<=` $R'_f$ for each procedure `function` $f(x_1{:}s_1, \ldots, x_n{:}s_n){:}s$ `<=` $R_f$ in $P$. The procedure body $R'_f$ is obtained from $R_f$ by replacing each occurrence of `*` in $R_f$ by some term from $\mathcal{T}(\Sigma(P'), \{x_1, \ldots, x_n\})$, where it does not matter whether different occurrences of `*` are replaced by the same or by different terms. In addition, the *fairness requirement* demands that the termination of procedure $f$ in $P'$ not be spoiled *just because* procedure $f$ is completed by a non-terminating result in a $*$-case or a non-terminating witness term is assigned to a selector.

For example, a fair completion of a program containing the incompletely defined procedure `minus` of Fig. 1 may contain the completely defined procedure `minus`. Also the occurrence of `*` in procedure `minus` may be replaced by $succ(y)$ or 13 or $minus(x, pred(y))$ etc. in a fair completion $P'$ of $P$. But we may not replace `*` by $minus(x, y)$ or by $loop(y)$, where `function loop(x:nat):nat <= succ(loop(x))` is a procedure of $P'$, as this violates the fairness requirement.

**Definition 2.** (Termination) *A procedure* `function` $f(x_1{:}s_1, \ldots, x_n{:}s_n){:}s$ `<=` $R_f$ *of an incompletely defined program* $P$ terminates *in* $P$ *iff for each* $P' \in \widehat{P}$ *procedure* `function` $f(x_1{:}s_1, \ldots, x_n{:}s_n){:}s$ `<=` $R'_f$ *of* $P'$ *terminates in* $P'$. $P$ *terminates iff each procedure of* $P$ *terminates in* $P$.

**Definition 3.** (Standard Model $\mathcal{M}_P$, Theory $Th_P$) *Let $P$ be an incompletely defined and terminating program. Then a* standard model $\mathcal{M}_P$ *of $P$ is a $\Sigma(P)$-algebra $\mathcal{M}_P = (\mathcal{T}(\Sigma(P)^c), \phi)$ such that some $P' \in \widehat{P}$ exists with $\phi_f(q_1, \ldots, q_n) = eval_{P'}(f(q_1, \ldots, q_n))$ for all $f \in \Sigma(P)_{s_1,\ldots,s_n,s}$ and all $q_i \in \mathcal{T}(\Sigma(P)^c)_{s_i}$. The theory $Th_P$ of $P$ is defined as $\{\varphi \in \mathcal{F}(\Sigma(P), \mathcal{V}) \mid \mathcal{M}_P \models \varphi$ for* each *standard model $\mathcal{M}_P$ of $P\}$. A verification system for $P$ is* sound *iff $\varphi \in Th_P$ for each $\varphi \in \mathcal{F}(\Sigma(P), \mathcal{V})$ verified by the system.*[2]

By Definition 3, incompletely defined procedures (and selectors) are understood as *loose specifications* of total functions. The standard models for incompletely defined (and terminating) programs differ only in the interpretation of functions applied to stuck arguments, but coincide for all other function applications. So $Th_P$ is *incomplete*, i.e. neither $\varphi \in Th_P$ nor $\neg\varphi \in Th_P$ for some $\varphi \in \mathcal{F}(\Sigma(P), \mathcal{V})$, whereas $Th_P$ is complete for completely defined programs $P$.

**Verification** When we formulate proof obligations of form "$\varphi \in Th_P$" for incompletely defined programs $P$ in the following, we assume the availability of some "sound verification system for $P$" to compute a proof for $\varphi$. We can do so as *(i)* several verifiers for functional programs exist, see e.g. [15],[16],[17] for references, and *(ii)* (most) logics used for the verification of terminating and completely defined programs can be applied without profound modifications to verify terminating but incompletely defined programs as well, see [14].

**Computation** To implement our programming language, we also have to define an interpreter $eval_P$ for *incompletely* defined programs $P$. As the formal definition of $eval_P$ does not matter here, we refer to [14] for details. For the purpose of this paper, it is enough to know that for each $t \in \mathcal{T}(\Sigma(P))$

$$eval_P(t) \in \mathcal{T}(\Sigma(P)^c) \iff \Big(eval_P(t) = eval_{P'}(t) \text{ for each } P' \in \widehat{P}\Big) . \qquad (1)$$

## 4 Termination Analysis with Argument Bounded Algorithms

Argument bounded algorithms are the key concept for the automated termination analysis proposed in [10]. The method has been implemented and proved successful in verification tools, [18],[19],[20],[21],[12], and provided the base for further developments of termination analysis [22],[23],[24],[3],[4],[25],[26],[27],[9] as well. Termination analysis with argument bounded algorithms is based on the syntactic estimation $\geqslant_{\Gamma,C}$ of terms, where selector and procedure calls are estimated above by some argument(s) of the call.

A total and totally determined function $\phi : \mathcal{T}(\Sigma(P)^c)_{s_1} \times \ldots \times \mathcal{T}(\Sigma(P)^c)_{s_n} \to \mathcal{T}(\Sigma(P)^c)_{s_p}$ is called *p-bounded* iff $p \in \{1, \ldots, n\}$ and $q_p \geqslant_{\#} \phi(q_1, \ldots, q_n)$ for all $q_i \in \mathcal{T}(\Sigma(P)^c)_{s_i}$.[3] A function $\phi$ is called *argument bounded* iff it is $p$-bounded for

[2] $\mathcal{F}(\Sigma, \mathcal{V})$ is the set of *closed formulas* over $\Sigma$ and $\mathcal{V}$.

[3] $>_{\#}$ is the *size order* comparing constructor ground terms $q \in \mathcal{T}(\Sigma(P)^c)_s$ by the number $\#_s(q)$ of *reflexive* $s$-constructors in $q$, and $q \geqslant_{\#} r$ abbreviates $q >_{\#} r$ *or* $q =_{\#} r$. A function symbol $h : s_1 \times \ldots \times s_n \to s$ is *reflexive* iff $s = s_i$ for some $i$.

```
function half(x:nat):nat <=
if x=0
  then 0
  else if pred(x)=0 then * else succ(half(pred(pred(x)))) fi
fi

function log(x:nat):nat <=
if x=0
  then *
  else if pred(x)=0
         then 0
         else if even(x) then succ(log(half(x))) else * fi
       fi
fi
```

**Fig. 2.** Incompletely defined procedures (cont.)

some $p \in \{1,\ldots,n\}$. Each $p$-bounded function $\phi$ is associated with a so-called (total and totally determined) *p-difference function* $\delta^p_\phi : \mathcal{T}(\Sigma(P)^c)_{s_1} \times \ldots \times \mathcal{T}(\Sigma(P)^c)_{s_n} \to \{\mathit{true}, \mathit{false}\}$ which satisfies $\delta^p_\phi(q_1,\ldots,q_n) \Leftrightarrow q_p >_\# \phi(q_1,\ldots,q_n)$ for all $q_i \in \mathcal{T}(\Sigma(P)^c)_{s_i}$.

Given a family $\Gamma = (\Gamma_p)_{p\in\mathbb{N}}$ of sets of $p$-bounded function symbols $g \in \Sigma(P)$ which denote $p$-bounded functions $\phi$, and the function symbols $\Delta^p_g$ denoting their $p$-difference functions $\delta^p_\phi$, inequalities can be proved by the *estimation calculus* [10] (called $E$-calculus for short). The formulas of the $E$-calculus are called *estimation pairs* $\langle \Delta, E\rangle$, where $\Delta \in \mathcal{CL}(\Sigma(P),\mathcal{V})$, consisting mainly of atoms of form $\Delta^p_g(\ldots)$, and $E$ is a finite set of expressions of form $r \succcurlyeq t$ with $r,t \in \mathcal{T}(\Sigma(P),\mathcal{V})_s$. The $E$-calculus is decidable and is sound in the sense that

$$\begin{array}{ll} \textit{(i)} & [\forall x_1{:}s_1,\ldots,x_n{:}s_n.\ \bigwedge C \to r \geqslant_\# t] \in \mathit{Th}_P \text{ , and} \\ \textit{(ii)} & [\forall x_1{:}s_1,\ldots,x_n{:}s_n.\ \bigwedge C \to (\bigvee \Delta \leftrightarrow r >_\# t)] \in \mathit{Th}_P \end{array} \tag{2}$$

hold if $\vdash_{\Gamma,C} \langle \Delta, r \succcurlyeq t\rangle$, i.e. if $\langle \Delta, \{r \succcurlyeq t\}\rangle$ is a *theorem* of the $E$-calculus, where $C \in \mathcal{CL}(\Sigma(P),\mathcal{V})$ and $x_i \in \mathcal{V}_{s_i}$ are the variables in $C, r$ and $t$.

The $E$-calculus is used *(i)* to generate *termination hypotheses* for completely defined procedures, *(ii)* to *test* whether a (terminating and completely defined) procedure `function` $g(x_1{:}s_1,\ldots,x_n{:}s_n){:}s_p$ <= $R_g$ computes a $p$-bounded function $\phi$, and (if so) *(iii)* to *synthesize* a *p-difference procedure* `function` $\Delta^p_g(x_1{:}s_1, \ldots, x_n{:}s_n){:}\mathit{bool}$ <= $R_{\Delta^p_g}$ which computes the $p$-difference function $\delta^p_\phi$ for $\phi$.

## 5 Incompletely Defined Argument Bounded Procedures

Consider the completely defined and 1-bounded procedure `minus` from Fig. 1, and assume that the incompletely defined procedure `half` of Fig. 2 is fairly completed by stipulating $\mathit{half}(1) := 0$ or $\mathit{half}(1) := 1$. Then `half` is 1-bounded too, and the following estimation proof can be obtained:

$$\texttt{x} \geqslant_\Gamma \texttt{pred(x)} \geqslant_\Gamma \texttt{half(pred(x))} \geqslant_\Gamma \texttt{minus(half(pred(x)),succ(y))} \tag{3}$$

Here $\geqslant_\Gamma$ abbreviates $\geqslant_{\Gamma,\emptyset}$, where $\geqslant_{\Gamma,C}$ is the *syntactic estimation relation* defined by $r \geqslant_{\Gamma,C} t$ iff $\vdash_{\Gamma,C} \langle \Delta, r \succcurlyeq t\rangle$ for some $\Delta \in \mathcal{CL}(\Sigma(P), \mathcal{V})$. However, in an incompletely defined program, where `minus` and `half` are given as in Figs. 1 and 2, the result of a function applied to a stuck argument is not determined. Therefore `pred`, `half` and `minus` fail to be argument bounded, hence an estimation proof like (3) cannot be obtained. This problem does not exist for completely defined programs, as we may stipulate any result we like for a function applied to a "don't-care" argument. Hence we may in particular use results which do not spoil the 1-boundedness of the above functions, and may define e.g. $\mathit{pred}(0) := \mathit{minus}(0, n) := \mathit{half}(1) := 0$.

Since the function computed by an incompletely defined procedure fails to be argument bounded for stuck arguments, the notion of argument boundedness as given in [10] has to be generalized:

**Definition 4.** (p-Boundedness) *Let $P$ be an incompletely defined program. Then each* reflexive *selector of a data structure in $P$ is 1*-bounded*. A procedure* `function` $f(x_1{:}s_1, \ldots, x_n{:}s_n){:}s$ <= $R_f$ *of $P$ is $p$*-bounded *iff*

1. $p \in \{1, \ldots, n\}$ *with* $s_p = s$,
2. $\Sigma(r) \cap \{*, f\} = \emptyset$ *for some result term $r$ in the procedure body $R_f$, and*
3. $x_p \geqslant^{\oplus}_{\Gamma,C_r} r$ *for each result term $r \neq *$ appearing under clause $C_r$ in $R_f$.*[4]

$\Gamma := \bigcup_{p\in\mathbb{N}} \Gamma_p$ *is the set of argument bounded function symbols in $P$, where each $\Gamma_p$ is the set of* $p$-bounded function symbols *in $P$. $\Gamma_p$ is defined as the smallest subset of $\Sigma(P)$ satisfying* (i) *rsel* $\in \Gamma_1$ *for each reflexive selector rsel of a data structure in $P$ and* (ii) $f \in \Gamma_p$ *for each $p$-bounded procedure $f$ in $P$.*

Requirement (3) of Definition 4 allows to ignore indetermined result terms when testing for argument boundedness and is the only relevant modification of the original definition. Requirement (2) is only an optimization, as procedures computing indetermined results only cannot contribute to the termination analysis.

For example, all reflexive selectors of the data structures given in Fig. 1 as well as the incompletely defined procedures `minus`, `remainder`, `half` and `log` of Figs. 1 and 2 now are 1-bounded, and `remainder` is 2-bounded too.

## 6 Domain Procedures

Having generalized the notion of argument boundedness by Definition 4, estimation proofs now can be obtained for incompletely defined programs too. However, such an estimation proof may be unsound, because the functions involved fail to be argument bounded for stuck arguments.

For instance, the estimation proof (3) is unsound, because a standard model exists which assigns 1 to $\mathit{pred}(0)$ and 2 to $\mathit{half}(1)$ as well as to $\mathit{minus}(0, 1)$. Hence $0 \not\geq \mathit{pred}(0)$, $2 \geq 1 \not\geq \mathit{half}(1)$ and $3 \geq 2 \geq 1 \not\geq \mathit{minus}(1, 2)$.

[4] We write $\vdash^{\oplus}_{\Gamma,C_r} \langle \ldots \rangle$ to denote the existence of an estimation proof which already may use the Argument Estimation rule (5) of Definition 5 for each recursive call $f(t_1, \ldots, t_n)$ in $r$. See [10] for a justification.

As a remedy, we have to exclude the applications of reflexive selectors and argument bounded procedures to stuck arguments in an estimation proof. To this effect, we use *domain procedures* which have been developed in [14] for reasoning about stuck computations explicitly: Domain procedures are given for non-procedure function symbols $\neq$"*if*"by stipulating `function` $\nabla_{=}(x{:}s, y{:}s){:}bool$ <= $true$, `function` $\nabla_{sel_i}(x{:}s){:}bool$ <= $?cons(x)$ and `function` $\nabla_{cons}(x_1{:}s_1, \ldots, x_n{:}s_n){:}bool$ <= $true$ for the selectors $sel_i$ and constructors $cons$ of a data structure definition `structure` $s$ <= $\ldots, cons(sel_1{:}s_1, \ldots, sel_n{:}s_n), \ldots$, where $?cons(x)$ abbreviates $x = cons(sel_1(x), \ldots, sel_n(x))$. For a procedure `function` $f(x_1{:}s_1, \ldots, x_n{:}s_n){:}s$ <= ..., a domain procedure `function` $\nabla_f(x_1{:}s_1, \ldots, x_n{:}s_n){:}bool$ <= ... can be uniformly synthesized.

As proved in [14], *(i)* each domain procedure $\nabla_f$ terminates iff its "mother" procedure $f$ terminates, *(ii)* computes a totally determined function, and *(iii)* equivalently characterizes whether the computation of a call of procedure $f$ results in a stuck computation, i.e. for all $q_i \in \mathcal{T}(\Sigma(P)^c)_{s_i}$

$$eval_P(f(q_1, \ldots, q_n)) \in \mathcal{T}(\Sigma(P)^c) \text{ iff } eval_P(\nabla_f(q_1, \ldots, q_n)) = true.$$

Since domain procedures are tail recursive and compute a truth value, the optimization techniques developed in [10] for *difference procedures* apply to domain procedures as well: Having generated a domain procedure $\nabla_f$, the body of $\nabla_f$ is *simplified* in a first optimization step, and then it is tried to *eliminate recursive calls* in the simplified procedure body. Recursion elimination is particularly important, because proofs are more easily obtained if the procedures "called" in a statement have no unnecessary recursive calls.

*Example 1.*

(i)
```
function ∇minus(x,y:nat):nat <=
 if y=0
   then true
   else if x=0 then false else ∇minus(pred(x),pred(y)) fi
  fi
```

is computed as the optimized domain procedure for the incompletely defined procedure `minus` from Fig. 1, and we find $\nabla_{\texttt{minus}}(n, m) = true$ iff $n \geq m$.

(ii)
```
function ∇remainder(x:nat, y:nat):bool <=
 if y=0 then false else true fi
```

is computed as the optimized domain procedure for the incompletely defined procedure `remainder` from Fig. 1, and $\nabla_{\texttt{remainder}}(n, m) = true$ iff $m \neq 0$.

(iii)
```
function ∇half(x:nat):nat <=
 if x=0
   then true
   else if pred(x)=0 then false else ∇half(pred(pred(x))) fi
  fi
```

is computed as the optimized domain procedure for procedure `half` from Fig. 2, and we find $\nabla_{\texttt{half}}(n) = true$ iff $n$ is *even*.

(iv)
```
function ∇_log(x:nat):nat <=
  if x=0
    then false
    else if pred(x)=0
           then true
           else if even(x) then ∇_log(half(x)) else false fi
         fi
   fi
```

is computed as the optimized domain procedure for procedure `log` from Fig. 2, and we find $\nabla_{\texttt{log}}(n) = \mathit{true}$ iff $n = 2^k$ for some $k \in \mathbb{N}$. □

To optimize domain procedure $\nabla_{\texttt{remainder}}$, recursion elimination is required, where the generated recursion elimination formulas are trivial to verify. All domain procedures of Example 1 are optimal because all recursive calls which survived recursion elimination are required.

From now on we assume that each incompletely defined program $P$ contains a domain procedure `function` $\nabla_f$ for each function symbol $f \in \Sigma(P)$ with $f \neq \mathit{if}$ and $f \neq \nabla_g$, where $g$ is any function symbol in $\Sigma(P)$.[5]

## 7 Estimation Proofs in Incompletely Defined Programs

Domain procedures provide the necessary prerequisite to exclude the applications of reflexive selectors and argument bounded procedures to stuck arguments in an estimation proof. For example, to guarantee soundness of the estimation proof (3) we only have to demand

$$\nabla_{\texttt{pred}}(\texttt{x}) \wedge \nabla_{\texttt{half}}(\texttt{pred(x)}) \wedge \nabla_{\texttt{minus}}(\texttt{half(pred(x)),succ(y)}) \; . \tag{4}$$

Requirement (4) expresses $x \neq 0$, $x{-}1$ is *even* and $(x{-}1)/2 \geq 1 + y$, thus excluding the unsound estimations from Section 6. In the general case, we scan an estimation proof

$$t_1 \geqslant_{\Gamma,C} t_2 \geqslant_{\Gamma,C} \cdots \geqslant_{\Gamma,C} t_{n-1} \geqslant_{\Gamma,C} t_n \tag{5}$$

step by step and create a procedure call $\nabla_f(r_1, \ldots, r_m)$ for each estimation step $t_i \geqslant_{\Gamma,C} t_{i+1}$ with $t_{i+1} = f(r_1, \ldots, r_m)$ and $f \in \Gamma$, where $1 \leq i \leq n-1$. These procedure calls are collected in a set $\nabla \in \mathcal{CL}(\Sigma(P), \mathcal{V})$, called the *determination clause* of the estimation proof (5).

To this effect, the estimation calculus from [10] is refined:

**Definition 5.** (pE-Calculus) *Let $P$ be an incompletely defined program, let $\Gamma$ be a family of argument bounded function symbols in $P$, and let $C \in \mathcal{CL}(\Sigma(P), \mathcal{V})$, called the* context clause*. Assume further that* ircons, ircons$_1$ *and* ircons$_2$ *are (not necessarily different) irreflexive constructors, and that* rcons, rcons$_1, \ldots,$ rcons$_n$ *are (not necessarily different) reflexive constructors of some data structures in $P$. Then the* partial estimation calculus *(*pE-calculus*) is given by:*

[5] This means that we do not need domain procedures of domain procedures.

1. ***Language*** Estimation triples, *i.e. expressions of form* $\langle \nabla, \Delta, E \rangle$ *where* $\nabla, \Delta \in \mathcal{CL}(\Sigma(P), \mathcal{V})$ *and* $E \subset \{r \succcurlyeq t \mid r, t \in \mathcal{T}(\Sigma(P), \mathcal{V})_s\}$ *with* $|E| < \infty$.
2. ***Inference Rules*** (Estimation Rules) [6]

*Identity*

$$(1) \quad \frac{\langle \nabla, \Delta, E \uplus \{t \succcurlyeq t\} \rangle}{\langle \nabla, \Delta, E \rangle}$$

*Equivalence*

$$(2) \quad \frac{\langle \nabla, \Delta, E \uplus \{r \succcurlyeq t\} \rangle}{\langle \nabla, \Delta, E \rangle}\ , \textit{ if } C \vdash \textit{?ircons}_2(r) \textit{ and } C \vdash \textit{?ircons}_1(t)$$

*Strong Estimation*

$$(3) \quad \frac{\langle \nabla, \Delta, E \uplus \{r \succcurlyeq t\} \rangle}{\langle \nabla, \Delta \cup \{true\}, E \rangle}\ , \textit{ if } C \vdash \textit{?rcons}(r) \textit{ and } C \vdash \textit{?ircons}(t)$$

*Strong Embedding*

$$(4) \quad \frac{\langle \nabla, \Delta, E \uplus \{r \succcurlyeq t\} \rangle}{\langle \nabla, \Delta \cup \{true\}, E \cup \{SEL_k(r) \succcurlyeq t\} \rangle}\ , \textit{ if } \begin{cases} C \vdash \textit{?rcons}(r), \textit{ and} \\ k \textit{ is a reflexive argument} \\ \textit{position of rcons} \end{cases}$$

*Argument Estimation*

$$(5) \quad \frac{\langle \nabla, \Delta, E \uplus \{r \succcurlyeq f(t_1, \ldots, t_n)\} \rangle}{\langle \nabla \cup \{\nabla_f(t_1, \ldots, t_n)\}, \Delta \cup \{\Delta_f^p(t_1, \ldots, t_n)\}, E \cup \{r \succcurlyeq t_p\} \rangle}\ , \textit{ if } f \in \Gamma_p$$

*Weak Embedding*

$$(6) \quad \frac{\langle \nabla, \Delta, E \uplus \{r \succcurlyeq t\} \rangle}{\langle \nabla, \Delta, E \cup \bigcup_{i=1}^{h} \{SEL_{j_i}(r) \succcurlyeq SEL_{j_i}(t)\} \rangle}\ , \textit{ if } \begin{cases} C \vdash \textit{?rcons}(r), \\ C \vdash \textit{?rcons}(t), \textit{ and} \\ j_1, \ldots, j_h \textit{ are all} \\ \textit{reflexive argument} \\ \textit{positions of rcons,} \end{cases}$$

*Minimum*

$$(7) \quad \frac{\langle \nabla, \Delta, E \uplus \{r \succcurlyeq t\} \rangle}{\langle \nabla, \Delta \cup \bigcup_{i=1}^{k} \{\textit{?rcons}_i(r)\}, E \rangle}\ , \textit{ if } \begin{cases} C \vdash \textit{?ircons}(t), \textit{ and} \\ \textit{rcons}_1, \ldots, \textit{rcons}_k \textit{ are } \text{all} \\ \textit{reflexive constructors of } s \end{cases}$$

3. ***Deduction*** *A* deduction *of* $\langle \nabla_n, \Delta_n, E_n \rangle$ *from* $\langle \nabla_1, \Delta_1, E_1 \rangle$ *is a finite sequence* $\langle \nabla_1, \Delta_1, E_1 \rangle, \ldots, \langle \nabla_n, \Delta_n, E_n \rangle$ *of estimation triples such that* $n \geq 1$ *and* $\langle \nabla_i, \Delta_i, E_i \rangle \Rrightarrow_{\Gamma,C} \langle \nabla_{i+1}, \Delta_{i+1}, E_{i+1} \rangle$, *i.e.* $\langle \nabla_{i+1}, \Delta_{i+1}, E_{i+1} \rangle$ *results from* $\langle \nabla_i, \Delta_i, E_i \rangle$ *by an application of some estimation rule for each* $i < n$.
   $r \geqslant_{\Gamma,C} t$ *abbreviates* $\vdash_{\Gamma,C} \langle \nabla, \Delta, r \succcurlyeq t \rangle$ *for some* $\nabla, \Delta \in \mathcal{CL}(\Sigma(P), \mathcal{V})$, *where* $\vdash_{\Gamma,C} \langle \nabla, \Delta, r \succcurlyeq t \rangle$ *denotes the existence of an* estimation proof *for* $r \succcurlyeq t$ *with* determination clause $\nabla$ *and* difference equivalent $\Delta$, *given by*

$$\vdash_{\Gamma,C} \langle \nabla, \Delta, r \succcurlyeq t \rangle \iff \langle \emptyset, \emptyset, \{r \succcurlyeq t\} \rangle \Rrightarrow^{+}_{\Gamma,C} \langle \nabla, \Delta, \emptyset \rangle \ .$$

[6] We write $C \vdash \textit{?cons}_i(r)$ iff *(i)* $r = \textit{cons}_i(\ldots)$ or *(ii)* $\textit{?cons}_i(r) \in C$ or *(iii)* $\{\neg\textit{?cons}_j(r) \mid j \in \{1, \ldots, n\} \setminus \{i\}\} \subset C$ for a data structure $s$ with constructors $\textit{cons}_1, \ldots, \textit{cons}_n$. $SEL_k(r)$ stands for $r_k$ if $r = \textit{rcons}(\ldots, r_k, \ldots)$, and abbreviates $\textit{sel}_k(r)$ otherwise.

**Theorem 1.** (Soundness of the pE-calculus) *Let $P$ be an incompletely defined and terminating program, and let* $\vdash_{\Gamma,C} \langle \nabla, \Delta, r \succcurlyeq t \rangle$ *where* $x_1, \ldots, x_n$ *with* $x_i \in \mathcal{V}_{s_i}$ *are all variable symbols in* $C, r$ *and* $t$. *Then*

1. $[\forall x_1{:}s_1, \ldots, x_n{:}s_n.\ \bigwedge\nabla \to (\bigwedge C \to r \geqslant_{\#} t)] \in Th_P$, *and*
2. $[\forall x_1{:}s_1, \ldots, x_n{:}s_n.\ \bigwedge\nabla \to (\bigwedge C \to (\bigvee\Delta \leftrightarrow r >_{\#} t))] \in Th_P$.[7]

By Theorem 1, the soundness of a *pE*-deduction is relativized by the domain clause $\nabla$ inferred. This means that the soundness statements of (2) in Section 4 hold for incompletely defined programs only if each literal of $\nabla$ is true, i.e. if the *absence of stuck computations* is guaranteed. E.g., we now may obtain the *pE*-deduction $\langle\emptyset,\ \emptyset,\ \{\texttt{x} \succcurlyeq \texttt{minus(half(pred(x)),succ(y))}\}\rangle \Rrightarrow^{+}_{\Gamma} \langle\{\nabla_{\texttt{pred}}\texttt{(x)},\ \nabla_{\texttt{half}}\texttt{(pred(x))},\ \nabla_{\texttt{minus}}\texttt{(half(pred(x)), succ(y))}\},\ \{\Delta^1_{\texttt{minus}}\texttt{(half(pred(x)),succ(y))},\ \Delta^1_{\texttt{half}}(\texttt{pred(x)}),\ \Delta^1_{\texttt{pred}}(\texttt{x})\},\ \emptyset\rangle$.

A proof procedure for the *pE*-calculus is easily obtained, because the set of theorems of the *pE*-calculus is decidable:

**Theorem 2.** (Decidability of pE-deductions) *Let $P$ be an incompletely defined program, let* $\mathbb{E} = \{r \succcurlyeq t \mid r, t \in \mathcal{T}(\Sigma(P), \mathcal{V})_s\}$, *and let* $M = \{\langle \nabla, \Delta, E\rangle \mid \nabla, \Delta \in \mathcal{CL}(\Sigma(P), \mathcal{V})$ *and* $E \subset \mathbb{E}$ *with* $|E| < \infty\}$. *Then*

1. $\{\langle \nabla, \Delta, \{r \succcurlyeq t\}\rangle \in M \mid \vdash_{\Gamma,C} \langle \nabla, \Delta, r \succcurlyeq t\rangle\}$ *is decidable, and*
2. $r \geqslant_{\Gamma,C} t$ *is decidable.*

## 8 Synthesis of Difference Procedures

The *pE*-calculus is used similarly to the *E*-calculus in [10] to recognize *p*-boundedness of a procedure `function` $f(x_1{:}s_1, \ldots, x_n{:}s_n){:}s <= R_f$ and to synthesize a *p*-difference procedure `function` $\Delta^p_f(x_1{:}s_1, \ldots, x_n{:}s_n){:}bool <= R_{\Delta^p_f}$ for procedure $f$. But we have to modify the synthesis process slightly to cope with the $*$-symbol which may occur in the procedure bodies $R_f$.

We define `*` as the result term of $R_{\Delta^p_f}$ under a clause $C$ whenever `*` appears as the result term under this clause in $R_f$. Consequently, a *p*-difference procedure $\Delta^p_f$ is incompletely defined iff its "mother" procedure $f$ is.

**Definition 6.** (*p*-Difference Procedures) *Let $P$ be an incompletely defined program. Then each* reflexive *selector $sel \in \{sel_1, \ldots, sel_n\}$ of a data structure definition* `structure` $s <= \ldots, cons(sel_1{:}s_1, \ldots, sel_n{:}s_n), \ldots$ *in $P$ is associated with the* 1*-difference procedure*

$$\texttt{function}\ \Delta^1_{sel}(x{:}s){:}bool <= \texttt{if}\ \ ?cons(x)\ \texttt{then true else}\ *\ \texttt{fi}\ .$$

*Each p*-bounded *procedure* `function` $f(x_1{:}s_1, \ldots, x_n{:}s_n){:}s <= R_f$ *of $P$ is associated with some p-difference procedure*

$$\texttt{function}\ \Delta^p_f(x_1{:}s_1, \ldots, x_n{:}s_n){:}bool <= R_{\Delta^p_f}$$

[7] We refer to [28] for omitted proofs.

*such that $R_{\Delta_f^p}$ is obtained from $R_f$ by keeping each result term $r$ with $r = *$ and by replacing each result term $r$ with $r \neq *$ which appears under some clause $C_r$ in $R_f$ by* `OR`$(\Delta_r)$*, where $\vdash^{\oplus}_{\Gamma,C_r} \langle \nabla_r, \Delta_r, x_p \not\geq r\rangle$.*[8]

**Theorem 3.** *Let $P$ be an incompletely defined program, let $f \in \Sigma(P)_{s_1,\ldots,s_n,s}$ be p-bounded, and let* `function` $\Delta_f^p$ *denote a p-difference procedure of $f$. Then for all $q_i \in \mathcal{T}(\Sigma(P)^c)_{s_i}$ and for all $P' \in \widehat{P}$*

1. $eval_{P'}(f(q_1,\ldots,q_n)) \in \mathcal{T}(\Sigma(P)^c) \Leftrightarrow eval_{P'}(\Delta_f^p(q_1,\ldots,q_n)) \in \{true, false\}$,
2. $eval_P(\nabla_f(q_1,\ldots,q_n)) = true \implies eval_P(\Delta_f^p(q_1,\ldots,q_n)) \in \{true, false\}$,
3. *$P$ terminates* $\Rightarrow [\forall x_1{:}s_1,\ldots,x_n{:}s_n.\ \nabla_f(x_1,\ldots,x_n) \to x_p \geqslant_{\#} f(x_1,\ldots,x_n) \wedge (\Delta_f^p(x_1,\ldots,x_n) \leftrightarrow x_p >_{\#} f(x_1,\ldots,x_n))] \in Th_P$.

By Theorem 3(1), a difference procedure terminates iff its “mother” procedure terminates. By Theorem 3(2), $\nabla_f(q_1,\ldots,q_n)$ entails that computation of $\Delta_f^p(q_1,\ldots,q_n)$ does not get stuck. We therefore abandon with generating a domain procedure $\nabla_{\Delta_f^p}$ for a difference procedure $\Delta_f^p$ but use the domain procedure $\nabla_f$ of its “mother” procedure $f$ instead. Finally by Theorem 3(3), a $p$-bounded procedure $f$ computes a $p$-bounded function and a $p$-difference procedure is *sound*, i.e. it represents an *equivalent* requirement for a procedure call $f(q_1,\ldots,q_n)$ being *strictly* bounded above by its $p^{th}$ argument $q_p$, provided the computation of $f(q_1,\ldots,q_n)$ does not get stuck.

After their synthesis, the difference procedures are optimized by simplification and recursion elimination as defined in [10].

*Example 2.*

(i) `function` $\Delta^1_{\texttt{minus}}$`(x,y:nat):bool <=`

```
  if y=0 then false else if x=0 then * else true fi fi
```

is computed as the optimized 1-difference procedure for the incompletely defined procedure `minus` of Fig. 1. Hence $\Delta^1_{\texttt{minus}}(n,m) = true$ iff $m \neq 0 \neq n$ and $\Delta^1_{\texttt{minus}}(n,m) = false$ iff $m = 0$.

(ii) `function` $\Delta^1_{\texttt{remainder}}$`(x:nat, y:nat):bool <=`

```
   if y=0 then * else if y>x then false else true fi fi
```

is computed as the optimized 1-difference procedure for the incompletely defined procedure `remainder` of Fig. 1. Hence $\Delta^1_{\texttt{remainder}}(n,m) = true$ iff $n \geq m \neq 0$ and $\Delta^1_{\texttt{remainder}}(n,m) = false$ iff $n < m$. Since `remainder` is 2-bounded too, we also obtain the optimized 2-difference procedure

`function` $\Delta^2_{\texttt{remainder}}$`(x:nat, y:nat):bool <=`

```
if y=0 then * else true fi
```

and $\Delta^2_{\texttt{remainder}}(n,m) = true$ iff $m \neq 0$ and $\Delta^2_{\texttt{remainder}}(n,m) \neq false$.

(iii) `function` $\Delta^1_{\texttt{half}}$`(x:nat):bool <=`

```
   if x=0 then false else if pred(x)=0 then * else true fi fi
```

is computed as the optimized 1-difference procedure for procedure `half` of Fig. 2, hence $\Delta^1_{\texttt{half}}(n) = true$ iff $n \geq 2$ and $\Delta^1_{\texttt{half}}(n) = false$ iff $n = 0$.

[8] `OR`$(C)$ denotes the disjunction of the elements in $C$ represented by *if*-conditionals.

(iv) `function` $\Delta^1_{\texttt{log}}$`(x:nat):bool <=`

```
   if x=0
     then *
     else if pred(x)=0 then true
                       else if even(x) then true else * fi fi fi
```

is computed as the optimized 1-difference procedure for procedure `log` of Fig. 2, and $\Delta^1_{\texttt{log}}(n) = true$ iff $n = 1$ or $n \neq 0$ is *even* and $\Delta^1_{\texttt{log}}(n) \neq false$. □

## 9 Generating Termination Hypotheses

Using the $pE$-calculus of Definition 5, we adjust the synthesis of termination hypotheses as defined in [10] to work also for incompletely defined procedures:

**Definition 7.** (Termination Hypotheses) *Let* `function` $f(x_1{:}s_1,\ldots,x_n{:}s_n){:}s$ `<=` $R_f$ *be a procedure of an incompletely defined program* $P$*, let* $f(t_1,\ldots,t_n)$ *be a recursive call which appears under some clause* $C$ *in* $R_f$*, and let* $\emptyset \neq \mathcal{P} \subset \{1,\ldots,n\}$ *such that* $\vdash_{\Gamma,C} \langle \nabla_i, \Delta_i, x_i \succcurlyeq t_i\rangle$ *for each* $i \in \mathcal{P}$*. Then a termination hypothesis* $\tau_f^{\mathcal{P}}$ *of procedure* $f$ *is defined as*

$$\tau_f^{\mathcal{P}} = \left[\forall x_1{:}s_1,\ldots,x_n{:}s_n.\ \bigwedge C \to \bigwedge_{i\in\mathcal{P}} \left(\bigwedge \nabla_i\right) \wedge \bigvee_{i\in\mathcal{P}} \left(\bigvee \Delta_i\right)\right] \ . \qquad (6)$$

**Theorem 4.** *Let* $P = P_0 \oplus \langle$`function` $f(x_1{:}s_1,\ldots,x_n{:}s_n){:}s$ `<=` $R_f\rangle$ *be an incompletely defined program such that* $P_0$ *terminates. Then procedure* $f$ *terminates in* $P$ *if some non-empty* $\mathcal{P} \subset \{1,\ldots,n\}$ *exists such that* $\tau_f^{\mathcal{P}} \in Th_{P_0}$ *for* each *termination hypothesis* $\tau_f^{\mathcal{P}}$*.*

*Example 3.* (i) We compute $\tau^{\{1\}}_{\texttt{minus}} = \tau^{\{2\}}_{\texttt{minus}} = [\forall x, y{:}nat.\ y \neq 0 \wedge x \neq 0 \to true \wedge true]$ for the incompletely defined procedure `minus` of Fig. 1.

(ii) We compute $\tau^{\{2\}}_{\texttt{remainder}} = [\forall x, y{:}nat.\ y \neq 0 \wedge y \not> x \to false]$ and $\tau^{\{1\}}_{\texttt{remainder}} = [\forall x, y{:}nat.\ y \neq 0 \wedge y \not> x \to \nabla_{\texttt{minus}}(x, y) \wedge \Delta^1_{\texttt{minus}}(x, y)]$ for the incompletely defined procedure `remainder` of Fig. 1.

(iii) We compute $\tau^{\{1\}}_{\texttt{half}} = [\forall x{:}nat.\ x \neq 0 \wedge pred(x) \neq 0 \to true \wedge true]$ for procedure `half` of Fig. 2.

(iv) We compute $\tau^{\{1\}}_{\texttt{log}} = [\forall x{:}nat.\ x \neq 0 \wedge pred(x) \neq 0 \wedge even(x) \to \nabla_{\texttt{half}}(x) \wedge \Delta^1_{\texttt{half}}(x)]$ for procedure `log` of Fig. 2. □

## 10 Summary and Conclusion

Our termination proof procedure for incompletely defined programs is implemented in the √`eriFun` system in the following way:

Upon definition of a data structure $s$, the domain procedures $\nabla_{sel}$ are generated for each selector $sel$ of $s$, each *reflexive* selector $sel'$ of $s$ is inserted into $\Gamma_1$ and the 1-difference procedures $\Delta^1_{sel'}$ for $sel'$ are generated, cf. Sections 6 and 8.

Upon definition of a procedure $f$, the $pE$-calculus is called to compute the termination hypotheses for procedure $f$, cf. Definition 7. Then the system tries to verify all termination hypotheses and—if successful—computes the domain procedure $\nabla_f$ and optimizes it, cf. Section 6. Next the $pE$-calculus is called again to test whether procedure $f$ is $p$-bounded for some argument position $p$, cf. Definition 4. For each such $p$ passing the test, the system computes the $p$-difference procedure $\Delta_f^p$ and optimizes it, cf. Section 8. Finally, $f$ is inserted into $\Gamma_p$ to be available for subsequent termination proofs, i.e. for proving the termination of procedures $g$ which use procedure $f$ in recursive $g$-calls.

Argument bounded algorithms proved as a useful concept to verify the termination of functional procedures by machine, easing the burden of a system user significantly as termination functions need to be supplied less frequently when defining procedures. Incompletely defined programs provide an elegant and easy way to write and to reason about programs which may halt with a run time error. Our proposal unifies the benefits of both approaches without sacrificing performance or simplicity, neither when proving termination nor when reasoning about programs.

Our method of proving the termination of incompletely defined programs automatically has proved successful in √`eriFun` [12],[13], a semi-automated verifier for functional programs. The √`eriFun` system is available from the web [11].

**Acknowledgement** We are grateful to Markus Aderhold and Andreas Schlosser for useful comments as well as to Jürgen Giesl for thorough and fruitful discussions and for constructive criticism on a draft of this paper.

## References

1. Boyer, R.S., Moore, J.S.: A Computational Logic. Acad. Press, NY (1979)
2. Giesl, J.: Termination Analysis for Functional Programs using Term Orderings. In: Proc. of the 2nd Intern. Static Analysis Symposium (SAS-95). Volume 983 of Lecture Notes in Artifical Intelligence., Glasgow, Springer (1995) 154–171
3. Giesl, J.: Termination of Nested and Mutually Recursive Algorithms. Journal of Automated Reasoning **19** (1997) 1–29
4. Giesl, J., Walther, C., Brauburger, J.: Termination Analysis for Functional Programs. In Bibel, W., Schmitt, P., eds.: Automated Deduction - A Basis for Applications. Volume 3. Kluwer Acad. Publ., Dordrecht (1998) 135–164
5. Kamareddine, F., Monin, F.: An extension of an automated termination method of recursive functions. Intern. J. of Found. of Comp. Sc. **13** (2002) 361–386
6. Manoury, P., Simonot, M.: Automatizing Termination Proofs of Recursively Defined Functions. Theoretical Computer Science **135** (1994) 319–343
7. Monin, F., Simonot, M.: An Ordinal Measure based Procedure for Termination of Functions. Theoretical Computer Science **254** (2001) 63–94
8. Nielson, F., Nielson, H.R.: Termination Analysis based on Operational Semantics. Technical report, Aarhus University, Denmark (1995)
9. Sengler, C.: Termination of Algorithms over Non−Freely Generated Data Types. In McRobbie, M.A., Slaney, J.K., eds.: Proc. of the 13th Inter. Conf. on Automated Deduction (CADE-13). Volume 1104 of Lecture Notes in Artifical Intelligence., New Brunswick, NJ, Springer (1996) 121–136

10. Walther, C.: On Proving the Termination of Algorithms by Machine. Artificial Intelligence **71** (1994) 101–157
11. http://www.verifun.de.
12. Walther, C., Schweitzer, S.: About $\surd$`eriFun`. In Baader, F., ed.: Proc. of the 19th Inter. Conf. on Automated Deduction (CADE-19). Volume 2741 of Lecture Notes in Artifical Intelligence., Miami Beach, Springer (2003) 322–327
13. Walther, C., Schweitzer, S.: Verification in the Classroom. Journal of Automated Reasoning - Special Issue on Automated Reasoning and Theorem Proving in Education **32** (2004) 35–73
14. Walther, C., Schweitzer, S.: Reasoning about Incompletely Defined Programs. Technical Report VFR 04/02, Programmiermethodik, Technische Universität Darmstadt (2004)
15. Bundy, A.: The Automation of Proof by Mathematical Induction. In Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. Volume I. Elsevier (2001) 845–911
16. Comon, H.: Inductionless Induction. In Robinson, A., Voronkov, A., eds.: Handb. of Autom. Reasoning. Volume I. Elsevier (2001) 913–962
17. Walther, C.: Mathematical Induction. In Gabbay, D., Hogger, C., Robinson, J., eds.: Handbook of Logic in Artificial Intelligence and Logic Programming. Volume 2. Oxford University Press, Oxford (1994) 127–228
18. Autexier, S., Hutter, D., Langenstein, B., Mantel, H., Rock, G., Schairer, A., Stephan, W., Vogt, R., Wolpers, A.: VSE: Formal Methods Meet Industrial Needs. Intern. J. on Software Tools for Technology Transfer **3** (2000) 66–77
19. Autexier, S., Hutter, D., Mantel, H., Schairer, A.: inka 5.0 - A Logic Voyager. In Ganzinger, H., ed.: Proc. 16th Inter. Conf. on Autom. Deduction (CADE-16). Volume 1632 of Lect. Notes in Artif. Intell., Trento, Springer (1999) 207–211
20. Hutter, D., Langenstein, B., Sengler, C., Siekmann, J., Stephan, W., Wolpers, A.: Verification Support Environment (VSE). High Integrity Syst. **1** (1996) 523–530
21. Hutter, D., Sengler, C.: INKA: The Next Generation. In McRobbie, M., J.Slaney, eds.: Proc. 13th Inter. Conf. on Autom. Deduction (CADE-13). Volume 1104 of Lect. Notes in Artif. Intell., New Brunswick, Springer (1996) 288–292
22. Brauburger, J.: Automatic Termination Analysis for Partial Functions using Polynomial Orderings. In: Proc. of the 4th Intern. Static Analysis Symposium (SAS-97). Volume 1302 of Lect. Notes in Artif. Intell., Paris, Springer (1997) 330–344
23. Brauburger, J., Giesl, J.: Approximating the Domains of Functional and Imperative Programs. Science of Computer Programming **35** (1999) 113–136
24. Giesl, J.: Automated Termination Proofs with Measure Functions. In: Proc. of the 19th Annual German Conf. on Artifical Intelligence (KI-95). Volume 981 of Lecture Notes in Artifical Intelligence., Bielefeld, Springer (1995) 149–160
25. Gow, J., Bundy, A., Green, I.: Extensions to the Estimation Calculus. In Ganzinger, H., McAllester, D.A., Voronkov, A., eds.: Proc. of the 6th Inter. Conf. on Logic Progr. and Autom. Reasoning (LPAR-6). Volume 1705 of Lect. Notes in Artif. Intelligence., Tbilisi, Georgia, Springer (1999) 258–272
26. Hutter, D.: Using Rippling to Prove the Termination of Algorithms. Technical Report RR 97-03, DFKI, Saarbrücken (1997)
27. McAllester, D., Arkoudas, K.: Walther Recursion. In McRobbie, M.A., Slaney, J.K., eds.: Proc. of the 13th Inter. Conf. on Autom. Deduction. Volume 1104 of Lect. Notes in Artif. Intell., New Brunswick, NJ, Springer (1996) 643–657
28. Walther, C., Schweitzer, S.: Automated Termination Analysis for Incompletely Defined Programs. Technical Report VFR 04/03, Programmiermethodik, Technische Universität Darmstadt (2004)