Termination Analysis by Inductive Evaluation[*]

Jürgen Brauburger and Jürgen Giesl

FB Informatik, TU Darmstadt, Alexanderstraße 10, 64283 Darmstadt, Germany
E-mail: {brauburger, giesl}@informatik.tu-darmstadt.de

Abstract. We present a new approach for automatic termination analysis of functional programs. Several methods have been presented which try to find a well-founded ordering such that the arguments in the recursive calls are smaller than the corresponding inputs. However, previously developed approaches for automated termination analysis often disregard the *conditions* under which the recursive calls are evaluated. Hence, the existing methods fail for an important class of algorithms where the necessary information for proving termination is 'hidden' in the conditions. In this paper we develop the *inductive evaluation* method which analyzes the auxiliary functions occurring in the conditions of the recursive calls. We also discuss an extension of our method to *partial* functions in order to determine their domains automatically. The proposed technique proved successful for termination analysis of numerous algorithms in functional as well as imperative programming languages.

1 Introduction

Proving termination is a central problem in the development of correct software. While most work on the automation of termination proofs has been done for *term rewriting systems* (for surveys see e.g. [Der87,Ste95]) and for *logic programs* (e.g. [UvG88,Plü90,SD94]), in this paper we consider *functional programs.*

A well-known method for termination proofs of LISP functions has been implemented in the NQTHM system of R. S. Boyer and J S. Moore [BM79]. To prove that arguments decrease w.r.t. a well-founded ordering, they use a *measure function* $|.|$ which maps data objects t to natural numbers $|t|$. In their approach, for each recursive call $f(r)$ in an algorithm $f(x)$, an *induction lemma* $\Delta \to |r| < |x|$ is required. It asserts that under the condition Δ, the argument of the recursive call has a smaller measure than the input. Now it remains to verify $\psi \to \Delta$ where ψ is the condition under which the recursive call $f(r)$ is performed. While in [BM79] the user has to supply all induction lemmata, the methods in [Wal94b,Gie95c,GWB98] synthesize a certain class of induction lemmata automatically. The technique in [Wal94b] is restricted to one fixed measure function $|.|$, but the approach of [Gie95c,GWB98] also allows an automatic generation of suitable measures by using techniques from the area of *term rewriting systems.*

To synthesize an induction lemma for a recursive call $f(r)$ under the condition ψ, these methods analyze the auxiliary functions occurring in the recursive

[*] Appeared in the *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, Lindau, Germany, LNAI 1421, 1998.

argument r. However, auxiliary functions in the *condition* ψ are ignored at this point. Consequently, the previous approaches often fail if the necessary information for the termination proof is given by the functions called in the conditions.

We illustrate this problem in Sect. 2. In Sect. 3 we present the *inductive evaluation* technique which overcomes this drawback by combining termination analysis with methods for *induction theorem proving.* While in Sect. 3 our aim is to show that a procedure terminates for *each* input, in Sect. 4 the method is generalized for termination analysis of functions which terminate for *some* inputs only. In Sect. 5 the techniques are extended for analysis of more complex procedures and Sect. 6 draws some conclusions.

2 Termination of Functional Programs

We regard an eager first-order functional language with free algebraic data types[1] and pattern matching where the patterns have to be exhaustive and exclusive. As an example consider the data types bool and nat (for naturals). The type bool has the nullary *constructors* true and false and the objects of type nat are built with the constructors 0 and $\mathsf{s} : \mathsf{nat} \to \mathsf{nat}$. The following procedures compute the 'less than or equal' relation for naturals and the subtraction function.

$$\begin{array}{ll} \textbf{function}\ \mathsf{le} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{bool} & \\ \quad \mathsf{le}(\mathsf{0}, v) & = \mathsf{true} \\ \quad \mathsf{le}(\mathsf{s}(u), \mathsf{0}) & = \mathsf{false} \\ \quad \mathsf{le}(\mathsf{s}(u), \mathsf{s}(v)) & = \mathsf{le}(u, v) \end{array} \qquad \begin{array}{ll} \textbf{function}\ \mathsf{minus} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat} & \\ \quad \mathsf{minus}(x, y) = & \mathsf{if}(\ \mathsf{le}(x, y), \\ & \quad \mathsf{0}, \\ & \quad \mathsf{s}(\mathsf{minus}(x, \mathsf{s}(y)))\ ) \end{array}$$

For each data type τ there is a pre-defined conditional function $\mathsf{if} : \mathsf{bool} \times \tau \times \tau \to \tau$. These conditionals are the only functions with non-eager semantics, i.e. when evaluating $\mathsf{if}(\psi, t_1, t_2)$, the boolean term[2] ψ is evaluated first and depending on the result of its evaluation either t_1 or t_2 is evaluated afterwards.

An algorithm f is terminating if the inputs are 'greater' than the arguments of the recursive calls. For instance, termination of le can be shown by inventing a measure function $|.|$ satisfying le's *termination hypothesis*

$$|u, v| < |\mathsf{s}(u), \mathsf{s}(v)|. \tag{1}$$

We only regard universally quantified formulas of the form $\forall \ldots \varphi$, where we omit the quantifiers to ease readability. Now 'verification of φ' means proving that φ evaluates to true for all instantiations of its variables with data objects. For termination proofs we extend the specification by a new data type weight, new measure function symbols $|.| : \tau \to \mathsf{weight}$ for each data type $\tau \neq \mathsf{weight}$, and a function symbol $< : \mathsf{weight} \times \mathsf{weight} \to \mathsf{bool}$. To compare *tuples* of terms, we also introduce new n-ary tuple symbols $\mathsf{tuple}_n : \mathsf{weight} \times .. \times \mathsf{weight} \to \mathsf{weight}$ for each $n \in \mathbb{N}$, where we often write $|t_1, .., t_n|$ instead of $\mathsf{tuple}_n(|t_1|, .., |t_n|)$. In

[1] See [NN96,Sen96,PS97] for extensions of termination analysis to higher-order languages, languages with lazy evaluation strategy, and to non-free algebraic data types.

[2] We use Greek letters φ, ψ, ω to denote boolean terms and often refer to them as 'formulas', where $\neg, \wedge, \vee, \to$ are pre-defined boolean functions with obvious semantics.

the following, we restrict ourselves to interpretations where the universe for objects of type weight is the set of natural numbers $\mathbb{N}$ and where $<$ is interpreted as the usual 'less than' relation on $\mathbb{N}$. For data types $\tau \neq \mathsf{weight}$ the corresponding universe must correspond to the set of all constructor ground terms and moreover, all defining equations of the algorithms must be valid (i.e. we only consider *inductive models* of the specification, cf. e.g. [ZKK88,Wal94a,BR95]).

In the area of term rewriting systems, several techniques have been developed to generate suitable interpretations for the measure function symbols $|.|$. For example, termination of le can be proved by using an appropriate *polynomial norm* $|.|_{\text{POL}}$ [Lan79]. A polynomial norm is defined by associating each n-ary constructor c with an n-ary polynomial $\text{POL}(c)$ with integer coefficients. In this way, each data object $c(t_1,..,t_n)$ is mapped to a number, i.e. $|c(t_1,..,t_n)|_{\text{POL}} = \text{POL}(c)(|t_1|_{\text{POL}},..,|t_n|_{\text{POL}})$, where we always demand that the choice of the coefficients ensures that data objects are only mapped to *non-negative* integers. For example, if $|\mathsf{0}|_{\text{POL}} = 0$ and $|\mathsf{s}(u)|_{\text{POL}} = |u|_{\text{POL}} + 1$, then each data object of type nat is mapped to a natural number, and we have $|u|_{\text{POL}} < |\mathsf{s}(u)|_{\text{POL}}$ since $|u|_{\text{POL}}$ is smaller than $|u|_{\text{POL}} + 1$ for each natural number $|u|_{\text{POL}}$.

Similarly, tuple symbols are also associated with polynomials mapping $\mathbb{N}^n$ to $\mathbb{N}$. If $\mathsf{tuple}_2(x,y)$ is associated with $x+y$, then we have $|u,v|_{\text{POL}} < |\mathsf{s}(u),\mathsf{s}(v)|_{\text{POL}}$, as $|u|_{\text{POL}} + |v|_{\text{POL}} < |u|_{\text{POL}} + 1 + |v|_{\text{POL}} + 1$ holds for all naturals $|u|_{\text{POL}}$ and $|v|_{\text{POL}}$.

If $|.|$ and tuple_2 are interpreted according to the above polynomial norm, then the termination hypothesis (1) is satisfied. Consequently, termination of le is proved. Techniques to generate suitable polynomial norms automatically have been developed in [Ste94,Gie95a], for instance.

Let t^* and r^* denote tuples of terms $t_1,..,t_n$ and $r_1,..,r_n$. To prove termination of f, for every recursive call in a defining equation $f(t^*) = \ldots f(r^*) \ldots$ we build a *termination hypothesis* $\psi \to |r^*| < |t^*|$ where ψ is the condition under which the recursive call $f(r^*)$ is evaluated. (We restrict ourselves to algorithms without recursive calls in conditions.) For instance, the recursive call of minus is evaluated under the condition $\neg\mathsf{le}(x,y)$. So minus' termination hypothesis is

$$\neg\mathsf{le}(x,y) \to |x,\mathsf{s}(y)| < |x,y|. \tag{2}$$

Termination of a functional program f is proved if one finds a polynomial norm such that *all* termination hypotheses of f are satisfied.

As the termination hypothesis (1) of le only contains terms built with constructors, a suitable polynomial norm can easily be generated automatically. However, this is not possible for the termination hypothesis (2) of minus. The reason is that minus calls another algorithm, le. In contrast to (1), the inequality $|t_1,\mathsf{s}(t_2)| < |t_1,t_2|$ in minus' termination hypothesis does not have to be satisfied for *all* data objects t_1 and t_2, but only for those where $\neg\mathsf{le}(t_1,t_2)$ is true. To determine these data objects we have to consider the *semantics* of the algorithm le. However, the existing methods for the automated generation of polynomial norms can only be used for termination proofs if the termination hypotheses do not contain *defined symbols*, i.e. function symbols defined by algorithms.

3 Termination Proofs with Inductive Evaluation

To enable automatic termination proofs for algorithms like minus, in this section we develop a calculus which transforms termination hypotheses like (2) into formulas *without* defined symbols. Our calculus operates on pairs $H; C$ where H is a set of formulas possibly containing defined symbols (the hypotheses) and C is a set of formulas without defined symbols (the constraints). The soundness of our calculus guarantees that if $H; C$ can be transformed into $H'; C'$, then every interpretation (of the form as described in Sect. 2) satisfying $H' \cup C'$ also satisfies $H \cup C$. For termination proofs, we initialize H to be the set of termination hypotheses and we let C be empty. Then rules of the calculus are applied repeatedly until we result in a pair $H'; C'$ where the first component H' is empty. By the soundness of the calculus, to prove the termination of the algorithm now it suffices to find a polynomial norm satisfying the constraints C'.

An obvious solution to eliminate the defined symbol le from minus' termination hypothesis is to omit its premise, i.e. we could use the following rule[3].

Premise Elimination	$$\frac{H \cup \{\psi \to \omega\}\ ;\ C}{H\ ;\ C \cup \{\omega\}}$$	if ω does not contain any defined symbols.

This is a sound transformation technique, because every interpretation satisfying ω for *all* instantiations of its variables will also satisfy ω for those instantiations which meet the condition ψ, i.e. $\psi \to |r^*| < |t^*|$ may indeed be transformed into $|r^*| < |t^*|$. Moreover, we also allow the application of this rule if the premise ψ is missing, i.e. for an algorithm like le, the unconditional termination hypothesis (1) can be directly inserted into the set of constraints.

For the termination proof of minus, we would initialize H to be $\{(2)\}$ and C to be empty. Then one application of the premise elimination rule transforms H into the empty set and C into $\{|x, \mathsf{s}(y)| < |x, y|\}$. However, in our example this naive solution cannot be used, because this constraint is unsatisfiable. Hence, the termination of minus cannot be proved if the premise of its termination hypothesis is neglected. For that reason, all previous approaches for automatic termination proofs fail with this example.

To enable termination proofs for algorithms like minus we now introduce a new rule which *evaluates* the auxiliary functions in the premises of termination hypotheses. To construct a set of constraints sufficient for the termination hypothesis (2) we use an *induction* w.r.t. the definition of the algorithm le. The base cases of this inductive construction correspond to le's non-recursive defining equations and the step case results from le's recursive (third) equation.

[3] In order to obtain constraints without defined symbols, this rule may only be applied if ω contains no calls of auxiliary algorithms. Hence, in this paper we restrict ourselves to termination hypotheses $\psi \to \omega$ where defined symbols may only occur in the *condition* ψ. For algorithms with defined symbols in the arguments of recursive calls, the technique of the present paper is extended by the calculus of [Gie95c,Gie97,GWB98] to eliminate the remaining defined symbols from the *conclusion* ω.

First we perform a case analysis w.r.t. le, i.e. the variables x, y in (2) are instantiated by the patterns of le's defining equations. Instead of (2) we demand

$$\neg\mathsf{le}(\mathsf{0}, v) \;\to\; |\mathsf{0}, \mathsf{s}(v)| < |\mathsf{0}, v|, \tag{3}$$

$$\neg\mathsf{le}(\mathsf{s}(u), \mathsf{0}) \;\to\; |\mathsf{s}(u), \mathsf{s}(\mathsf{0})| < |\mathsf{s}(u), \mathsf{0}|, \tag{4}$$

$$\neg\mathsf{le}(\mathsf{s}(u), \mathsf{s}(v)) \;\to\; |\mathsf{s}(u), \mathsf{s}(\mathsf{s}(v))| < |\mathsf{s}(u), \mathsf{s}(v)|. \tag{5}$$

In order to detect redundant cases, in each resulting formula we now check whether the premise is unsatisfiable. For example, (3) may be omitted as its negated premise $\neg\neg\mathsf{le}(\mathsf{0}, v)$ can be verified by evaluation of le and $\neg$. As the premises of (3) and (4) are satisfiable, the corresponding proofs must fail.

In the third case (5) we use that each le-call produces a finite sequence of recursive calls. Hence, we assume as an *induction hypothesis* that the termination hypothesis (2) is true for the arguments u and v of le's recursive call, i.e.

$$\neg\mathsf{le}(u, v) \;\to\; |u, \mathsf{s}(v)| < |u, v|. \tag{6}$$

To apply the induction hypothesis, we check if the premise of the induction conclusion (5) entails the premise of the induction hypothesis (6), i.e. we prove

$$\neg\mathsf{le}(\mathsf{s}(u), \mathsf{s}(v)) \to \neg\mathsf{le}(u, v). \tag{7}$$

Again, the proof is trivial since $\mathsf{le}(\mathsf{s}(u), \mathsf{s}(v))$ evaluates to $\mathsf{le}(u, v)$. For that reason we may now *apply* the induction hypothesis (6), i.e. instead of (5) we demand

$$\neg\mathsf{le}(\mathsf{s}(u), \mathsf{s}(v)) \;\wedge\; |u, \mathsf{s}(v)| < |u, v| \quad\to\quad |\mathsf{s}(u), \mathsf{s}(\mathsf{s}(v))| < |\mathsf{s}(u), \mathsf{s}(v)|. \tag{8}$$

The existing techniques for generating polynomial norms expect a set of *inequalities* as constraints, i.e. they cannot treat constraints like $|r_1^*| < |t_1^*| \to |r_2^*| < |t_2^*|$. To eliminate the inequality in the premise of (8) we use that we restricted ourselves to interpretations where naturals are compared by the usual 'less than' relation. For arbitrary naturals k, l, m, n the conjecture $[m + l \leq n + k] \to [k < l \to m < n]$ holds. Hence, instead of (8) we may demand

$$\neg\mathsf{le}(\mathsf{s}(u), \mathsf{s}(v)) \quad\to\quad |\mathsf{s}(u), \mathsf{s}(\mathsf{s}(v))| + |u, v| \;\leq\; |\mathsf{s}(u), \mathsf{s}(v)| + |u, \mathsf{s}(v)|. \tag{9}$$

Here, $+$ and $\leq$ are new function symbols (on weight) and we require that all interpretations map $+$ to the addition and $\leq$ to the 'less than or equal' relation.

In this way, the termination hypothesis (2) can be transformed into the formulas (4) and (9). By eliminating their premises (using the premise elimination rule), we obtain two constraints without defined symbols. Therefore we can now apply the existing techniques to generate a polynomial norm satisfying these constraints. For example, we may use the polynomial norm where $|\mathsf{0}|_{\text{POL}} = 0$, $|\mathsf{s}(u)|_{\text{POL}} = |u|_{\text{POL}} + 1$, and $\mathsf{tuple}_2(x, y)$ is associated with $(x - y)^2$. (Note that this is a legal polynomial norm, because all tuples of data objects are mapped to non-negative numbers[4].) Hence, termination of minus is proved.

[4] In contrast to conventional termination proofs of *term rewriting systems*, for *functional programs* one may use orderings which are not even weakly monotonic, cf.

Recall that during the transformation of minus' termination hypothesis we had to verify the formulas $\neg\neg\mathsf{le}(\mathsf{0}, v)$ and (7). To perform the required proofs, we simply applied *symbolic evaluation*, i.e. we used the defining equations as rewrite rules. In general this verification could require an *induction theorem proving system*, e.g. [BM79,ZKK88,Bun$^+$89,Wal94a,BR95,HS96]. However, when testing our method on numerous algorithms, we found that in almost all examples the required conjectures could already be proved by symbolic evaluation.

Let $\psi \to |r^*| < |t^*|$ be a termination hypothesis containing at least the pairwise different variables $y_1, .., y_n$ of the data types $\tau_1, .., \tau_n$. Moreover, let $g : \tau_1 \times .. \times \tau_n \to \tau$ be defined by a terminating algorithm with k defining equations. To ease the presentation, we restrict ourselves to functions g which are defined without using the conditional if (for an extension see Sect. 5).

Then we use the following rule for induction w.r.t. the recursions of g and subsequent evaluation. In this rule, for any terms $p, s_1, \ldots, s_n$ let $p[s^*]$ be an abbreviation for $p[y_1/s_1, \ldots, y_n/s_n]$, i.e. $p[s^*]$ abbreviates $\sigma(p)$ where σ substitutes each y_i by s_i. Moreover, throughout the paper we always assume that the variables occurring in different algorithms are disjoint.

Inductive Evaluation $\qquad \dfrac{H \cup \{\psi \to |r^*| < |t^*|\}\ ;\ C}{H \cup \{\varphi_1, .., \varphi_k\}\ ;\ C}$

If $g(s^*) = q$ is the i-th defining equation of g, then φ_i is defined as follows:

- $\varphi_i := \mathsf{true}$	if $\neg\psi[s^*]$ can be verified,
- $\varphi_i := \psi[s^*] \to$ $\quad |r^*[s^*]| + |t^*[q^*]| \leq |t^*[s^*]| + |r^*[q^*]|$	$\Big\}$ otherwise, if q contains a term $g(q^*)$ where $\psi[s^*] \to \psi[q^*]$ can be verified,
- $\varphi_i := \psi[s^*] \to |r^*[s^*]| < |t^*[s^*]|$	otherwise

In our example, for le's first equation we have $s^* = \big(\mathsf{0}, v\big)$ and φ_1 is true, as $\neg\psi[s^*]$ (i.e. $\neg\neg\mathsf{le}(\mathsf{0}, v)$) can be verified. Similarly, φ_2 is (4), i.e. here s^* is $\big(\mathsf{s}(u), \mathsf{0}\big)$. For le's recursive equation we have $s^* = \big(\mathsf{s}(u), \mathsf{s}(v)\big)$ and $q^* = \big(u, v\big)$. As the condition (7) could be proved, the resulting formula φ_3 is (9). The following theorem proves that our rule performs a Noetherian induction, since it only allows inductions w.r.t. functions g whose termination has been proved before.

Theorem 1. *If $H; C$ can be transformed into $H'; C'$ by premise elimination and inductive evaluation, then we have $H' \cup C' \models H \cup C$.*

Proof. The soundness of premise elimination is obvious. For inductive evaluation, recall that we restricted ourselves to inductive models I. Now assume that $I \models \varphi_i = \mathsf{true}$ for all i, but there exists a counterexample, i.e. a tuple of constructor ground terms p^* such that $I \models \psi[p^*] = \mathsf{true}$ and $I \not\models |r^*[p^*]| < |t^*[p^*]| = \mathsf{true}$. Let $\prec_g$ be the relation where $p_2^* \prec_g p_1^*$ holds iff evaluation of $g(p_1^*)$ leads to

[AG97]. In fact, termination of the algorithm minus from Sect. 2 cannot be proved by any monotonic well-founded ordering. For that reason, in our approach we restricted ourselves to orderings based on polynomial norms, as most other classes of orderings (that are amenable to automation) possess the monotonicity property.

evaluation of $g(p_2^*)$. Then by termination of g we know that $\prec_g$ is well founded. Hence, we may choose p^* to be a *minimal* counterexample w.r.t. $\prec_g$.

There is a defining equation $g(s^*) = q$ (say, the i-th) such that $p^* = \sigma(s^*)$ for some σ. Obviously, $\psi[s^*]$ is satisfiable as its instantiation $\psi[p^*]$ is valid in I. Hence, $\varphi_i \neq \mathsf{true}$. Thus, q must contain a subterm $g(q^*)$ where $\psi[s^*] \to \psi[q^*]$ can be verified. Then $I \models \psi[p^*] = \mathsf{true}$ implies $I \models \psi[\sigma(q^*)] = \mathsf{true}$. But as $I(\sigma(q^*)) \prec_g p^*$, due to the minimality of p^*, $I(\sigma(q^*))$ cannot be a counterexample. Hence, we have $I \models |r^*[\sigma(q^*)]| < |t^*[\sigma(q^*)]| = \mathsf{true}$. But then $I \models \varphi_i = \mathsf{true}$ implies that p^* is no counterexample either, which leads to a contradiction. $\square$

To select suitable functions g for inductive evaluation, we use a well-known heuristic from induction theorem proving. For a termination hypothesis $\psi \to |r^*| < |t^*|$, we check whether ψ contains a subterm $g(y_1, .., y_n)$ where y_i are pairwise different variables. Such a term suggests an induction w.r.t. g using $y_1, .., y_n$ as induction variables, cf. e.g. [BM79,ZKK88,Bun$^+$89,Wal94a]. Hence, for the termination hypothesis of minus this heuristic suggests inductive evaluation w.r.t. le using the induction variables x and y. Further refined heuristics to choose among several suggested induction relations can be found in [Gie95b].

Inductive evaluation is used for algorithms where the *conditions* of recursive calls have to be analyzed in order to prove termination. In particular, this holds for algorithms like minus where some value is repeatedly increased until it reaches some bound. This class of algorithms is also used extensively in *imperative* programming languages. A straightforward approach to prove termination of imperative programs is to transform them into functional ones and to verify termination of the resulting functions, cf. e.g. [Hen80,GWB98]. For example, the imperative program '$r := 0;$ *while* $x > y$ *do* $y := y+1;$ $r := r+1$ *od*' is transformed into a function whose termination can be proved analogously to minus. Hence, inductive evaluation is particularly useful when extending termination analysis to imperative programs, cf. [BG98].

4 Termination Analysis for Partial Functions

Up to now we tried to prove that an algorithm terminates *totally*, i.e. for *each input.* In the following, we also regard procedures which terminate for *some* inputs only. For example, consider the data type list with the constructors empty and $\bullet$: nat $\times$ list $\to$ list where $x{\bullet}y$ represents the insertion of the number x in front of the list y. Then nextindex(x, y, z) returns the smallest index $i \geq x$ such that z is the i-th element of the list y (where the first element has index 0).

```
function nth : nat × list → nat
  nth(u, empty)  = 0
  nth(0, v•w))   = v
  nth(s(u), v•w) = nth(u, w)

function eq : nat × nat → bool
  eq(0, 0)       = true
  eq(0, s(v))    = false
  eq(s(u), 0)    = false
  eq(s(u), s(v)) = eq(u, v)

function nextindex : nat × list × nat → nat
  nextindex(x, y, z) = if( eq(nth(x, y), z), x, nextindex(s(x), y, z) )
```

Let '$u \bullet v \bullet w$' abbreviate '$u \bullet (v \bullet w)$'. Hence, $\mathsf{nextindex}(2, 5 \bullet 6 \bullet 3 \bullet 5 \bullet 7 \bullet 5 \bullet \mathsf{empty}, 5) = 3$. While termination of nth and eq can easily be proved, $\mathsf{nextindex}(x, y, z)$ only terminates iff z occurs in y at a position whose index is greater than or equal to x or if $z = \mathsf{0}$ (as $\mathsf{nth}(x, y) = \mathsf{0}$ whenever x is not an index of y). Thus, evaluation of $\mathsf{nextindex}(2, 5 \bullet 6 \bullet 3 \bullet 5 \bullet 7 \bullet 5 \bullet \mathsf{empty}, 6)$ does not halt.

4.1 Termination Predicates

To represent subsets of inputs where procedures like nextindex terminate, in [BG96] we introduced termination predicates. An n-ary boolean function θ_f is a *termination predicate* for an n-ary function f iff θ_f is total and if $\theta_f(t_1, .., t_n) = \mathsf{true}$ implies that evaluation of $f(t_1, .., t_n)$ halts. Our aim is to synthesize termination predicates which return true as often as possible, but of course in general this goal cannot be reached as the domains of functions are undecidable.

In [BG96,GWB98], rules for the synthesis of termination predicates are developed. Given an algorithm f and a measure function $|.|$ these rules generate a procedure for θ_f such that $\theta_f(t_1, .., t_n)$ returns true *iff* for $f(t_1, .., t_n)$

(i) the sequence of arguments of (recursive) f-calls decreases under $|.|$ and

(ii) $\theta_g(r_1, .., r_n)$ holds for each resulting auxiliary function call $g(r_1, .., r_n)$.

For example, given $|.|$ the following procedure is synthesized for $\theta_{\mathsf{nextindex}}$.

```
function $\theta_{\mathsf{nextindex}}$ : $\mathsf{nat} \times \mathsf{list} \times \mathsf{nat} \to \mathsf{bool}$
  $\theta_{\mathsf{nextindex}}(x, y, z) = \mathsf{if}(\,\mathsf{eq}(\mathsf{nth}(x, y), z),\ \mathsf{true},\ \mathsf{if}(\,|\mathsf{s}(x), y, z| < |x, y, z|,$
                                                      $\theta_{\mathsf{nextindex}}(\mathsf{s}(x), y, z),$
                                                      $\mathsf{false}\,)\,)$
```

The procedure $\theta_{\mathsf{nextindex}}$ satisfies (i), since under the condition $\neg\mathsf{eq}(\mathsf{nth}(x, y), z)$ of the only recursive call in nextindex, $\theta_{\mathsf{nextindex}}$ returns true iff the arguments of this recursive call decrease (i.e. $|\mathsf{s}(x), y, z| < |x, y, z|$) and if the arguments of the subsequent recursive nextindex-calls decrease under $|.|$, too (i.e. $\theta_{\mathsf{nextindex}}(\mathsf{s}(x), y, z)$). Furthermore, (ii) is satisfied since the only auxiliary functions, nth and eq, are total. As the constructors also denote total functions, we may neglect their termination predicates. Note that $\theta_{\mathsf{nextindex}}$ terminates totally by construction since it is called recursively only if the arguments decrease under $|.|$.

4.2 Inductive Evaluation for Partial Functions

The synthesis of termination predicates described in [BG96] requires the user to provide a measure function $|.|$. To get independent from this input, our aim is an automated generation of suitable polynomial norms for termination predicates, such that the corresponding termination hypotheses are satisfied 'as often as possible'[5]. In order to find a suitable choice for the measure $|.|$ in the termination

[5] For algorithms with auxiliary functions in the recursive *arguments* (instead of the *conditions*), the techniques developed for total termination [Gie95c] can be adapted to partial functions [Bra97], cf. [GWB98].

predicate algorithm $\theta_{\mathsf{nextindex}}$, consider the termination hypothesis of nextindex,

$$\neg\mathsf{eq}(\mathsf{nth}(x,y),z) \to |\mathsf{s}(x),y,z| < |x,y,z|. \tag{10}$$

The formula (10) cannot be transformed into satisfiable constraints, since then total termination of nextindex would be falsely proved. However, we can use inductive evaluation to generate an interpretation that satisfies (10) for a *maximal* number of instances. In this way, we finally obtain a termination predicate for nextindex that is true as often as possible.

In general, the set H of termination hypotheses is transformed into a (possibly empty) set C of inequalities that are satisfied by a polynomial norm. For that purpose we now also use *unsound* transformation rules. However, based on the faulty transformation, for each termination hypothesis $\psi \to |r^*| < |t^*|$ containing the variables x^*, a *soundness predicate* λ is generated such that every interpretation satisfying C also satisfies the restricted termination hypothesis $\lambda(x^*) \wedge \psi \to |r^*| < |t^*|$. So the soundness predicate λ indicates for which data objects the transformation of $\psi \to |r^*| < |t^*|$ into the constraints C is correct[6]. Hence, if we interpret $|.|$ by a polynomial norm satisfying the obtained constraints C, then we can modify the termination predicate algorithm and replace the inequality $|r^*| < |t^*|$ by the corresponding soundness predicate $\lambda(x^*)$.

For instance, to find a suitable measure function for $\theta_{\mathsf{nextindex}}$ we transform the termination hypothesis (10). To exploit the premise $\neg\mathsf{eq}(\mathsf{nth}(x,y),z)$ our heuristic suggests inductive evaluation w.r.t. nth. According to the definition of nth we have to consider two base cases and one step case. For none of these cases the premise is unsatisfiable. In the third case the induction hypothesis may be applied as the formula $\neg\mathsf{eq}(\mathsf{nth}(\mathsf{s}(u),v\bullet w),z) \to \neg\mathsf{eq}(\mathsf{nth}(u,w),z)$ can be proved by symbolic evaluation. Thus inductive evaluation and subsequent premise elimination transform (10) into the following inequalities.

$$|\mathsf{s}(u),\mathsf{empty},z| < |u,\mathsf{empty},z| \tag{11}$$

$$|\mathsf{s}(0),v\bullet w,z| < |0,v\bullet w,z| \tag{12}$$

$$|\mathsf{s}(\mathsf{s}(u)),v\bullet w,z| + |u,w,z| \le |\mathsf{s}(u),v\bullet w,z| + |\mathsf{s}(u),w,z| \tag{13}$$

Of course, (11)-(13) are unsatisfiable, since nextindex is not totally terminating. Hence, we do no longer demand *all* inequalities but we select a satisfiable *subset* of (11)-(13). For instance, if (11) is rejected, then (12) and (13) are satisfied by the polynomial norm where $|\mathsf{empty}|_{\mathrm{POL}} = |0|_{\mathrm{POL}} = 0$, $|\mathsf{s}(u)|_{\mathrm{POL}} = |u|_{\mathrm{POL}} + 1$, $|v\bullet w|_{\mathrm{POL}} = |w|_{\mathrm{POL}} + 1$, and $\mathsf{tuple}_3(x,y,z)$ is associated with $(y-x)^2$.

In general, our aim is to find a maximal satisfiable subset of the inequalities and to reject as few inequalities as possible. As the number of hypotheses is always finite, exhaustive search could be used to determine such a maximal set[7].

[6] This is similar to the approach of [Pro96] where a *proof predicate* is generated from an unsound induction proof in order to extend faulty conjectures to valid ones.

[7] Rejection of an inequality means that one suspects that the algorithm does not terminate for *any* input corresponding to this inequality. (Otherwise, this rejection will result in a termination predicate which only describes a subset of the domain.)

Efficiency can be improved if 'probably polynomially satisfiable' inequalities are selected by the heuristics of [Gie95b] which have proved successful in practice.

In our example, we associate the following soundness predicate λ with the faulty transformation of (10) into (12) and (13), where $\lambda(x, y, z)$ is true for all those instantiations of x, y, and z where this transformation is correct.

$$\begin{array}{ll}\textbf{function } \lambda : \mathsf{nat} \times \mathsf{list} \times \mathsf{nat} \to \mathsf{bool} & \\ \quad \lambda(u, \mathsf{empty}, z) & = \mathsf{false} \\ \quad \lambda(\mathsf{0}, v \bullet w, z) & = \mathsf{true} \\ \quad \lambda(\mathsf{s}(u), v \bullet w, z) & = \lambda(u, w, z)\end{array}$$

The case analysis of λ is given by the case analysis of the algorithm nth which has been used for inductive evaluation of (10). The results of λ are created depending on the transformation steps performed during the inductive evaluation. Since inequality (11) for $y = \mathsf{empty}$ has been *rejected*, λ returns false for that case. Analogously, as (12) was kept as a constraint, this results in the value true for $x = \mathsf{0}$ and $y = v \bullet w$. For inputs of the form $\mathsf{s}(u), v \bullet w, z$, the soundness of the transformation depends on the soundness of the transformation for u, w, z, because inequality (13) of the third case has been created by applying the *induction hypothesis*. Hence, in this case the result $\lambda(u, w, z)$ is generated. The procedure λ terminates totally by construction as it is called recursively under the same condition as nth whose total termination has already been verified.

Thus, λ returns true iff the first argument (the natural x) is less than the length of the second argument (the list y). If the inequality $|\mathsf{s}(x), y, z| < |x, y, z|$ in the termination predicate $\theta_{\mathsf{nextindex}}$ is replaced by $\lambda(x, y, z)$, then $\theta_{\mathsf{nextindex}}(x, y, z)$ is true iff z occurs in y at a position $i \geq x$ or if $z = \mathsf{0}$, i.e. we have indeed generated a termination predicate that returns true as often as possible.

To formalize the generation of soundness predicates λ_φ for termination hypotheses φ, we modify the calculus of Sect. 3. The resulting calculus operates on triples $H; C; E$ where the third component E contains the defining equations of the newly synthesized soundness predicates. The correctness of the calculus guarantees that if $H; C; E$ can be transformed into $H'; C'; E'$, then every interpretation satisfying C' and $\lambda_\varphi(x^*) \to \varphi$ for all $\varphi \in H'$ also satisfies C and $\lambda_\varphi(x^*) \to \varphi$ for all $\varphi \in H$. Here, the semantics of λ_φ is given by E'.

To use our calculus for termination proofs, we again initialize H with the termination hypotheses and let C and E be empty. Then the rules of the calculus are applied repeatedly until we have obtained a triple of the form $\emptyset; C'; E'$. Now the defining equations E' of the generated soundness predicates are added to our specification. Then by the correctness of the calculus every interpretation satisfying the constraints C' also satisfies the original termination hypotheses $\varphi \in H$ for those inputs where the corresponding soundness predicates λ_φ return true. Hence, if there exists a polynomial norm satisfying C', then in the definitions of termination predicates each inequality $|r^*| < |t^*|$ may be replaced by the soundness predicate for the termination hypothesis $\psi \to |r^*| < |t^*|$.

In the following rules, let $x_1, .., x_l$ (x^* for short) be the variables in $\psi \to \omega$ of types $\delta_1, .., \delta_l$ and let $\lambda_{\psi \to \omega}$ be a new boolean function symbol with the argument types $\delta_1 \times .. \times \delta_l$. As in Sect. 3 we also allow an application of the next two rules if the condition ψ is missing.

Premise Elimination $$\frac{H \cup \{\psi \to \omega\}\,;\quad C \quad ; E}{H\,;\, C \cup \{\omega\}\,; E \cup \{\lambda_{\psi\to\omega}(x^*) = \mathsf{true}\}}$$
if ω does not contain any defined symbols.

Rejection $$\frac{H \cup \{\psi \to \omega\}\,;\, C\,; E}{H\,;\, C\,; E \cup \{\lambda_{\psi\to\omega}(x^*) = \mathsf{false}\}}$$

In the third rule, let x^* be the variables of $\psi \to |r^*| < |t^*|$ and to ease readability we write λ instead of $\lambda_{\psi\to|r^*|<|t^*|}$. Let the variables $y_1, .., y_n$ of the types $\tau_1, .., \tau_n$ be contained in x^* and let $g : \tau_1 \times .. \times \tau_n \to \tau$ be defined by a terminating algorithm with k equations. Again $p[s^*]$ abbreviates $p[y^*/s^*]$ and moreover, $\lambda[s^*]$ is used as an abbreviation for $\lambda(x^*[y^*/s^*])$, i.e. $\lambda[s^*]$ abbreviates $\sigma(\lambda(x^*))$ where σ substitutes each y_i with s_i but does not change the remaining variables of x^*.

Inductive Evaluation $$\frac{H \cup \{\psi \to |r^*| < |t^*|\}\;;\; C\;;\; E}{H \cup \{\varphi_1, .., \varphi_k\}\;;\; C\;;\; E \cup \{e_1, .., e_k\}}$$
If $g(s^*) = q$ is the i-th defining equation of g, then φ_i and e_i are defined as

- $\left.\begin{array}{l}\varphi_i := \mathsf{true}\\ e_i := \lambda[s^*] = \mathsf{true}\end{array}\right\}$ if $\neg\psi[s^*]$ can be verified,
- $\left.\begin{array}{l}\varphi_i := \psi[s^*] \to |r^*[s^*]| + |t^*[q^*]| \le |t^*[s^*]| + |r^*[q^*]|\\ e_i := \lambda[s^*] = \lambda[q^*] \wedge \lambda_{\varphi_i}(z^*)\end{array}\right\}$ else, if q contains $g(q^*)$ and $\psi[s^*] \to \psi[q^*]$ can be verified,
- $\left.\begin{array}{l}\varphi_i := \psi[s^*] \to |r^*[s^*]| < |t^*[s^*]|,\\ e_i := \lambda[s^*] = \lambda_{\varphi_i}(z^*)\end{array}\right\}$ otherwise.

Here, z^* are the variables occurring in φ_i.

Similar to Sect. 3, the heuristic for the application of these rules is that inductive evaluation should be applied first if possible and otherwise, premise elimination is preferable to rejection.

Using this calculus, the termination hypothesis (10) can be inductively evaluated w.r.t. nth. For nth's first equation, φ_1 is $\neg\mathsf{eq}(\mathsf{nth}(u, \mathsf{empty}), z) \to$ (11) and e_1 is the equation $\lambda_{(10)}(u, \mathsf{empty}, z) = \lambda_{(11)}(u, z)$. Similarly, φ_2 is $\neg\mathsf{eq}(\mathsf{nth}(\mathsf{0}, v{\bullet}w), z)$ $\to$ (12) and e_2 is $\lambda_{(10)}(\mathsf{0}, v{\bullet}w, z) = \lambda_{(12)}(v, w, z)$. Finally, for nth's third equation φ_3 is $\neg\mathsf{eq}(\mathsf{nth}(\mathsf{s}(u), v{\bullet}w), z) \to$ (13) and e_3 is $\lambda_{(10)}(\mathsf{s}(u), v{\bullet}w, z) = \lambda_{(10)}(u, w, z) \wedge \lambda_{(13)}(u, v, w, z)$. As (11) is rejected and as both (12) and (13) are inserted into the constraints using premise elimination, this results in the defining equations $\lambda_{(11)}(u, z) = \mathsf{false}, \lambda_{(12)}(v, w, z) = \mathsf{true}, \lambda_{(13)}(u, v, w, z) = \mathsf{true}$. Hence, by symbolic evaluation one obtains the algorithm λ given at the beginning of the section. The following theorem shows that our calculus is sound.

Theorem 2. *If $H; C; E$ is transformed into $H'; C'; E'$ by our calculus, then we have* $\{\lambda_\varphi(x^*) \to \varphi \mid \varphi \in H'\} \cup C' \cup E' \;\models\; \{\lambda_\varphi(x^*) \to \varphi \mid \varphi \in H\} \cup C \cup E$.

Proof. The soundness of premise elimination and rejection is trivial. For inductive evaluation we proceed as in the proof of Thm. 1. Assume $I \models \lambda_{\varphi_i}(z^*) \to \varphi_i = \mathsf{true}$ and $I \models e_i$ holds for all i, but $I \models \lambda[p^*] = \mathsf{true}$, $I \models \psi[p^*] = \mathsf{true}$, and $I \not\models |r^*[p^*]| < |t^*[p^*]| = \mathsf{true}$ for a minimal counterexample p^*.

Again this implies that for some i-th defining equation $g(s^*) = q$ we have $p^* = \sigma(s^*)$ and q contains a subterm $g(q^*)$ such that $\psi[s^*] \to \psi[q^*]$ can be verified. Thus we have $I \models \psi[\sigma(q^*)] = \mathsf{true}$ and $I \models \lambda[\sigma(q^*)] = \mathsf{true}$ (by $I \models e_i$). As $I(\sigma(q^*))$ is smaller than p^*, it cannot be a counterexample and so we obtain $I \models |r^*[\sigma(q^*)]| < |t^*[\sigma(q^*)]| = \mathsf{true}$. But as $I \models \sigma(\lambda_{\varphi_i}(z^*)) = \mathsf{true}$ (due to $I \models e_i$), p^* cannot be a counterexample either, which is a contradiction. □

This extension of inductive evaluation for termination analysis of partial functions generalizes our first approach, i.e. whenever total termination of f can be verified by the technique of Sect. 3, the technique of the present section can generate a termination predicate θ_f that returns true for each input.

The handling of partial functions is also necessary for termination analysis of *imperative* programs, because when translating imperative programs into functional ones, *while*-loops are often transformed into *partial* functions, as termination of *while*-loops often depends on their contexts, cf. [GWB98].

5 Refinements

In this section we present extensions of our approach which increase its power considerably. As an example regard the following algorithms.

```
function max : list → nat          function add_if_mem : nat × list × list → list
 max(empty)    = 0                  add_if_mem(x, empty, z) = z
 max(u•empty)  = u                  add_if_mem(x, u•y, z)   = if( eq(x, u),
 max(u•v•w)    = if( le(u, v),                                     x•z,
                     max(v•w),                                     add_if_mem(x, y, z) )
                     max(u•w) )
```

```
function sort : nat × list → list
 sort(x, y) = if( eq(x, max(y)), x•empty, add_if_mem(x, y, sort(s(x), y)) )
```

Total termination of max and $\mathsf{add\_if\_mem}$ is easily proved (where $\mathsf{add\_if\_mem}(x, y, z)$ returns $x{\bullet}z$ if x occurs in y and z otherwise[8]). The function $\mathsf{sort}(x, y)$ returns a sorted list composed of all elements of y which are greater or equal to x where multiple occurrences of elements are removed. Hence, $\mathsf{sort}(0, y)$ sorts the entire list y. This procedure terminates iff x is less than or equal to the maximal element of y (where the maximum of empty is 0). Consider sort's termination hypothesis,

$$\neg\mathsf{eq}(x, \mathsf{max}(y)) \to |\mathsf{s}(x), y| < |x, y|. \tag{14}$$

[8] In the algorithm sort, we use $\mathsf{add\_if\_mem}(x, y, \mathsf{sort}(\mathsf{s}(x), y))$ instead of $\mathsf{if}(\mathsf{member}(x, y), x{\bullet}\mathsf{sort}(\mathsf{s}(x), y), \mathsf{sort}(\mathsf{s}(x), y))$ to ease the readability of our presentation.

To construct a soundness predicate for (14) according to our heuristic we have to use inductive evaluation w.r.t. the algorithm max. However, for functions like max which are defined using the conditional if, we now have to refine the inductive evaluation rule. For max' recursive (third) equation, the idea is to perform a case analysis w.r.t. its if-condition $\mathsf{le}(u,v)$. We first add the condition $\mathsf{le}(u,v)$ to the condition of (14) and treat max as if it only had the recursive call $\mathsf{max}(v{\bullet}w)$, cf. (17). Then we add the negated condition $\neg\mathsf{le}(u,v)$ instead and now we only regard the recursive call $\mathsf{max}(u{\bullet}w)$, cf. (18). In each resulting case the premise is satisfiable and in both step cases the induction hypothesis can be applied. Hence by inductive evaluation we obtain the following new hypotheses.

$$\neg\mathsf{eq}(x,\mathsf{max}(\mathsf{empty})) \to |\mathsf{s}(x),\mathsf{empty}| < |x,\mathsf{empty}| \tag{15}$$

$$\neg\mathsf{eq}(x,\mathsf{max}(u{\bullet}\mathsf{empty})) \to |\mathsf{s}(x),u{\bullet}\mathsf{empty}| < |x,u{\bullet}\mathsf{empty}| \tag{16}$$

$$\neg\mathsf{eq}(x,\mathsf{max}(u{\bullet}v{\bullet}w))\wedge\ \mathsf{le}(u,v) \to |\mathsf{s}(x),u{\bullet}v{\bullet}w|+|x,v{\bullet}w| \le |x,u{\bullet}v{\bullet}w|+|\mathsf{s}(x),v{\bullet}w| \tag{17}$$

$$\neg\mathsf{eq}(x,\mathsf{max}(u{\bullet}v{\bullet}w))\wedge\neg\mathsf{le}(u,v) \to |\mathsf{s}(x),u{\bullet}v{\bullet}w|+|x,u{\bullet}w| \le |x,u{\bullet}v{\bullet}w|+|\mathsf{s}(x),u{\bullet}w| \tag{18}$$

The generation of soundness predicates proceeds in an analogous way by building $\lambda_{(14)}(x,y)$ according to the algorithm max, where the results of max are replaced by the corresponding soundness predicates for the new hypotheses. For a formal definition of the inductive evaluation rule for conditional algorithms see [BG98].

```
**function** $\lambda_{(14)}$ : nat × list → bool
  $\lambda_{(14)}(x,\mathsf{empty}) \quad = \lambda_{(15)}(x)$
  $\lambda_{(14)}(x,u{\bullet}\mathsf{empty}) = \lambda_{(16)}(x,u)$
  $\lambda_{(14)}(x,u{\bullet}v{\bullet}w) \quad = \mathsf{if}(\,\mathsf{le}(u,v), \lambda_{(14)}(x,v{\bullet}w) \wedge \lambda_{(17)}(x,u,v,w),$
                                   $\lambda_{(14)}(x,u{\bullet}w) \wedge \lambda_{(18)}(x,u,v,w)\,)$
```

Our heuristic suggests no further inductive evaluation for (15) and (16), since no term $g(y_1,..,y_n)$ with pairwise different y_i occurs in their premises. But then the inequalities in (15) and (16) have to be rejected, since they are unsatisfiable. Thus, both $\lambda_{(15)}$ and $\lambda_{(16)}$ would always be false and hence, the soundness predicate $\lambda_{(14)}$ for sort's termination hypothesis would also return false for each input. Hence, we would obtain an unsatisfiable soundness predicate although evaluation halts for some recursive calls of sort.

To construct a better soundness predicate, we should again perform inductive evaluation on the obtained hypothesis (16). For that purpose the occurring max-term is *symbolically evaluated.* If we replace the term $\mathsf{max}(u{\bullet}\mathsf{empty})$ by its *symbolic value* u then instead of (16) we obtain

$$\neg\mathsf{eq}(x,u) \to |\mathsf{s}(x),u{\bullet}\mathsf{empty}| < |x,u{\bullet}\mathsf{empty}|. \tag{19}$$

Hence, we extend our calculus by an additional *symbolic evaluation rule* which allows to replace a term $g(t^*)$ in a premise by the term r whenever $g(t^*)$ can be evaluated to r, where C and E do not change.

Now for (19) our heuristic suggests another inductive evaluation w.r.t. eq. We use inductive evaluation as often as possible, but to ensure that it is only applied a finite number of times, we never perform inductive evaluation w.r.t. the same algorithm twice. In our example, if we finally reject the hypothesis (15) and the hypothesis resulting from eq's second equation during the inductive evaluation of (19), then the resulting constraints are satisfied by the polynomial norm $|\mathsf{empty}|_{\text{POL}} = |0|_{\text{POL}} = 0$, $|\mathsf{s}(u)|_{\text{POL}} = |u|_{\text{POL}} + 1$, $|v \bullet w|_{\text{POL}} = |v|_{\text{POL}} + |w|_{\text{POL}}$, where $\mathsf{tuple}_2(x, y)$ is associated with $(y - x)^2$. Hence, we generate soundness predicates $\lambda_{(15)}(x) = \mathsf{false}$, $\lambda_{(17)}(x) = \mathsf{true}$, $\lambda_{(18)}(x) = \mathsf{true}$, and for $\lambda_{(16)}$ we obtain a predicate computing the 'less than or equal' relation on naturals, cf. [BG98]. Thus, the soundness predicate $\lambda_{(14)}(x, y)$ for sort's termination hypothesis is true iff y is non-empty and if x is less than or equal to the maximal element of y. Using this soundness predicate we finally obtain the termination predicate procedure

function $\theta_{\mathsf{sort}} : \mathsf{nat} \times \mathsf{list} \to \mathsf{bool}$
$\theta_{\mathsf{sort}}(x, y) = \mathsf{if}(\,\mathsf{eq}(x, \mathsf{max}(y)),\ \mathsf{true},\ \mathsf{if}(\lambda_{(14)}(x, y),\ \theta_{\mathsf{sort}}(\mathsf{s}(x), y),\ \mathsf{false})\,)$.

The procedure θ_{sort} defines the exact domain of sort, i.e. it returns true iff x is less than or equal to the maximal element of y. Hence, in this way a predicate describing the domain of sort can be generated automatically.

6 Conclusion

We have illustrated that termination of many interesting algorithms cannot be verified if the premises of the termination hypotheses are neglected. Therefore, in this paper we presented the *inductive evaluation* method which analyzes auxiliary functions occurring in the conditions of recursive calls. Our calculus transforms termination hypotheses into inequalities such that existing automated methods can be used to check whether they are satisfied by a polynomial norm. In this way, total termination of algorithms can be proved automatically.

Subsequently, we have generalized our approach for analyzing partially terminating procedures. For that purpose our calculus is extended in order to synthesize *soundness predicates* which are used for the construction of termination predicates describing the domain of the function under consideration.

We combined our method to handle auxiliary functions in the *conditions* with techniques to deal with defined functions in the *arguments* of recursive calls [Gie95b,Gie95c,Gie97,GWB98] and implemented it within the induction theorem prover INKA [HS96]. In this way we obtained an extremely powerful approach for automated termination analysis which performed successfully on a large collection of benchmarks (including all 82 algorithms from [BM79], all 60 examples from [Wal94b], and all 92 examples in [Gie95b] and [BG96]).

See [BG98] for a collection of 36 algorithms whose termination behaviour could not be analyzed with any other automatic method up to now, but where inductive evaluation enables termination analysis without user interaction. For all these examples, termination predicates describing the *exact* domains of the functions could be synthesized. We also applied our approach to *imperative programs* by translating them into equivalent functional programs. In this way, in 33 of 45 examples from [Gri81] the exact domain could be determined automatically.

Acknowledgements. We thank the referees for many helpful suggestions. This work was supported by the DFG (focus program 'Deduktion') under grant no. Wa 652/7-2.

References

[AG97] T. Arts and J. Giesl. Proving innermost normalisation automatically. In *Proc. RTA-97*, Sitges, Spain, LNCS 1232, 1997.

[BM79] R. S. Boyer and J S. Moore. *A computational logic.* Academic Press, 1979.

[BR95] A. Bouhoula and M. Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14:189–235, 1995.

[BG96] J. Brauburger and J. Giesl. Termination analysis for partial functions. In *Proc. 3rd Int. Static Analysis Symp.*, Aachen, Germany, LNCS 1145, 1996. Extended version appeared[9] as Report IBN-96-33, TU Darmstadt, 1996.

[BG98] J. Brauburger and J. Giesl. Termination analysis with inductive evaluation. Technical Report IBN-98-47, TU Darmstadt, Germany, 1998[9].

[Bra97] J. Brauburger. Automatic termination analysis for partial functions using polynomial orderings. In *Proc. 4th SAS*, Paris, France, LNCS 1302, 1997.

[Bun$^+$89] A. Bundy, F. van Harmelen, J. Hesketh, A. Smaill, and A. Stevens. A rational reconstruction and extension of recursion analysis. In *Proc. IJCAI '89*, Detroit, USA, 1989.

[Der87] N. Dershowitz. Termination of rewriting. *J. Symb. Comp.*, 3:69-115, 1987.

[Gie95a] J. Giesl. Generating polynomial orderings for termination proofs. In *Proc. RTA-95*, Kaiserslautern, Germany, LNCS 914, 1995.

[Gie95b] J. Giesl. *Automatisierung von Terminierungsbeweisen für rekursiv definierte Algorithmen.* PhD thesis, Infix-Verlag, St. Augustin, Germany, 1995.

[Gie95c] J. Giesl. Termination analysis for functional programs using term orderings. In *Proc. 2nd Int. Static Analysis Symp.*, Glasgow, UK, LNCS 983, 1995.

[Gie97] J. Giesl. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning*, 19:1-29, 1997.

[GWB98] J. Giesl, C. Walther, and J. Brauburger. Termination analysis for functional programs. In W. Bibel and P. Schmitt, editors, *Automated Deduction – A Basis for Applications*, vol. 3. Kluwer Academic Publishers, 1998.

[Gri81] D. Gries. *The science of programming.* Springer-Verlag, New York, 1981.

[Hen80] P. Henderson. *Functional programming.* Prentice-Hall, London, 1980.

[HS96] D. Hutter and C. Sengler. INKA: The next generation. In *Proc. CADE-13*, New Brunswick, USA, LNAI 1104, 1996.

[KZ95] D. Kapur and H. Zhang. An overview of Rewrite Rule Laboratory (RRL). *J. Computer Math. Appl.*, 29:91-114, 1995.

[Lan79] D.S. Lankford. On proving term rewriting systems are noetherian. Memo MTP-3, Math. Dept., Louisiana Tech. Univ., Ruston, USA, 1979.

[NN96] F. Nielson and H. R. Nielson. Operational semantics of termination types. *Nordic Journal of Computing*, 3(2):144–187, 1996.

[9] Available from `http://www.inferenzsysteme.informatik.tu-darmstadt.de/~reports/notes/{ibn-96-33.ps,ibn-98-47.ps}`.

[PS97] S. E. Panitz and M. Schmidt-Schauß. TEA: Automatically proving termination of programs in a non-strict higher-order functional language. In *Proc. 4th Int. Static Analysis Symp.*, Paris, France, LNCS 1302, 1997.

[Plü90] L. Plümer. *Termination proofs for logic programs.* LNAI 446, 1990.

[Pro96] M. Protzen. Patching faulty conjectures. In *Proc. CADE-13*, LNAI 1104, New Brunswick, USA, 1996.

[SD94] D. De Schreye and S. Decorte. Termination of logic programs: The never-ending story. *Journal of Logic Programming*, 19/20:199–260, 1994.

[Sen96] C. Sengler. Termination of algorithms over non-freely generated data types. In *Proc. CADE-13*, New Brunswick, USA, LNAI 1104, 1996.

[Ste94] J. Steinbach. Generating polynomial orderings. *IPL*, 49:85-93, 1994.

[Ste95] J. Steinbach. Simplification orderings: History of results. *Fundamenta Informaticae*, 24:47-87, 1995.

[UvG88] J. D. Ullman and A. van Gelder. Efficient tests for top-down termination of logical rules. *Journal of the ACM*, 35(2):345-373, 1988.

[Wal94a] C. Walther. Mathematical induction. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, vol. 2. Oxford University Press, 1994.

[Wal94b] C. Walther. On proving the termination of algorithms by machine. *Artificial Intelligence*, 71(1):101–157, 1994.

[ZKK88] H. Zhang, D. Kapur, and M. S. Krishnamoorthy. A mechanizable induction principle for equational specifications. In *Proc. CADE-9*, Argonne, USA, LNCS 310, 1988.