Automated Termination Analysis for Programs with Second-Order Recursion

Markus Aderhold

Technische Universität Darmstadt, Germany
aderhold@informatik.tu-darmstadt.de

Abstract. Many algorithms on data structures such as *terms* (finitely branching trees) are naturally implemented by second-order recursion: A first-order procedure f passes itself as an argument to a second-order procedure like *map*, *every*, *foldl*, *foldr*, etc. to recursively apply f to the direct subterms of a term. We present a method for automated termination analysis of such procedures. It extends the approach of *argument-bounded functions* (i) by inspecting type components and (ii) by adding a facility to take care of second-order recursion. Our method has been implemented and automatically solves the examples considered in the literature. This improves the state of the art of inductive theorem provers, which (without our approach) require user interaction even for termination proofs of simple second-order recursive procedures.

1 Introduction

Functional programs frequently use higher-order procedures such as *map* and *every* that expect functions as parameters [7,12]. For instance, *map* applies a function to each element of a list and returns the list of the result values. Similarly, $every(p, k)$ yields *true* iff $p(x)$ evaluates to *true* for all elements x of a list k. If a procedure f calls a higher-order procedure g using f as an argument for g, e. g., $g(f, \ldots)$, we say that f is defined by *higher-order recursion* [8,14].

In this paper, we consider the automated termination analysis of functional programs that may use *second-order recursion*.[1] Typical examples arise in algorithms on finitely branching trees such as terms; e. g., applying a substitution to a term or collecting variables in a term. Termination analysis for such programs is non-trivial: In the higher-order theorem provers Isabelle [8,10,14] and PVS [11], the user needs to assist the system to prove termination in these cases. In contrast, the method we propose solves typical termination problems automatically. Furthermore, our method supplies information that allows a theorem prover to generate useful induction axioms for proofs about such programs.

Figure 1 shows an example program. In Fig. 1(a), data types *bool*, $\mathbb{N}$, and $list[@A]$ are defined by enumerating the respective data constructors *true*, *false*, 0, *succ*, $\emptyset$, and "::". Each argument position of a data constructor is assigned a

[1] As in [3], we define the order $o(\tau)$ of base types τ like $\mathbb{N}$ or $list[\mathbb{N}]$ as 0; the order of a functional type $\tau_1 \times \ldots \times \tau_n \to \tau$ is $1 + \max_i o(\tau_i)$ for a base type τ.

J. Esparza and R. Majumdar (Eds.): TACAS 2010, LNCS 6015, pp. 221–235, 2010.

(a)
```
structure bool <= true, false
structure ℕ <= 0, succ(pred : ℕ)
structure list[@A] <= ∅, ::(hd : @A, tl : list[@A])
procedure last(k : list[@A]) : @A <=
assume k≠∅; if tl(k)=∅ then hd(k) else last(tl(k)) end
```

(b)
```
structure variable.symbol <= variable(varID : ℕ)
structure function.symbol <= func(funcID : ℕ)
structure term <=
  var(vsym : variable.symbol),
  apply(fsym : function.symbol, args : list[term])
procedure every(p : @A → bool, k : list[@A]) : bool <=
if k=∅ then true else if p(hd(k)) then every(p, tl(k)) else false end end
procedure groundterm(t : term) : bool <=
if ?var(t) then false else every(groundterm, args(t)) end
```

Fig. 1. A functional program with (a) the first-order procedure *last* and (b) the second-order procedure *every* and second-order recursion in procedure *groundterm*

selector function; e. g., selector *pred* denotes the predecessor function. Expressions of the form $?cons(t)$ check if t denotes a value of the form $cons(\ldots)$. In Fig. 1(b), procedure *every* is a second-order procedure that gets a first-order function p as argument. Procedure *groundterm* uses *second-order recursion* to check if a term t (modeled by data type *term*) does not contain any variables.

Our approach extends the method of *argument-bounded functions* [15,18] that is used, for instance, in the semi-automated verifier ✓`eriFun` [17] for termination analysis and the synthesis of suitable induction axioms. Using this approach, termination of *every* can be easily proved: Selector *tl* is *argument-bounded*, which intuitively means $\#(k) \geq \#(tl(k))$ for all lists $k \neq \emptyset$, where $\#(k)$ counts the occurrences of *list*-constructors $\emptyset$ and :: in k (and thus corresponds to the length of list k plus 1). A system-generated *difference procedure* [15,18] $\Delta_{tl} : list[@A] \to bool$ decides if this inequality is strict for a given list k, which is the case if $k \neq \emptyset$. To prove that the second argument of procedure *every* gets strictly smaller in the recursive call $every(p, tl(k))$, it suffices to show the trivial *termination hypothesis* $\forall k : list[@A].\ k \neq \emptyset \wedge p(hd(k)) \to \Delta_{tl}(k)$.

Proving termination of *groundterm*, however, is challenging and hence is the main problem we tackle in this paper. The key observation is that *every* applies p only to members x of list k. While in Isabelle the user needs to state and prove this knowledge explicitly as a *congruence theorem*, our approach automatically extracts such information from the definition of *every*. More specifically, our approach detects that for any instantiation of type variable $@A$ with a type τ, the number of τ-constructors in each value $x : \tau$ that p is applied to by *every* is bounded by the number of τ-constructors in the elements e of list k: $\sum_{e \in k} \#(e) \geq \#(x)$. We say that *every* is *call-bounded* wrt. p. For the second-order recursion in *groundterm* and $args(t) = t_1 :: \ldots :: t_n :: \emptyset$ this means $\#(t_1) + \ldots + \#(t_n) \geq$

$\#(x)$. Since $t = \mathit{apply}(\mathit{fsym}(t), \mathit{args}(t))$ contains one *term*-constructor more than $\mathit{args}(t)$, we have $\#(t) > \#(t_1) + \ldots + \#(t_n) \geq \#(x)$, so *groundterm* is only called recursively with arguments x that are smaller than t, which ensures termination.

Formally, we parameterize the size measure $\#$ by a type position so that for $\mathit{args}(t) : \mathit{list}[\mathit{term}]$ we can separately count the *list*- and *term*-constructors. This allows us to consider $\mathit{args} : \mathit{term} \to \mathit{list}[\mathit{term}]$ as argument-bounded wrt. type component *term* (i. e., $\mathit{args}(t)$ contains no more *term*-constructors than t).

The contributions of this paper are:

(1) An extended notion of *argument-boundedness* that also considers *components* of types (Sect. 2), along with a corresponding extension of the *estimation calculus* to automate size estimation proofs (Sect. 3). These extensions allow our approach to prove termination of several purely first-order procedures that cannot be handled by the original approach in [15,18].
(2) The novel notion of *call-boundedness* to automatically prove termination of procedures with second-order recursion (Sect. 4). This extension maintains the advantage that "optimized" induction axioms can be synthesized.

We discuss related work and experimental results in Sect. 5. Proofs of the theorems in this paper are given in [1] and [2] along with further details and examples.

2 Size Estimation for Polymorphic Data Types

In this section we define the basic ingredients for size estimation proofs. We begin with a brief account of the programming language $\mathcal{L}$ (which is the input language of ✓`eriFun` [17] and roughly corresponds to the second-order fragment of Haskell with strict evaluation); see [1,2,16] for formal details on $\mathcal{L}$.

2.1 Programming Language

The input language $\mathcal{L}$ of ✓`eriFun` consists of definition principles for freely generated polymorphic data types, for first-order and second-order procedures (based on non-mutual recursion,[2] case analyses via *if*-expressions, and functional composition) that operate on these data types, and for statements about the data types and procedures. Each function symbol can be associated with a so-called *context requirement*, which is stipulated explicitly for procedures (as for *last* in Fig. 1) and implicitly for all selectors. ✓`eriFun` enforces via proof obligations that the context requirement be satisfied for each function call [13]; e. g., *last*, *hd*, and *tl* may only be called on non-empty lists.

A *base type* is a type variable $@A$ or an expression of the form $\mathit{str}[\tau_1, \ldots, \tau_k]$, where $\tau_1, \ldots, \tau_k$ are base types and *str* is a k-ary type constructor ($k \geq 0$). A *type* is a base type or an expression of the form $\tau_1 \times \ldots \times \tau_k \to \tau$ for types $\tau_1, \ldots, \tau_k, \tau$. *Type constructors* are defined by expressions of the following form:

$$\texttt{structure}\ \mathit{str}[@A_1, \ldots, @A_k] <= \ \ldots,\ \mathit{cons}(\mathit{sel}_1 : \tau_1, \ldots, \mathit{sel}_n : \tau_n),\ \ldots \quad (1)$$

[2] Our approach can be extended to handle mutual recursion without much difficulty.

The τ_j are base types, and *str* may only occur as $str[@A_1, \ldots, @A_k]$ in the τ_j. Each *cons* is called a *data constructor* and the sel_j are called *selectors*.

We will address *type symbols* (i. e., type constructors and type variables) in a base type by their *position* $\pi \in \mathbb{N}^*$: $@A|_\epsilon := @A$, $str[\tau_1, \ldots, \tau_k]|_\epsilon := str$, and $str[\tau_1, \ldots, \tau_k]|_{h\pi'} := \tau_h|_{\pi'}$ for $h \in \{1, \ldots, k\}$. $Pos(\tau) \subseteq \mathbb{N}^*$ denotes the set of all valid positions in type τ. For a data constructor $cons(sel_1 : \tau_1, \ldots, sel_n : \tau_n)$ and a type symbol S, the set

$$Pos_S(cons) := \{(j, \pi) \in \{1, \ldots, n\} \times \mathbb{N}^* \mid \pi \in Pos(\tau_j),\ \tau_j|_\pi = S\} \tag{2}$$

contains the positions of all occurrences of S in the selector types of *cons*, given by a selector number j and a position π in τ_j. Data constructor *cons* is called *reflexive* if $Pos_{str}(cons) \neq \emptyset$, and *irreflexive* otherwise.

Subsequently, we let $\Sigma(P)$ denote the signature of all function symbols defined by an $\mathcal{L}$-program P. As usual, $\mathcal{T}(\Sigma(P), \mathcal{V})$ denotes the set of all *terms* over $\Sigma(P)$ and a set $\mathcal{V}$ of variables. We write $\mathcal{T}(\Sigma(P))$ instead of $\mathcal{T}(\Sigma(P), \emptyset)$ for the set of all *ground terms* over $\Sigma(P)$. $\Sigma(P)^c \subset \Sigma(P)$ contains all data constructors of P. $\mathcal{CL}(\Sigma(P), \mathcal{V})$ is the set of *clauses* over $\Sigma(P)$, i. e., sets of literals. A *literal* is an *if*-free Boolean term or the negation $if(b, false, true)$ of such a term.

For a ground type[3] τ, $\mathbb{V}(P)_\tau$ denotes the "values" of type τ: If τ is a ground base type, $\mathbb{V}(P)_\tau := \mathcal{T}(\Sigma(P)^c)_\tau$, and for each ground type $\tau = \tau_1 \times \ldots \times \tau_k \to \tau_{k+1}$, $\mathbb{V}(P)_\tau$ contains all closed (i. e., no free variables) λ-expressions of type τ; e. g., $\lambda t : term.\ groundterm(t) \in \mathbb{V}(P)_{term \to bool}$.

The call-by-value interpreter $eval_P : \mathcal{T}(\Sigma(P)) \mapsto \mathbb{V}(P)$ is a partial function that defines the operational semantics of $\mathcal{L}$ [2]. The evaluation of a ground term t either (i) succeeds and yields a value $eval_P(t) \in \mathbb{V}(P)$ or (ii) diverges, because a procedure called in t does not terminate.

A procedure `procedure` $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau <=$ `assume` c_f; B_f of a program P *terminates* iff the interpreter $eval_P$ returns a value for each procedure call $f(q_1, \ldots, q_n)$. The q_i are either constructor ground terms or λ-expressions that contain only calls of arbitrary, but *terminating* functions. Program P *terminates* iff all procedures f defined in P terminate.

A universally quantified formula of the form $\forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ b$, where $b \in \mathcal{T}(\Sigma(P), \mathcal{V})_{bool}$, is *true* iff P terminates and $eval_{P'}(b[\vec{q}]) = true$ for each terminating program $P' \supseteq P$ and all $q_1, \ldots, q_n \in \mathbb{V}(P')$.[4]

2.2 A Size Measure for Values of Base Types

Our size measure $\#(t, \pi)$ for terms t is parameterized with a type position π so that we can precisely specify which data constructors are to be counted. Figure 2 sketches an implementation of Mergesort: Procedure *split* splits list k into two lists that are recursively sorted and then merged together by some procedure *merge*. To prove termination of *msort*, we need to show that *split* is strictly argument-bounded: $\#(k, \epsilon) > \#(split(k), \mathbf{1})$ and $\#(k, \epsilon) > \#(split(k), \mathbf{2})$. The type position distinguishes between the *list*-constructors of the pair of lists.

[3] A *ground (base) type* is a (base) type without type variables; e. g., $list[\mathbb{N}]$.

[4] Program P' may define additional data types and procedures.

$$\texttt{structure}\ pair[@A, @B] <= mkpair(fst : @A,\ snd : @B)$$

$$\texttt{procedure}\ split(k : list[@A]) : pair[list[@A], list[@A]] <= \ldots$$

$$\begin{array}{l}\texttt{procedure}\ msort(k : list[\mathbb{N}]) : list[\mathbb{N}] <= \\ if\ k = \emptyset\ then\ \emptyset\ else\ if\ tl(k) = \emptyset\ then\ k \\ \quad else\ merge(msort(fst(split(k))), msort(snd(split(k))))\ end\ end\end{array}$$

$$\begin{array}{l}\texttt{procedure}\ filter(k : list[@A],\ p : @A \to bool) : list[@A] <= \\ if\ k = \emptyset\ then\ \emptyset\ else\ if\ p(hd(k))\ then\ hd(k) :: filter(tl(k), p) \\ \qquad\qquad\qquad\qquad\qquad\qquad else\ \ filter(tl(k), p)\ \ end\ end\end{array}$$

$$\begin{array}{l}\texttt{procedure}\ qsort(k : list[\mathbb{N}]) : list[\mathbb{N}] <= \\ if\ k = \emptyset\ then\ \emptyset\ else\ qsort(filter(tl(k),\ \lambda n : \mathbb{N}.\ n \leq hd(k))) <> hd(k) :: \\ \qquad\qquad\qquad\qquad qsort(filter(tl(k),\ \lambda n : \mathbb{N}.\ n > hd(k)))\ \ end\end{array}$$

Fig. 2. Implementation of Mergesort (sketch) and Quicksort

Definition 1. *For each ground base type* $\tau = str[\tau'_1, \ldots, \tau'_k]$ *as in* (1) *the* size measure $\#_\tau : \mathcal{T}(\Sigma(P)^c)_\tau \times Pos(\tau) \to \mathbb{N}$ *is defined by* $\#_\tau(cons(t_1, \ldots, t_n), \pi) :=$

$$\begin{cases} 1 & \text{if } \pi = \epsilon \text{ and } cons \text{ is irreflexive,} \\ 2 + \sum_{(j,\pi') \in Pos_{str}(cons)} \#_{\theta(\tau_j)}(t_j, \pi') & \text{if } \pi = \epsilon \text{ and } cons \text{ is reflexive,} \\ \sum_{(j,\pi') \in Pos_{@A_h}(cons)} \#_{\theta(\tau_j)}(t_j, \pi'\pi'') & \text{if } \pi = h\pi'', \end{cases}$$

where $\theta := \{@A_1/\tau'_1, \ldots, @A_k/\tau'_k\}$ *instantiates the type variables of* str*. If type* τ *is obvious from the context, we will usually omit the type index in* $\#_\tau$*.*

Intuitively, the size $\#(t, \pi)$ of a term $t \in \mathcal{T}(\Sigma(P)^c)_\tau$ is computed as follows: We replicate the type (and data) constructor definitions so that each type constructor occurs at most once in type τ. Then $\#(t, \pi)$ counts the $\tau|_\pi$-constructors in t. For example, $list[list[\mathbb{N}]]$ is transformed into $listA[listB[\mathbb{N}]]$, so $\#(t, \epsilon)$ counts the $listA$-constructors in t and $\#(t, \mathbf{1})$ counts the $listB$-constructors in t.

The formal definition of the size measure above directly uses the type position without needing to replicate any type constructors. Irreflexive data constructors get weight 1. A reflexive data constructor $cons(sel_1 : \tau_1, \ldots, sel_n : \tau_n)$ in a term $cons(t_1, \ldots, t_n)$ is counted with weight[5] 2 and we recurse into those $t_j : \theta(\tau_j)$ that *by definition of cons* may also contain str-constructors ($\tau_j|_{\pi'} = str$). For instance, for $\tau := list[list[\mathbb{N}]]$ and $\pi := \epsilon$ we recursively add the size $\#(t_2, \epsilon)$ of the tl-component of $t_1 :: t_2$, whereas we do *not* recurse into the hd-component t_1. Finally, for $\pi = h\pi''$ we recursively add up the sizes of those t_j that contain $\tau'_h|_{\pi''}$-constructors, so we recurse into the occurrences of the h-th type parameter $@A_h$ in τ_j. For example, for $\tau := list[list[\mathbb{N}]]$, term $t_1 :: t_2$, and $\pi := \mathbf{1}$, $\#_{list[\mathbb{N}]}(t_1, \epsilon) + \#_{list[list[\mathbb{N}]]}(t_2, \mathbf{1})$ counts the $list$-constructors of the *inner* lists.

Example 1. For type $\tau := list[\mathbb{N}]$, $\#_{list[\mathbb{N}]}(t, \epsilon) = 2R + I$ for the numbers R and $I = 1$ of occurrences of $::$ and $\emptyset$ in t, respectively, whereas $\#_{list[\mathbb{N}]}(t, \mathbf{1})$ is the

[5] This simplifies some size estimation proofs; e. g., one can prove that $apply(f, l)$ is greater than $var(v)$ without having to check if the argument list l is non-empty.

sum of the sizes of the elements in list t. Note that $\#_{list[\mathbb{N}]}(\emptyset, \mathbf{1}) = 0$, whereas $\#_{list[\mathbb{N}]}(\mathsf{0} :: \emptyset, \mathbf{1}) = \#_{\mathbb{N}}(\mathsf{0}, \epsilon) = 1 \neq 0$. Thus $\#_{list[\mathbb{N}]}(t, \mathbf{1}) = 0$ iff $t = \emptyset$. This a useful property, cf. the end of Sect. 2.4. ♢

Example 2. For terms $t \in \mathcal{T}(\Sigma(P)^c)_{term}$, $\#_{term}(t, \epsilon)$ counts the occurrences of *term*-constructors *var* (with weight 1) and *apply* (with weight 2) in t. ♢

2.3 Argument-Bounded Functions

A function f is called *argument-bounded* iff the result $f(\ldots, t, \ldots)$ of a function call is bounded by argument t of the call wrt. the size measure (provided that the function may be applied to t); e. g., $\#(tl(k), \epsilon) \leq \#(k, \epsilon)$ for each $k \neq \emptyset$. Such facts are used to show that some parameter x of a procedure p decreases in recursive calls if f is used in the argument of a recursive call; e. g., $p(f(x))$. For the sake of readability we consider only unary functions here, which can be easily generalized to arbitrary arity [1].

Definition 2. *A function $f : \tau \to \tau'$ with context requirement c_f is* (π, ϱ)-argument-bounded *for $\pi \in Pos(\tau)$ and $\varrho \in Pos(\tau')$ iff (i) τ is a base type with $\tau|_\pi = \tau'|_\varrho$ and (ii) $\#(q, \pi) \geq \#(eval_P(f(q)), \varrho)$ for all $q \in \mathbb{V}(P)$ with $eval_P(c_f[q]) = true$.*[6]

Example 3. Procedure *last* (Fig. 1) is $(\mathbf{1}, \epsilon)$-argument-bounded: The size of the last element of list k is bounded by the sum of the sizes of k's elements.

Procedure *filter* (Fig. 2) is (ϵ, ϵ)-argument-bounded wrt. k, because the list of all elements x in k that satisfy $p(x)$ is not longer than k. ♢

Selectors are argument-bounded, as they return a component of their input:

Theorem 1. *Let $sel_j : \tau \to \tau_j$ be a selector as in* (1)*, $\tau = str[@A_1, \ldots, @A_k]$, $\pi \in Pos(\tau)$, and $\varrho \in Pos(\tau_j)$. If $\tau|_\pi = \tau_j|_\varrho$, then sel_j is (π, ϱ)-argument-bounded.*

Example 4. $pred(\ldots) : \mathbb{N} \to \mathbb{N}$ is (ϵ, ϵ)-argument-bounded. $hd : list[@A] \to @A$ is $(\mathbf{1}, \epsilon)$-argument-bounded: The size of the first element of a non-empty list k is bounded by the sum of the sizes of all elements in k. $tl : list[@A] \to list[@A]$ is (ϵ, ϵ)-argument-bounded, as $tl(k)$ contains fewer *list*-constructors "::" than k. *tl* is also $(\mathbf{1}, \mathbf{1})$-argument-bounded, because $tl(k)$ contains a subset of the elements in k. Finally, selector $args : term \to list[term]$ is $(\epsilon, \mathbf{1})$-argument-bounded. ♢

2.4 Difference Procedures

Using argument-bounded functions, we can establish inequalities like $\#(k, \epsilon) \geq \#(tl(k), \epsilon)$ to ensure that the second argument of procedure *every* does not increase in the recursive call (cf. Fig. 1). However, this inequality needs to be strict to guarantee termination of *every*. Strictness of such inequalities is expressed by so-called *difference procedures*; e. g., $\Delta_{tl}^{\epsilon,\epsilon} : list[@A] \to bool$ returns *true* iff $\#(k, \epsilon) > \#(tl(k), \epsilon)$.

[6] "$q \in \mathbb{V}(P)$" implicitly means that τ is instantiated to a ground base type.

(a) $\texttt{procedure}\ \Delta^{\epsilon,\epsilon}_{pred(\ldots)}(x:\mathbb{N}) : bool <= \texttt{assume}\ ?succ(x);\ \ true$
$\texttt{procedure}\ \Delta^{\epsilon,\epsilon}_{tl}(k:list[@A]) : bool <= \texttt{assume}\ ?::(k);\ \ true$
$\texttt{procedure}\ \Delta^{\epsilon,\mathbf{1}}_{args}(t:term) : bool <= \texttt{assume}\ ?apply(t);\ \ true$

(b) $\texttt{procedure}\ \Delta^{\mathbf{1},\epsilon}_{hd}(k:list[@A]) : bool <= \texttt{assume}\ ?::(k);\ \ ?::(tl(k))$
$\texttt{procedure}\ \Delta^{\mathbf{1},\mathbf{1}}_{tl}(k:list[@A]) : bool <= \texttt{assume}\ ?::(k);\ \ true$

Fig. 3. Some automatically synthesized difference procedures for selectors

Definition 3. *For a (π,ϱ)-argument-bounded function $f:\tau\to\tau'$ with context requirement c_f, $\Delta^{\pi,\varrho}_f:\tau\to bool$ is a* (π,ϱ)-difference function for *f iff (i) $\Delta^{\pi,\varrho}_f$ also has context requirement c_f and (ii) for all $q\in\mathbb{V}(P)$ with $eval_P(c_f[q]) = true$*

$$eval_P\big(\Delta^{\pi,\varrho}_f(q)\big) = true \iff \#(q,\pi) > \#(eval_P(f(q)),\varrho)\ .$$

(ϵ,ϱ)-argument-bounded selectors have quite simple difference procedures, because the selector cancels the leading data constructor, cf. Fig. 3(a):

Theorem 2. *Let $sel_j:\tau\to\tau_j$ be an (ϵ,ϱ)-argument-bounded selector for some ϱ. Then a (ϵ,ϱ)-difference procedure for sel_j is given by*

$$\texttt{procedure}\ \Delta^{\epsilon,\varrho}_{sel_j}(x:\tau) : bool <= \texttt{assume}\ ?cons(x);\quad true\ .$$

The synthesis of (π,ϱ)-difference procedures for selectors with $\pi\neq\epsilon$ is a bit more involved and described in [1,2]. Figure 3(b) illustrates the idea by examples. $\Delta^{\mathbf{1},\epsilon}_{hd}$ returns *true* iff list k contains at least two elements: Since the size of each element in k is ≥ 1, the size of the first element $hd(k)$ is smaller than the sum of the sizes of all elements in k. The uniform synthesis of such procedures uses the fact that $\#(hd(k)::tl(k),\mathbf{1}) > \#(hd(k),\epsilon)$ iff $\#(tl(k),\mathbf{1}) > 0$, i. e., iff $?::(tl(k))$.

3 Estimation Proofs

So-called estimation proofs can be used to verify that a procedure computes an argument-bounded function. We obtain estimation proofs from the estimation calculus, which is also used to synthesize difference procedures for argument-bounded procedures and to generate termination hypotheses for recursively defined procedures.

3.1 The Estimation Calculus

The estimation calculus is used to prove inequalities $\#(t_1,\pi_1)\geq\#(t_2,\pi_2)$. The inequalities to be shown are given by some set E. When proving an inequality, a clause Δ (called a *difference equivalent of* E) is synthesized such that the proved inequality is strict iff one of the literals in Δ holds.

Definition 4. *For a terminating program P, let $\Gamma_{\pi,\varrho}$ be a family of (π,ϱ)-argument-bounded function symbols in P. Given a* call context *$C\in\mathcal{CL}(\Sigma(P),\mathcal{V})$, the* estimation calculus *is defined by:*

Language: *Estimation tuples of the form* $\langle \Delta, E \rangle$*, where* $\Delta \in \mathcal{CL}(\Sigma(P), \mathcal{V})$ *and* $E \subset_{fin} \mathcal{E}(\Sigma(P), \mathcal{V}) := \{(t_1, \pi_1) \succcurlyeq (t_2, \pi_2) \mid t_i \in \mathcal{T}(\Sigma(P), \mathcal{V})_{\tau_i}$ *for some base types* τ_1, τ_2*,* $\pi_i \in Pos(\tau_i)$ *for* $i = 1, 2$ *and* $\tau_1|_{\pi_1} = \tau_2|_{\pi_2}\}$*.*

Inference Rules: *The following estimation rules are given for each type constructor str and data constructors cons, rcons, ircons,* $ircons_1$*, and* $ircons_2$ *of str, where rcons is* reflexive *and all* $ircons_i$ *are* irreflexive*:*[7]

Identity

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

Equivalence

(2) $$\frac{\langle \Delta, E \uplus \{(t_1, \epsilon) \succcurlyeq (t_2, \epsilon)\} \rangle}{\langle \Delta, E \rangle} \quad \text{if } C \vdash ?ircons_1(t_1) \text{ and } C \vdash ?ircons_2(t_2)$$

Strong Estimation

(3) $$\frac{\langle \Delta, E \uplus \{(t_1, \epsilon) \succcurlyeq (t_2, \epsilon)\} \rangle}{\langle \Delta \cup \{true\}, E \rangle} \quad \text{if } C \vdash ?rcons(t_1) \text{ and } C \vdash ?ircons(t_2)$$

Strong Embedding

(4) $$\frac{\langle \Delta, E \uplus \{(t_1, \epsilon) \succcurlyeq (t_2, \pi_2)\} \rangle}{\langle \Delta \cup \{true\}, E \cup \{(SEL_j(t_1), \pi_1) \succcurlyeq (t_2, \pi_2)\} \rangle} \quad \begin{array}{l} \text{if } C \vdash ?rcons(t_1) \text{ and} \\ (j, \pi_1) \in Pos_{str}(rcons) \end{array}$$

Argument Estimation

(5) $$\frac{\langle \Delta, E \uplus \{(t', \pi') \succcurlyeq (f(t, t_1, \ldots, t_n), \varrho)\} \rangle}{\langle \Delta \cup \{\Delta_f^{\pi, \varrho}(t, t_1, \ldots, t_n)\}, E \cup \{(t', \pi') \succcurlyeq (t, \pi)\} \rangle} \quad \text{if } f \in \Gamma_{\pi, \varrho}$$

Weak Embedding

(6) $$\frac{\langle \Delta, E \uplus \{(t_1, \epsilon) \succcurlyeq (t_2, \epsilon)\} \rangle}{\langle \Delta, E \cup \{ (SEL_j(t_1), \pi) \succcurlyeq (SEL_j(t_2), \pi) \mid (j, \pi) \in Pos_{str}(rcons)\} \rangle}$$
if $C \vdash ?rcons(t_1)$ *and* $C \vdash ?rcons(t_2)$

Constructor Wrapping

(7) $$\frac{\langle \Delta, E \uplus \{(t, \varrho) \succcurlyeq (cons(t_1, \ldots, t_n), h\pi')\} \rangle}{\langle \Delta, E \cup \{(t, \varrho) \succcurlyeq (t_j, \pi\pi')\} \rangle} \quad \text{if } Pos_{@A_h}(cons) = \{(j, \pi)\}$$

Minimum

(8) $$\frac{\langle \Delta, E \uplus \{(t_1, \epsilon) \succcurlyeq (t_2, \epsilon)\} \rangle}{\langle \Delta \cup \{ ?rcons(t_1) \mid rcons \in R\}, E \rangle}$$
if $C \vdash ?ircons(t_2)$ *and* R *is the set of all reflexive constructors of str*

Deduction: *We write* $\langle \Delta_0, E_0 \rangle \Rrightarrow_{\Gamma, C} \langle \Delta_1, E_1 \rangle$ *iff* $\langle \Delta_1, E_1 \rangle$ *results from* $\langle \Delta_0, E_0 \rangle$ *by applying some estimation rule.* $\Rrightarrow^*_{\Gamma, C}$ *denotes the reflexive and transitive closure of* $\Rrightarrow_{\Gamma, C}$*.* $\langle \Delta_0, E_0 \rangle \Rrightarrow^*_{\Gamma, C} \langle \Delta_n, E_n \rangle$ *is called a* deduction *of* $\langle \Delta_n, E_n \rangle$ *from* $\langle \Delta_0, E_0 \rangle$*. We use the notation* $\vdash_{\Gamma, C} \langle \Delta, (t_1, \pi_1) \succcurlyeq (t_2, \pi_2) \rangle$ *iff* $\langle \emptyset, \{(t_1, \pi_1) \succcurlyeq (t_2, \pi_2)\} \rangle \Rrightarrow^*_{\Gamma, C} \langle \Delta, \emptyset \rangle$*.* $(t_1, \pi_1) \succcurlyeq_{\Gamma, C} (t_2, \pi_2)$ *denotes the existence of a* difference equivalent Δ *with* $\vdash_{\Gamma, C} \langle \Delta, (t_1, \pi_1) \succcurlyeq (t_2, \pi_2) \rangle$*.*

[7] The rules are applied from top to bottom. We write $C \vdash ?cons(t)$ iff (i) $t = cons(\ldots)$ or (ii) $?cons(t) \in C$ or (iii) $\neg ?cons'(t) \in C$ for all *str*-constructors $cons' \neq cons$. $SEL_j(t)$ stands for t_j if $t = cons(\ldots, t_j, \ldots)$, and abbreviates $sel_j(t)$ otherwise.

Definition 4 extends the calculus from [15,18] by type positions π and rule (7) for data constructors such as *mkpair* (Fig. 2) that just wrap the item of interest; e. g., to show $\#(t, \varrho) \geq \#(mkpair(t_1, t_2), \mathbf{1})$ it suffices to show $\#(t, \varrho) \geq \#(t_1, \epsilon)$.

Example 5. We get the following estimation proof for call context $C := \{k \neq \emptyset\}$:

$$\begin{aligned}
&\langle \emptyset,\ \{(k,\epsilon) \succcurlyeq (\mathit{filter}(tl(k), g), \epsilon)\}\rangle \\
\Rrightarrow_{\Gamma,C}\ &\langle \{\Delta^{\epsilon,\epsilon}_{\mathit{filter}}(tl(k), g)\},\ \{(k,\epsilon) \succcurlyeq (tl(k), \epsilon)\}\rangle && \text{by (5)} \\
\Rrightarrow_{\Gamma,C}\ &\langle \{\mathit{true},\ \Delta^{\epsilon,\epsilon}_{\mathit{filter}}(tl(k), g)\},\ \{(tl(k),\epsilon) \succcurlyeq (tl(k), \epsilon)\}\rangle && \text{by (4)} \\
\Rrightarrow_{\Gamma,C}\ &\langle \{\mathit{true},\ \Delta^{\epsilon,\epsilon}_{\mathit{filter}}(tl(k), g)\},\ \emptyset\rangle && \text{by (1)}\ \diamondsuit
\end{aligned}$$

In the following, we use expressions of the form (i) $(t_1, \pi_1) \geqslant_{\#} (t_2, \pi_2)$ and (ii) $(t_1, \pi_1) >_{\#} (t_2, \pi_2)$ for terms $t_i \in \mathcal{T}(\Sigma(P), \mathcal{V})_{\tau_i}$ and positions $\pi_i \in Pos(\tau_i)$, $i = 1, 2$, with $\tau_1|_{\pi_1} = \tau_2|_{\pi_2}$. Such expressions are *true* iff (i) $\#(eval_P(t_1), \pi_1) \geq \#(eval_P(t_2), \pi_2)$ or (ii) $\#(eval_P(t_1), \pi_1) > \#(eval_P(t_2), \pi_2)$, respectively.

Theorem 3. *The estimation calculus is* sound*: If* $\vdash_{\Gamma,C} \langle \Delta, (t_1, \pi_1) \succcurlyeq (t_2, \pi_2)\rangle$*, then the following formulas are true (where* $x_1, \ldots, x_n$ *are all variables in* C*,* t_1*, and* t_2 *such that* $x_i \in \mathcal{V}_{\tau_i}$ *for all* $i \in \{1, \ldots, n\}$*):*

(1) $\forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ \bigwedge C \to (t_1, \pi_1) \geqslant_{\#} (t_2, \pi_2)$
(2) $\forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ \bigwedge C \to \left[\bigvee \Delta \leftrightarrow (t_1, \pi_1) >_{\#} (t_2, \pi_2)\right]$

Theorem 4. *The set* $\{(t_1, \pi_1) \succcurlyeq (t_2, \pi_2) \in \mathcal{E}(\Sigma(P), \mathcal{V}) \mid (t_1, \pi_1) \geqslant_{\Gamma,C} (t_2, \pi_2)\}$ *of provable size estimation problems is decidable.*

Thus whenever a proof procedure for the estimation calculus finds a proof of $(t_1, \pi_1) \geqslant_{\Gamma,C} (t_2, \pi_2)$, we know that t_1 is at least as big as t_2 by Theorem 3. If no estimation proof exists, the inequality might still hold, because the estimation calculus is incomplete. However, it is powerful enough to solve termination problems that are relevant in practice, see Sect. 5.

3.2 Proving Argument-Boundedness of Procedures

Using the estimation calculus, we can prove argument-boundedness of a procedure f by analyzing the *result terms* $t_1, \ldots, t_n$ of f (these are maximal *if*-free terms outside an *if*-condition in the body B_f). The *call context* $C_i \in \mathcal{CL}(\Sigma(P), \mathcal{V})$ of a result term t_i consists of the conditions in B_f that lead to t_i.

Theorem 5. *Let* `procedure` $f(x : \tau) : \tau'$ `<= assume` c_f*;* B_f *be a terminating procedure,* $\pi \in Pos(\tau)$*, and* $\varrho \in Pos(\tau')$ *such that (i)* τ *is a base type with* $\tau|_\pi = \tau'|_\varrho$ *and (ii)* $\vdash^f_{\Gamma,C_i} \langle \Delta_i, (x, \pi) \succcurlyeq (t_i, \varrho)\rangle$ *for each result term* t_i *of* f *under call context* C_i *and some* Δ_i*, where* $\vdash^f_{\Gamma,C_i}$ *differs from* $\vdash_{\Gamma,C_i}$ *in that the* Argument Estimation *rule (5) may also be used for each recursive call* $f(t')$ *in* t_i*.*

Then f *is* (π, ϱ)*-argument-bounded and* `procedure` $\Delta^{\pi,\varrho}_f(x : \tau) : bool$ `<=` B_{Δ_f} *is a* (π, ϱ)*-difference procedure for* f*, where* B_{Δ_f} *is derived from* B_f *by replacing each result term* t_i *with the disjunction* $\bigvee \Delta_i$ *(represented by if-conditionals).*

$\texttt{procedure}\ \Delta^{\mathbf{1},\epsilon}_{last}(k : list[@A]) : bool <=$
$\texttt{assume}\ k \neq \emptyset;\ \ if\ tl(k) = \emptyset\ then\ false\ else\ true\ end$

$\texttt{procedure}\ \Delta^{\epsilon,\epsilon}_{filter}(k : list[@A],\ p : @A \to bool) : bool <=$
$if\ k = \emptyset\ then\ false\ else\ if\ p(hd(k))\ then\ \Delta^{\epsilon,\epsilon}_{filter}(tl(k), p)\ else\ true\ end\ end$

Fig. 4. Difference procedures for argument-bounded procedures

Example 6. Procedure *last* shown in Fig. 1(a) is $(\mathbf{1}, \epsilon)$-argument-bounded, because $\vdash^{last}_{\Gamma,C_1} \langle \Delta_1, (k, \mathbf{1}) \succcurlyeq (hd(k), \epsilon)\rangle$ for $C_1 := \{k \neq \emptyset, tl(k) = \emptyset\}$, $\Delta_1 := \{\Delta^{\mathbf{1},\epsilon}_{hd}(k)\}$, and $\vdash^{last}_{\Gamma,C_2} \langle \Delta_2, (k, \mathbf{1}) \succcurlyeq (last(tl(k)), \epsilon)\rangle$ for $C_2 := \{k \neq \emptyset,\ tl(k) \neq \emptyset\}$, $\Delta_2 := \{\Delta^{\mathbf{1},\mathbf{1}}_{tl}(k),\ \Delta^{\mathbf{1},\epsilon}_{last}(tl(k))\}$. $\bigvee \Delta_1$ simplifies to *false* and $\bigvee \Delta_2$ simplifies to *true* using the definition of the difference procedures (Fig. 3) and call contexts C_1 and C_2. Difference procedure $\Delta^{\mathbf{1},\epsilon}_{last}$ is shown in Fig. 4: The last element of list k is smaller than the sum of the sizes of all list elements if the length of k is ≥ 2.$\diamondsuit$

Example 7. For procedure *filter* (Fig. 2), the difference procedure $\Delta^{\epsilon,\epsilon}_{filter}$ wrt. parameter k (Fig. 4) reflects the intuition that the returned sublist of k is shorter than k iff at least one element x of k does *not* satisfy $p(x)$. $\diamondsuit$

4 Automated Termination Proofs

We implicitly assume procedure bodies to be in η-long form to clearly exhibit *indirect* function calls; e. g., $every(p, tl(k))$ abbreviates $every(\lambda x : @A.\ p(x), tl(k))$ in Fig. 1, because $p =_\eta \lambda x : @A.\ p(x)$. Subterm $p(x)$ is an *indirect* function call, whereas $p(hd(k))$ and $every(\lambda x : @A.\ p(x), tl(k))$ are *direct* function calls:

Definition 5. *A* direct call *of a function f in a term t is a subterm $f(t_1, \ldots, t_n)$ of t that occurs outside a λ-expression. A subterm $f(t_1, \ldots, t_n)$ of t inside a λ-expression is an* indirect call *of f.*

Definition 6. *For a procedure or λ-expression f with body B_f and parameters $x_1, \ldots, x_n$, a procedure or λ-expression g, and $q_1, \ldots, q_n$, $q'_1, \ldots, q'_m \in \mathbb{V}(P)$, we write $f(q_1, \ldots, q_n) \rhd g(q'_1, \ldots, q'_m)$ iff B_f contains a subterm $h(t'_1, \ldots, t'_m)$ under some call context C such that for $\sigma := \{x_1/q_1, \ldots, x_n/q_n\}$, $\sigma(h) =_\eta g$, $eval_P(\sigma(c)) = true$ for all $c \in C$, and $q'_i = eval_P(\sigma(t'_i))$ for all $i = 1, \ldots, m$.*

Intuitively, relation $\rhd$ means "requires evaluation of". For instance, we have $every(groundterm, var(q) :: \emptyset) \rhd groundterm(var(q))$.

Now we are ready to state a termination criterion for procedures *without* second-order recursion. The formulas *(ii)* of Theorem 6 are so-called *termination hypotheses*; if these formulas are true, the procedure terminates.

Theorem 6. *A procedure* $\texttt{procedure}\ f(x : \tau) : \tau' <= \texttt{assume}\ c_f;$ *B_f terminates if all procedures $g \neq f$ occurring in B_f and c_f terminate and if there is some $\pi \in Pos(\tau)$ such that each recursive call $f(t)$ in B_f under some call context $C \in \mathcal{CL}(\Sigma(P), \mathcal{V})$ is a* direct *procedure call such that (i) $\vdash_{\Gamma,C} \langle \Delta, (x, \pi) \succcurlyeq (t, \pi)\rangle$ for some Δ, and (ii) $\forall x : \tau.\ \bigwedge C \to \bigvee \Delta$ is true.*

Example 8. Procedure *qsort* (Fig. 2) terminates. For $\pi := \epsilon$, $C := \{k \neq \emptyset\}$, and any g, $(k, \epsilon) \geqslant_{\Gamma,C} (\mathit{filter}(\mathit{tl}(k), g), \epsilon)$ with $\bigvee \Delta = \mathit{true}$, cf. Example 5. ♢

Example 9. Procedure *termlist.size* terminates according to Theorem 6:

```
procedure termlist.size(k : list[term]) : ℕ <=
if k = ∅ then 0
  else if ?var(hd(k))
         then 1 + termlist.size(tl(k))
         else 1 + termlist.size(tl(k)) + termlist.size(args(hd(k)))
end   end
```

For position $\pi := \mathbf{1}$ it is easy to show $(k, \mathbf{1}) \geqslant_{\Gamma,C_1} (\mathit{tl}(k), \mathbf{1})$ and $(k, \mathbf{1}) \geqslant_{\Gamma,C_2} (\mathit{args}(\mathit{hd}(k)), \mathbf{1})$ for the respective call contexts C_1 and C_2. The resulting termination hypotheses $\forall k : \mathit{list}[\mathit{term}].\ k \neq \emptyset \wedge \ldots \rightarrow \Delta^{\mathbf{1},\mathbf{1}}_{\mathit{tl}}(k)$ and $\forall k : \mathit{list}[\mathit{term}].\ k \neq \emptyset \wedge \neg\, ?\mathit{var}(\mathit{hd}(k)) \rightarrow \big(\Delta^{\mathbf{1},\epsilon}_{\mathit{hd}}(k) \vee \Delta^{\epsilon,\mathbf{1}}_{\mathit{args}}(\mathit{hd}(k))\big)$ are obviously true, cf. Fig. 3. ♢

Example 9 cannot be solved by the original method in [15,18], because there a list is always measured by its *length* (the special case $\pi = \epsilon$ of our theorem).

4.1 Call-Bounded Procedures

Call-bounded procedures f are well-behaved in the sense that they call their functional parameter only with arguments of a bounded size: For each sequence $f(g, q) \rhd^* g(q')$ of procedure calls, the size of q is a bound of the size of q'. We consider only procedures with two parameters in the following definition for readability reasons; the straightforward generalization is given in [1].

Definition 7. *A procedure* `procedure` $f(F : \tau' \rightarrow \tau'',\ x : \tau) : \tau'''$ `<= assume` c_f; B_f *is* (π, ϱ)-call-bounded *for* $\pi \in \mathit{Pos}(\tau)$ *and* $\varrho \in \mathit{Pos}(\tau')$ *iff* τ *is a base type with* $\tau|_\pi = \tau'|_\varrho$ *such that* $\#(q, \pi) \geq \#(q', \varrho)$ *for all* $g \in \mathbb{V}(P)_{\tau' \rightarrow \tau''}$ *and* $q \in \mathbb{V}(P)_\tau$ *with* $f(g, q) \rhd h_1(\ldots) \rhd \ldots \rhd h_n(\ldots) \rhd g(q')$, *where* $h_i \neq g$ *for all* $i = 1, \ldots, n$.

Example 10. every is $(\mathbf{1}, \epsilon)$-call-bounded, because parameter p will only be called with an argument x with $\#(k, \mathbf{1}) \geq \#(x, \epsilon)$. More formally, $\#(q, \mathbf{1}) \geq \#(q', \epsilon)$ whenever $\mathit{every}(g, q) \rhd \mathit{every}(g, q_1) \rhd \ldots \rhd \mathit{every}(g, q_n) \rhd g(q')$ for some $n \geq 0$. For the same reason, *filter* is also $(\mathbf{1}, \epsilon)$-call-bounded. ♢

The next theorem allows us to easily identify many call-bounded procedures.

Theorem 7. *A procedure* `procedure` $f(F : \tau' \rightarrow \tau'',\ x : \tau) : \tau'''$ `<= assume` c_f; B_f *is* (π, ϱ)*-call-bounded for* $\pi \in \mathit{Pos}(\tau)$ *and* $\varrho \in \mathit{Pos}(\tau')$ *if* τ *is a base type with* $\tau|_\pi = \tau'|_\varrho$ *and* F *occurs in* B_f *only*

(1) in direct function calls $F(t)$ *under some call context* C *such that* $\vdash_{\Gamma,C} \langle \Delta, (x, \pi) \succcurlyeq (t, \varrho) \rangle$ *for some* Δ, *or*
(2) in direct recursive calls $f(F, t')$ *under some call context* C' *such that* $\vdash_{\Gamma,C'} \langle \Delta', (x, \pi) \succcurlyeq (t', \pi) \rangle$ *for some* Δ'.

Example 11. Procedure *every* (Fig. 1) is easily proved $(\mathbf{1}, \epsilon)$-call-bounded:

(1) $\vdash_{\Gamma,C} \langle \{\Delta^{\mathbf{1},\epsilon}_{\mathit{hd}}(k)\}, (k, \mathbf{1}) \succcurlyeq (\mathit{hd}(k), \epsilon) \rangle$, where $C = \{k \neq \emptyset\}$
(2) $\vdash_{\Gamma,C'} \langle \{\Delta^{\mathbf{1},\mathbf{1}}_{\mathit{tl}}(k)\}, (k, \mathbf{1}) \succcurlyeq (\mathit{tl}(k), \mathbf{1}) \rangle$, where $C' = \{k \neq \emptyset,\ p(\mathit{hd}(k))\}$ ♢

Generalized Detection of Call-Bounded Procedures. Theorem 7 handles the frequently occurring special case of Definition 7 where $h_1 = \ldots = h_n = f$, i. e., the functional parameter F is either called directly or passed to the recursive call $f(F, t')$ without modification. The theorem can be generalized (i) to allow f to pass F to another call-bounded procedure $h_i \neq f$, (ii) to allow modification of F in recursive calls by encapsulating it in a λ-expression $\lambda x'.\ \ldots F(t'')\ldots$, and (iii) to allow F to occur in *indirect* recursive calls as well, see [1].

4.2 Proving Termination of Procedures

The concept of call-bounded procedures allows us to prove termination of procedures that pass themselves to a call-bounded second-order procedure: In the following theorem, the arguments t of *direct* recursive calls need to decrease, cf. requirements *(1)* and *(2)*. *Indirect* recursive calls need to occur via a call-bounded procedure g, cf. *(3)*. This procedure g must be called with a bounding argument t' that is strictly smaller than the argument x of f, cf. *(4)* and *(5)*.

Theorem 8. *A procedure* $f(x:\tau):\tau' <=$ `assume` c_f; B_f *terminates if all procedures* $g \neq f$ *occurring in* B_f *and* c_f *terminate and if there is some* $\pi \in Pos(\tau)$ *such that for each* direct *recursive call* $f(t)$ *in* B_f *under some call context* C

(1) $\vdash_{\Gamma,C} \langle \Delta, (x,\pi) \succcurlyeq (t,\pi)\rangle$ *for some* Δ *and*
(2) $\forall x:\tau.\ \bigwedge C \rightarrow \bigvee \Delta$ *is true*

and for each indirect *recursive call* $g(f,t')$ *in* B_f *under some call context* C'

(3) procedure g *is* (π',π)*-call-bounded for some* π'*,*
(4) $\vdash_{\Gamma,C'} \langle \Delta', (x,\pi) \succcurlyeq (t',\pi')\rangle$ *for some* Δ'*, and*
(5) $\forall x:\tau.\ \bigwedge C' \rightarrow \bigvee \Delta'$ *is true.*

Example 12. Procedure *groundterm* of Fig. 1 terminates by Theorem 8:

(3) *every* is $(\mathbf{1},\epsilon)$-call-bounded, see Example 11 (i. e., $\pi := \epsilon$)
(4) $\vdash_{\Gamma,C'} \langle \{\Delta^{\epsilon,\mathbf{1}}_{args}(t)\}, (t,\epsilon) \succcurlyeq (args(t),\mathbf{1})\rangle$, where $C' := \{\neg?var(t)\}$
(5) $\forall t:term.\ \neg?var(t) \rightarrow \Delta^{\epsilon,\mathbf{1}}_{args}(t)$ is trivially true, see Fig. 3 $\diamondsuit$

Similarly to [15,18], Theorem 8 can be generalized in a straightforward way from a single parameter and type position to a set of parameter indices and positions (e. g., for a lexicographic combination of size orders to prove termination of procedures like the Ackermann function). Furthermore, indirect recursive calls $f(t'')$ may be (deeply) nested within λ-expressions; e. g., in $g(\lambda y.\ldots f(t'')\ldots, t')$, see [1] for details and examples.

Induction Axioms. From a terminating procedure one can uniformly synthesize a sound induction axiom. Our method maintains the advantage of the original approach [15,18] that the induction axiom(s) can be optimized by analyzing the termination proof: Some variables can be universally quantified in the induction hypotheses (as in [9]) and irrelevant premises are removed [2].

5 Related Work and Experimental Results

Our method is intended to be used in inductive theorem provers: In this setting it is important that many procedures can be proved terminating without user interaction so that the user can quickly move on to the actual verification task.

In Isabelle 2009 [8,10,14] a termination proof of a procedure with second-order recursion requires the user to state and prove a *congruence theorem* about the second-order procedure involved. For instance, Isabelle can only prove termination of *groundterm* when the user has proved and explicitly tagged $k = k' \wedge \big(\forall x : @A.\ x \in k \to p(x) = p'(x)\big) \to \mathit{every}(p, k) = \mathit{every}(p', k')$ as a *congruence rule* about procedure *every*. Our approach with call-bounded procedures can be considered as *automatically* discovering and proving congruence theorems such as $k = k' \wedge \big(\forall x : @A.\ \#(x, \epsilon) < \#(k, \mathbf{1}) \to p(x) = p'(x)\big) \to \mathit{every}(p, k) = \mathit{every}(p', k')$.

In PVS [11] the user needs to supply a measure function that computes the size of a data object, so there is no automation as in our approach.

Since Coq does not offer automated termination analysis either, Barthe et al. [4] suggest an approach that ensures termination by typing. Their system uses *sized types*, i. e., types that contain information about the size of values. For instance, argument-boundedness of procedure *split* (cf. Fig. 2) is expressed by assigning type $\mathit{list}^{\ell}[@A] \to \mathit{pair}[\mathit{list}^{\ell}[@A], \mathit{list}^{\ell}[@A]]$ to *split*, where $\mathit{list}^{\ell}[@A]$ represents lists of length $\leq \ell$. This analysis is less detailed than ours, because it does not detect the cases when the resulting lists are strictly smaller than the input list. Thus the termination proof of Mergesort fails in this approach, whereas our method succeeds using difference procedures that identify these cases [1]. In the terminology of sized types, call-boundedness of *every* could be expressed by assigning type $(@A^{\ell} \to \mathit{bool}) \times \mathit{list}[@A^{\ell}] \to \mathit{bool}$ to *every*, thus constraining the *items* of the list to be of size $\leq \ell$. The approach by Barthe et al. has *not* been integrated into Coq.

ACL2 [5,9] offers heuristics for automated termination proofs, but procedures cannot be defined by second-order recursion.

There are also several stand-alone approaches for termination analysis, which are useful if only termination of a procedure is to be proved (i. e., if there is no need to synthesize induction axioms for subsequent proofs about the procedure). As an example, we mention the Haskell termination analyzer by Giesl et al. [6].

Experimental Results. Our approach has been integrated into the verifier ✓`eriFun`, which allows us to compare its performance with Isabelle.[8] Table 1 shows the results of our experiments: We evaluated our approach on 16 representative procedures with second-order recursion, including all examples from [8,10,11,14]. These procedures are based on 8 common second-order procedures without second-order recursion (e. g., *map*, *foldl*, and *every*). The set of first-order examples comprises auxiliary procedures for the examples of second-order recursion as well as the examples from [15,18], which we included to make sure that our

[8] See `http://www.informatik.tu-darmstadt.de/pm/~aderhold` for an experimental version of ✓`eriFun` and a list of the example procedures.

Table 1. Termination proving capabilities of inductive provers

number and category of examples	Isabelle	✓eriFun
16 procedures with second-order recursion	0	15
8 second-order procedures without second-order recursion	8	8
40 first-order procedures without second-order recursion	24	38
64 procedures in total	32	61

approach properly subsumes the original approach. Indeed, our approach only fails on procedures that the original approach already fails on (e. g., because a parameter is *increased* in recursive calls).

Isabelle fails on the examples of second-order recursion, because we only supplied the raw definition of the procedures. When the user states and proves the required congruence theorems, 15 procedures can be shown terminating as well. The remaining procedure is an artificial example (computing the constant zero function) by Krauss [8] that our approach also fails on, because we would need information about the procedure's semantics before proving its termination.

We did not base our method on the *sized types* approach [4] (although this would be feasible in principle), because the latter cannot solve many naturally occurring examples; e. g., it succeeds on only 21 of 40 first-order procedures in the example set and fails on several common sorting algorithms from [15].

In summary, the experimental results show that the extended estimation calculus (though incomplete) is powerful enough to prove termination of the everyday examples of second-order recursion that frequently occur in practice.

6 Conclusion

We extended the concept of argument-bounded functions [15,18] in two respects: Firstly, we parameterized the size measure to also consider *components* of types. This facilitates automated termination proofs (e. g., for the Mergesort implementation in Fig. 2) that were impossible with the original method. Secondly, we identified the new notion of *call-boundedness* to automate termination proofs for procedures with second-order recursion, which the original method could not cope with at all.

Our method has been integrated into ✓eriFun [17]. It *automatically* solves the typical examples of second-order recursion considered in the literature [8,10,11,14] and in this paper within few seconds, whereas other state-of-the-art theorem provers require guidance by the user.

Information gathered from termination analysis is among the most important keys for guiding highly automated verifiers such as ACL2 and ✓eriFun when selecting useful induction axioms. In all examples of second-order recursion, ✓eriFun synthesizes optimal induction axioms using the results of our method for termination analysis. Although the examples are not overly difficult, such procedures using second-order recursion via *map*, *every*, *filter*, *foldl*, *foldr*, and similar procedures on other data types (e. g., trees) are widely used in func-

tional programming. Hence our method significantly improves the state of the art in automated theorem proving by reducing the need for user interaction.

Acknowledgment. I am grateful to Jürgen Giesl, Alexander Krauss, Simon Siegler, and Christoph Walther for helpful discussions, to Nathan Wasser for implementing the approach, and to the anonymous referees for valuable feedback.

References

1. Markus Aderhold. Automated termination analysis for programs with second-order recursion. Technical report, TU Darmstadt, 2009.
2. Markus Aderhold. *Verification of Second-Order Functional Programs*. Doctoral dissertation, TU Darmstadt, 2009.
3. Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Kluwer Academic Publishers, 2002.
4. Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. CIC$\widehat{\ }$: Type-based termination of recursive definitions in the calculus of inductive constructions. In *Proc. of LPAR-13*, volume 4246 of *LNCS*, pages 257–271. Springer, 2006.
5. Robert S. Boyer, David M. Goldschlag, Matt Kaufmann, and J Strother Moore. Functional instantiation in first-order logic. In Vladimir Lifschitz, editor, *Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.
6. J. Giesl, S. Swiderski, P. Schneider-Kamp, and R. Thiemann. Automated termination analysis for Haskell: From term rewriting to programming languages. In *Proc. of RTA-17, Seattle, USA*, volume 4098 of *LNCS*, pages 297–312. Springer, 2006.
7. Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
8. Alexander Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*. Doctoral dissertation, TU München, Germany, 2009.
9. Panagiotis Manolios and Aaron Turon. All-termination(T). In *Proc. of TACAS-2009*, volume 5505 of *LNCS*, pages 398–412. Springer, 2009.
10. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
11. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, November 2001.
12. Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.
13. A. Schlosser, C. Walther, M. Gonder, and M. Aderhold. Context dependent procedures and computed types in √`veriFun`. In *Proc. of 1st Workshop Programming Languages meet Program Verification*, volume 174 of *ENTCS*, pages 61–78, 2007.
14. Konrad Slind. *Reasoning about Terminating Functional Programs*. PhD thesis, TU München, Germany, 1999.
15. Christoph Walther. On proving the termination of algorithms by machine. *Artificial Intelligence*, 71(1):101–157, 1994.
16. Christoph Walther, Markus Aderhold, and Andreas Schlosser. The $\mathcal{L}$ 1.0 Primer. Technical Report VFR 06/01, Technische Universität Darmstadt, 2006.
17. Christoph Walther and Stephan Schweitzer. Verification in the classroom. *Journal of Automated Reasoning*, 32(1):35–73, 2004.
18. Christoph Walther and Stephan Schweitzer. Automated termination analysis for incompletely defined programs. In F. Baader and A. Voronkov, editors, *Proc. of LPAR-11*, volume 3452 of *LNAI*, pages 332–346. Springer, 2005.