TEA: Automatically proving termination of programs in a non-strict higher-order functional language

Sven Eric Panitz, Manfred Schmidt-Schauß

Fachbereich Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt
Germany
e-mail: {panitz,schauss}@informatik.uni-frankfurt.de
Tel: +49 (69)-798-22890 Fax: +49 (69)-798-28919

Abstract. We present TEA, a tool that is able to detect termination of functions written in a non-strict high-level functional programming language like Haskell. Since almost every compiler for lazy functional languages transforms programs into a functional core language, we use such a core language as the source language for the analysis. TEA is able to detect two kinds of termination: nf-termination and lazy termination. Intuitively, nf-termination of f means that given arguments a_i in normal form, the normal order evaluation of the expression $f\ a_1 \ldots a_n$ terminates with a normal form. If an expression evaluates to an infinite normal form, we will call it lazy terminating.
In order to prove nf-termination of an expression, TEA tries to generate a preclosed tableau using abstract reduction and path analysis. The preclosed tableau issues a set of ordering constraints, which have to be satisfied by some Noetherian ordering. A subsystem is called to generate this Noetherian ordering, and to verify that the set of ordering constraint is satisfied. In the case of success, this proves termination of the expression at the root of the tableau. Lazy termination is recognized using a specialized check on a preclosed tableau.
TEA performs the analysis automatically in reasonable time with a success rate of about 90 % for the functions of a real program script written in Haskell.

1 Introduction

It is well-known that there is no program to test all algorithms for termination. Nevertheless it is of course highly desirable to have a tool that automatically checks programs whether they terminate or not. Our main goal is proving automatically termination of functions written in a higher-order lazy functional language like Haskell or Clean. In this paper we present the basic methodology of TEA as well as the promising results, which demonstrate that, to the best of our knowledge, TEA is currently the most powerful termination analyzer for lazy, higher-order polymorphically typed functional languages.

There are methods and tools for proving termination of term rewriting systems [Der87, Ste95, AG97] and for programs written in a strict functional language [Wal91, Wal94, Gie95b, Gie95a] or some early work in the context of an induction prover [BM79].

Termination of term rewriting system T means that for all expressions t all possible reductions terminate. The same holds for strict functional languages: termination of a function f means that for all arguments $a_i, i = 1, \ldots, n$ in normal form, the evaluation of the expression $f\ a_1 \ldots a_n$ terminates, if a strict reduction strategy is used. With the exception of a special evaluation rule for the conditional `if`, this means that all reductions of $f\ a_1 \ldots a_n$ terminate.

In lazy functional languages, the situation is different. There are sensible expressions that do not have a normal form, for example in Haskell, the expression `[1..]` evaluates to the list of all positive integers in ascending order. Extracting the first element using the function `hd` of the list returns the integer 1, and hence the expression `hd [1..]` is terminating in Haskell [HAB$^+$96], but from the view of term rewriting or strict functional languages, this expression is non-terminating, i.e. not strongly normalizing but normalizing, since a normal form exists.

There is only a small amount of relevant work on proving termination of higher-order lazy functional programs. In [SNN94] termination analysis is called totality analysis, but its applicability to lazy functional programs is limited. There is relevant work of termination analysis in the context of partial evaluation for strict higher-order functional languages [AH96]. There is also some work on semantical methods to prove termination [Kah95], without demonstrating it in practice. Abstract interpretation [CC77] when approximating the denotational semantics [AH87, Bur91] does not really help in proving termination, since in general, the approximation of the exact semantics is in the wrong direction.

Since proving "termination for all possible reductions" (strongly normalizing) is far too restrictive for lazy functional languages, we have developed a method for showing that given an n-ary function f and arguments $a_i, i = 1, \ldots, n$ in normal form, the expression $f\ a_1 \ldots a_n$ can be reduced to a normal form. This kind of termination is called *nf-termination*. Other kinds of termination analyses can be encoded as instances of the nf-termination problem.

The method, which we shall use for termination analysis is based on the operational semantics. To prove that a function f is nf-terminating, the method aims at showing that the abstract expression $f\ T_1 \ldots T_n$, where the variables T_i represent normal forms, represents only expressions that can be reduced to a normal form. Based on the operational semantics, this is done by trying to capture all possible normal-order (i.e. outermost) reductions of instances of $f\ T_1 \ldots T_n$. If a finite representation (a closed tableau) together with an admissible well-founded ordering can be constructed, then this proves nf-termination of f. This method is similar to *abstract reduction* and has been used for strictness analysis [Nöc93, SSPS95].

TEA is an implementation of the method of termination analysis, which in every respect has a good performance on a wide range of practical examples (see

[Pan97, Kic97]).
This paper is structured as follows: in section 2 the source language is introduced; in sections 3 and 4 we introduce termination tableaux and ordering tableaux. In section 5 we describe some extensions like lazy termination, in the final section 6 we discuss the performance, the figures, and the applicability of TEA.

2 The source language

Since we assume that the compiler for the functional language translates all function definitions and expressions into a syntactically simpler functional core language (preserving the semantics), we base the termination analysis on a lazy functional core language Λ.

2.1 The Type System

The (rudimentary) type system for Λ consists of a set of types (i.e. names), of corresponding constructors, which have a fixed arity, and a case constant $\mathtt{case}_A$ for every type A. To ease notation and the definition of the operational semantics, the constructors for type A are denoted $c_{A,i}$. We assume that there is a predicate $WT(.)$ that recognizes well-typed Λ-expressions. If we speak of well-typed Λ-expressions e, we mean that $WT(e)$ holds.

2.2 Expressions and Supercombinator Definitions

Λ-expressions are built using the following syntax:

$$e ::= \mathtt{case}_A \mid (e\ e) \mid \langle Var \rangle \mid \langle F \rangle$$

Where Var means a variable name, and F a defined function (a supercombinator). We use the usual bracketing conventions and denote syntactic equality by $\equiv$.
A program script consists of a set of supercombinator definitions of the form

$$\langle \mathtt{F} \rangle\ v_1 \ldots v_n = e$$

where $v_1, \ldots, v_n$ are different variables, and every free variable from e is also contained in $\{v_1, \ldots, v_n\}$. The supercombinator `main` is defined without variables and is the expression to be evaluated. The set of all Λ-expressions is denoted as Λ.

2.3 Operational Semantics

Definition 1. The operational semantics consists of the following rules:

- $\mathtt{F}\ t_1 \ldots t_n \to d[t_1/x_1, \ldots, t_n/x_n]$ provided the definition of F is $F\ x_1 \ldots x_n = d$.

- $\texttt{case}_A (c_{A,i}\ a_1 \ldots a_m)\ t_1 \ldots t_n \to t_i\ a_1 \ldots a_m$, if the arity of $c_{A,i}$ is m and the type A has n constructors.

Definition 2. (normal-forms)
An expression that cannot be reduced any further is in *normal-form*.
An expression is in head normal-form (HNF) iff it has of one the following forms:

- $(c_A\ e_1 \ldots e_n)$ and c is a constructor with $arity(c) \geq n$.
- $(F\ e_1 \ldots e_n)$ and F is a supercombinator with $arity(F) > n$.

We say an expression has a NF (HNF), if it can be reduced to a NF (HNF), respectively. The normal form of t is denoted by $t\Downarrow$, provided it exists.

Definition 3. An expression e is *dynamically ill-typed*, denoted as $IT(e)$, iff e has a subexpression of one of the following forms

- $\texttt{case}_A (a_0\ a_1 \ldots a_m)\ t_1 \ldots t_n$, and a_0 is a constructor with $arity(a_0) \neq m$, or it does not belong to the type A, or a_0 is a supercombinator and $(a_0\ a_1 \ldots a_m)$ is in HNF .
- $(\texttt{c}\ t_1 \ldots t_m)$ and c is a constructor with $arity(c) < m$.

Note that $IT(e)$ is effectively computable.

Definition 4. The well-typed predicate $WT(.)$ on Λ is *admissible*, iff

- $WT(e)$ and $e \to e'$ imply $WT(e')$.
- $IT(e)$ implies that $WT(e)$ is false.
- $WT(e)$ implies $WT(e')$ for every subexpression e' of e.
- If $WT(e)$ and e has a HNF with a supercombinator as head, then there is some expression e' with $WT(e')$ and $WT(e\ e')$.

In the following we assume that an admissible predicate $WT(.)$ is given. Note that WT needn't be tested in the algorithm.
The reduction relation $\to$ has the Church-Rosser property for expressions e with $WT(e)$. Let $\xrightarrow{*}$ be the reflexive, transitive closure of $\to$.

Definition 5. (nf-termination)
The set $\Lambda_{NF} \subset \{e \mid e \in \Lambda, WT(e)\}$ of *nf-terminating* expressions is the smallest set of expressions e with:

- e can be reduced to a normal form containing only constructors or,
- e has a HNF $\texttt{c}\ e_1 \ldots e_n$, where $\texttt{c}$ is a constructor of arity n and all e_i nf-terminate or,
- e has a HNF $\texttt{F}\ e_1 \ldots e_n$, where $\texttt{F}$ is a supercombinator of arity greater than n and for each nf-terminating e_0: $WT(e\ e_0)$ implies that $(e\ e_0)$ nf-terminates.

Note that in lazy functional languages there are sensible expressions that are not nf-terminating: a non-nf-terminating function f may be different from $\perp$ in the standard semantics, an example being `[1..]`, which is the infinite lists of integers in ascending sequence.

3 Termination Analysis of Λ-functions

3.1 Termination Tableaux

Definition 6. The language $\Lambda^{\#}$ of abstract expressions has the same syntax as Λ, extended by abstract variables that represent nf-terminating expressions. We denote these abstract variables by a T, perhaps with some index.

By $AV(e)$, we denote the set of abstract variables in the $\Lambda^{\#}$-expression e.

Definition 7. An expression $e^{\#} \in \Lambda^{\#}$ is said to be in *abstract normal form* if e consists entirely of constructor constants and abstract variables.

The abstract reduction relation $\rightarrow^{\#}$ is obtained by applying the reduction rules in Definition 1 to $\Lambda^{\#}$-expressions.
Abstract expressions represent sets of Λ-expressions, which can be obtained from an abstract expression applying a concretization map:

Definition 8. A *concretization map* is a substitution γ that instantiates abstract variables by nf-terminating Λ-expressions. The result $\gamma(e)$ of the application of γ to an abstract expression e is also called *concretization* of e.

We lift the notion of nf-termination to $\Lambda^{\#}$: $e^{\#} \in \Lambda^{\#}$ nf-terminates iff for all concretization maps γ: $WT(\gamma(e^{\#}))$ implies that $\gamma(e^{\#})$ nf-terminates.
As a notational convention $C[.,\ldots,.]$ will denote a context, i.e. an expression with some holes. Then $C[t_1,\ldots,t_n]$ is an expression where the holes are filled with the expressions t_i. We assume that for fresh variables x_i, every variable x_i occurs exactly once in the expression $C[x_1,\ldots,x_n]$.

Definition 9. *Termination tableaux* are trees such that every node N is marked with an abstract expression t_N. A tableau is *sound*, if for all nodes e: if all direct sons of e nf-terminate then e nf-terminates.

In Figure 1 we introduce rules which expand a termination tableau. The alternatives in the consequent mean that for every alternative a leaf has to be added.

Lemma 10. *The rules in Figure 1 are sound: they transform sound tableaux into sound tableaux.*

There are several different possibilities to decide that a path in a termination tableau requires no further expansion. We permit to incorporate external knowledge of nf-terminating supercombinators or expressions by permitting an external proof procedure $P_{\mathtt{nft}}$. We require that $WT(e)$ and $P_{\mathtt{nft}}(e)$ implies that e is nf-terminating. The external proof procedure may be the termination tableau method itself. A desirable behavior of $P_{\mathtt{nft}}$ is that $P_{\mathtt{nft}}(e)$ holds for all expressions e in abstract normal form.

Definition 11. Closing Rules

Abstract reduction

$$\frac{e}{e'} \qquad \text{if } e \to^{\#} e'$$

Approximation

$$\frac{C[e]}{C[T_i] \mid e} \qquad T_i \text{ is a new variable.}$$

Argument saturation

$$\frac{e}{e\ T_i} \qquad \text{where } T_i \text{ is a new variable.}$$

If e is a HNF where the head is a supercombinator

Branching

$$\frac{C[case_A\ T_1\ e_1 \ldots e_m]}{\begin{array}{l}\{T_1 \mapsto (c_{A,1}\ T_{1,1} \ldots T_{1,n_1})\}C[e_1\ T_{1,1} \ldots T_{1,n_1}] \\ \mid \ \ldots \\ \mid \{T_1 \mapsto (c_{A,m}\ T_{m,1} \ldots T_{m,n_m})\}C[e_m\ T_{m,1} \ldots T_{m,n_m}]\end{array}}$$

Where $T_{i,j}$ are new variables, m is the number of constructors of type A
$c_{A,j}$ are the constructors of the type A, n_j is the arity of $c_{A,j}$
and the `case`-expression is a normal order redex.
The substitution $\{T_1 \mapsto (c_{A,j}\ T_{j,1} \ldots T_{j,n_j})\}$ is also called split-substitution.

Fig. 1. Expansion rules for termination-tableaux.

- If e at a leaf is dynamically ill-typed, then label the leaf as "closed".
- If for e at a leaf $P_{\mathtt{nft}}(e)$ holds, then label the leaf as "closed".
- If the expression t_L at the leaf L is of the form $C[e_1, \ldots e_n]$, where $C[\ldots]$ does not contain abstract variables, $P_{\mathtt{nft}}(e_i)$ holds for $i = 1, \ldots, n$, and there is an ancestor node N labelled $C[T_1, \ldots T_n]$, such that $T_i \equiv T_j \Rightarrow e_i \equiv e_j$, then label L with the pair ("*closed*", N).

A termination tableau in which every leaf is labelled "closed", is called *preclosed.*

The subsuming ancestor nodes are also called *recursion ancestor nodes.* Note that different leaves may have the same recursion ancestor node.

Definition 12. A *termination ordering* is a Noetherian partial order $>$ on $(\Lambda_{NF})^n$ for some n, which is invariant w.r.t. reduction, i.e., If $s \xrightarrow{*} s', t \xrightarrow{*} t'$, then $s > t$ is equivalent to $s' > t'$,

Definition 13. Let $(s_1, \ldots, s_n)$ and $(t_1, \ldots, t_n)$ be n-tuples of $\Lambda^{\#}$-expressions and let R be a relation name. Then $(s_1, \ldots, s_n)\ R\ (t_1, \ldots, t_n)$ is an *ordering constraint.*
A set S of ordering constraints can be partitioned into the sets S_R for all the relation names R occurring in S. The set S_R consists of all ordering constraints in S with the same relation name R.

A component S_R is *satisfiable*, if there is a termination ordering $>_R$ on $(\Lambda_{NF})^n$, where n is the arity of the tuples in S_R such that for all concretization maps γ and for all ordering constraints $(s_1, \ldots, s_n)\ R\ (t_1, \ldots, t_n)$: $WT(\gamma(s_1, \ldots, s_n))$ and $WT(\gamma(t_1, \ldots, t_n))$ imply $\gamma(s_1, \ldots, s_n) >_R \gamma(t_1, \ldots, t_n)$.
A set S of ordering constraints is *satisfiable*, iff for every relation name R, the sets S_R of ordering constraints are satisfiable.

The ordering constraints are also called termination hypotheses in the work on termination of strict functional languages.
Let $\mathcal{T}$ be a preclosed termination tableau. Let L be a leaf with ancestor node N, let the terms be $T_L \equiv C[t_1, \ldots, t_n]$ and $T_N \equiv C[T_1, \ldots, T_n]$ and let σ be the composition of the split-substitutions on the path from N to L. Then the corresponding ordering constraint is: $\sigma(T_1, \ldots, T_n)\ >_N\ (t_1, \ldots, t_n)$. The set of all ordering constraints of $\mathcal{T}$ is $OC(\mathcal{T})$.

Definition 14. Let $\mathcal{T}$ be a preclosed termination tableau, such that for every path from a recursion ancestor node to one of its subsumed leaves, there is no further recursion ancestor node on the path. If moreover for every recursive ancestor node the same context is used for subsuming all corresponding leaves, then we say the preclosed termination tableau is *without crossings*.

Definition 15. A termination tableau $\mathcal{T}$ is called *closed*, if it is preclosed, without crossings and $OC(\mathcal{T})$ is satisfiable.

3.2 Correctness of Termination Tableaux

In this subsection we argue that the method for proving termination is correct. I.e., if a closed tableau can be constructed for an abstract term at the root node, then this term represents only nf-terminating terms.

Lemma 16. *Let e be a non-leaf node in a termination tableau. Let γ be any concretization map such that $WT(\gamma(e))$ and $\gamma(e)$ is not nf-terminating. Then there is a direct son e' of e, such that one of the following holds:*

- *The expansion rule is "Abstract reduction", $WT(\gamma(e'))$ and $\gamma(e')$ is not nf-terminating.*
- *The expansion rule is "Approximation" or "Argument saturation" and there is a concretization map γ' with $\gamma'(T) = \gamma(T)$ for all variables $T \in AV(e)$, $WT(\gamma'(e'))$ and $\gamma'(e')$ is not nf-terminating.*
- *The expansion rule is "Branching", the substitution on the path from e to e' is σ, and there is a concretization map γ' with $\gamma'(\sigma(T_i)) = \gamma(T_i)$ for all variables $T_i \in AV(e)$, $WT(\gamma'(e'))$ and $\gamma'(e')$ is not nf-terminating.*

Proof. By inspecting the rules and using that WT is admissible.

Lemma 17. *Let e be a non-leaf node in a termination tableau, let γ be any concretization map such that $WT(\gamma(e))$ and $\gamma(e)$ is not nf-terminating. Then* *there is a descendant leaf e', where σ is the substitution on the path from e to e', a concretization map δ, such that $\delta(\sigma(T)) = \gamma(T)$ for all $T \in AV(e)$ and such that $WT(\delta(e'))$ and $\delta(e')$ is not nf-terminating*

Proof. This follows from Lemma 16 by induction on the length of the path.

Theorem 18. *Let $\mathcal{T}$ be a closed termination tableau with root t_0. Then t_0 is nf-terminating.*

Proof. Let there be fixed termination orderings $>_N$ on the tupels over Λ_{NF} for every recursion ancestor node N, which satisfy the ordering constraints for the closed tableau. Assume that there is a concretization map γ_0, such that $WT(\gamma_0(t_0))$ and $\gamma_0(t_0)$ is not nf-terminating. Then Lemma 17 shows that there is some leaf with a well-typed but not nf-terminating concretization. This leaf must have a recursion ancestor node. Select a leaf L with term $C[t_1, \ldots, t_n]$ and recursion ancestor N, and a concretization map γ such that $WT(\gamma(C[t_1, \ldots, t_n]))$, $\gamma(C[t_1, \ldots, t_n])$ is not nf-terminating, and the following holds:

- The distance (measured in number of edges) of the root to the ancestor node corresponding to the leaf L is maximal.
- $\gamma(t_1, \ldots, t_n)$ is minimal w.r.t $>_N$.

The concretization $\gamma(C[t_1, \ldots, t_n])$ is also a concretization of the term $(C[T_1, \ldots, T_n])$, i.e., there is a concretization map β, such that $\gamma(C[t_1, \ldots, t_n]) = \beta(C[T_1, \ldots, T_n])$, since all t_i are nf-terminating and $WT(\gamma(t_i))$.
Lemma 17 shows that there is a leaf L' that is a descendant of N, and which has a non-nf-terminating instance. Since $\mathcal{T}$ is without crossings and due to our minimal choice, N is also a recursion ancestor for L' with the same context. Hence the term at L' is $C[t'_1, \ldots, t'_n]$, there is a concretization map δ, such that $\delta(\sigma(T_i)) = \beta(T_i)$ for all $T_i \in AV(t_N)$, and $\delta(C[t'_1, \ldots, t'_n])$ is well-typed and not nf-terminating. The ordering constraints are satisfied, which means $\sigma(T_1, \ldots, T_n) >_N (t'_1, \ldots t'_n)$. This implies $\delta(\sigma(T_1, \ldots, T_n)) >_N \delta(t_1, \ldots t_n)$, and hence $\beta(T_1, \ldots, T_n)) >_N \delta(t'_1, \ldots t'_n)$. We obtain $\gamma(t_1, \ldots, t_n) = \beta(T_1, \ldots, T_n) >_N \delta(t'_1, \ldots t'_n)$. This is a contradiction to the minimal choice of L and γ.
The final contradiction shows that the expression t_0 at the root is nf-terminating.

Since the problem of proving nf-termination is undecidable, (it is in fact non-r.e.), there are cases, where the construction of a preclosed termination tableau for an nf-terminating expression does not succeed. The construction is then non-terminating, since either there are paths without recursion ancestor, or there are recursion ancestors, but with crossings. The restriction to preclosed trableaux without crossings is necessary. For a counterexample see [Pan97].
To illustrate the calculus we will give a simple example:

Example 1. We want to prove that the result of applying the length function to a finite list is a defined number, even if there are undefined elements in the list:

```
bot = bot
mkbot x = bot
fin = map mkbot T

map f xs = case xs Nil (map2 f)
map2 f x xs = Cons (f x) (map f xs)

length xs = case xs 0 length2
length2 length2 x xs = 1 + length xs
```

A preclosed tableau for the application of `length` to finite lists is given in figure 2. The tableau makes use of the fact that the built-in function `+` nf-terminates.

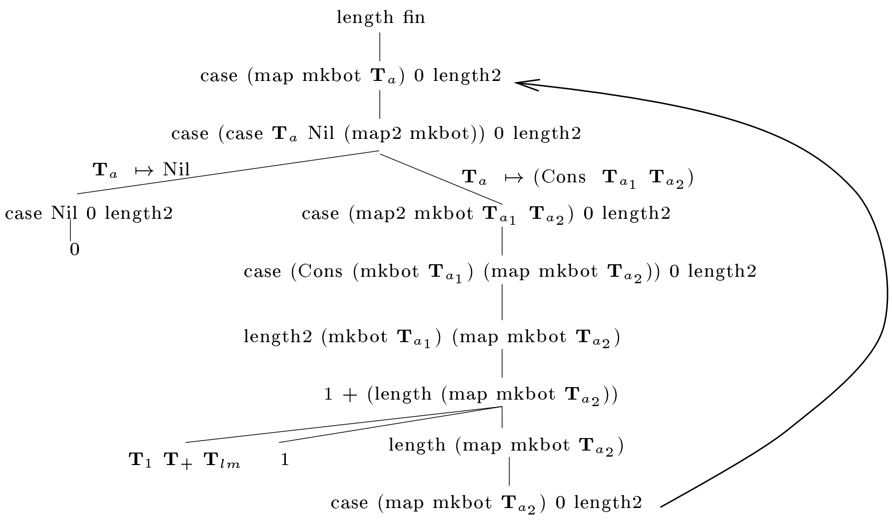

Fig. 2. A preclosed tableau for the application of `length` to finite lists.

4 Termination orderings on Λ

We will interpret constructors, and hence normal forms, as functions over the nonnegative integers. For non-strict functional languages, interpreting supercombinators as integer functions is too restrictive, since this precludes the use of subexpressions without normal form like infinite lists.

An n-ary function p on integers is called *nonnegative*, iff $p(a_1, \ldots, a_n) \geq 0$ for all nonnegative a_i. p is called *strictly monotonous*, iff $y > z$ implies $p(x_1, \ldots, x_{i-1}, y, x_{i+1}, \ldots, x_n) > p(x_1, \ldots, x_{i-1}, z, x_{i+1}, \ldots, x_n)$ for all nonnegative integers x_i, y, z, and all i. Let π be an interpretation of the normal forms, where every constructor c of arity n is interpreted as a nonnegative function that

is strictly monotonous for all arguments at positions in $I_{\mathtt{c},\mathtt{sm}}$ and constant for the arguments in position $[1,n] \setminus I_{\mathtt{c},\mathtt{con}}$. Expressions in HNF that have a supercombinator as head are interpreted as zero. The interpretation π can be extended to all well-typed Λ-expressions t having a normal form by $\pi(t) := \pi(t\Downarrow)$. A termination ordering $>_\pi$ is induced on nf-terminating Λ-expressions by defining $s >_\pi t$ iff $\pi(s) > \pi(t)$.
We describe a proof method which given an interpretation π for the normal forms, shows that the ordering constraint $s\ R\ t$ is satisfied by $>_\pi$.
Though termination tableaux as described in section 3 issue only ordering constraints where the left hand sides consist of a tuple of variables, we permit more general ordering constraints.
TEA makes only use of linear polynomial interpretations with small coefficients. Though this defines only a very restricted class of orderings, Figure 3 shows that this is in practice sufficient.
In the following due to space limitations we will not explore the syntactic nature of ordering tableaux in detail, but rather introduce the semantical ideas and the correctness proof.

4.1 Verifying ordering constraints

We assume a given interpretation π of constructors. The top level tupels in ordering constraints are treated like constructed objects with a special constructor. Given an ordering constraint, the verification algorithm has the task to check that the ordering constraint is satisfied. In this section, we assume that the abstract variables may be instantiable with arbitrary expressions.
We denote the inequations used in the algorithm as
$p_l\langle s_1,\ldots,s_n\rangle \geq p_r\langle t_1,\ldots,t_m\rangle$, where the nonnegative, strictly monotonous functions p_l, p_r are assumed to be known, are of arity n and m, respectively, and the expressions s_i, t_i are $\Lambda^\#$-expressions.

Definition 19. Let f_1, f_2, g_1, g_2 be functions. Let x_i, y_i be variables, where equal variables are permitted. We say $f_1(x_1,\ldots,x_m) \geq f_2(y_1,\ldots,y_n)$ is *valid*, if for all replacements of nonnegative integers for the variables the inequation holds. We say $f_1(x_1,\ldots,x_m) \geq f_2(y_1,\ldots,y_n)$ *implies* $g_1(x'_1,\ldots,x'_m) \geq g_2(y'_1,\ldots,y'_n)$, iff for all replacements ρ of nonnegative integers for the variables: If $f_1(\rho(x_1),\ldots,\rho(x_m)) \geq f_2(\rho(y_1),\ldots,\rho(y_n))$ holds, then $g_1(\rho(x'_1),\ldots,\rho(x'_m)) \geq g_2(\rho(y'_1),\ldots,\rho(y'_n))$ holds.
We say $f_1\langle s_1,\ldots,s_m\rangle \geq f_2\langle t_1,\ldots,t_n\rangle$ is *valid*, if $f_1(x_1,\ldots,x_m) \geq f_2(y_1,\ldots,y_n)$ is valid after replacing the terms s_i, t_i by variables, where equal (different) terms have to be replaced with equal (different) variables.
Similarly, we define $f_1\langle s_1,\ldots,s_m\rangle \geq f_2\langle t_1,\ldots,t_n\rangle$ *implies* $g_1\langle s'_1,\ldots,s'_m\rangle \geq g_2\langle t'_1,\ldots,ty'_n\rangle$ iff $f_1(x_1,\ldots,x_m) \geq f_2(y_1,\ldots,y_n)$ implies $g_1(x'_1,\ldots,x'_m) \geq g_2(y'_1,\ldots,y'_n)$, where equal (different) terms have to be replaced with equal (different) variables.

The verification method is based on abstract reduction to built up a tree of possible reductions (called an *ordering tableau*). I.e., a tree which is labelled with inequations, and which is generated by placing the initial inequation at the root, using expansion and closing rules described below. Initially, the verification algorithm starts with $lhs > rhs$, which is easily translated into $lhs \geq rhs + 1$. The following simplifications are performed with high priority:

1. **simplify-comb:** A left- or right side of the following form is immediately simplified: $p\langle s_1, \ldots, s_{i-1}, (c\ s_{i,1} \ldots s_{i,k}), s_{i+1}, \ldots, s_n\rangle$ simplifies to $q\langle s_1, \ldots, s_{i-1},\ s_{i,j_1}, \ldots, s_{i,j_h}, s_{i+1}, \ldots, s_n\rangle$, where $q(x_1, \ldots, x_{i-1}, x_{i,j_1}, \ldots, x_{i,j_h}, x_{i+1}, \ldots, x_n)$ $:= p(x_1, \ldots, x_{i-1}, p_c(x_{i,1}, \ldots, x_{i,k}), x_{i+1}, \ldots, x_n)$, and $I_{\mathtt{c},\mathtt{sm}} = \{j_1, \ldots, j_h\}$.
2. **simplify-zero:** A left- or right side, where s_i is in HNF but without a constructor as head is immediately simplified: $p\langle s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n\rangle$ simplifies to $q\langle s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n\rangle$, where $q(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) := p(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$.

Note that the defined function q is nonnegative and strictly monotonous.

Definition 20. The expansion rules for expanding the ordering tableau are "abstract reduction" and "branching" (cf. termination tableau) applied to ordering constraints, where the rules are to be applied only to normal order redexes. In the following closing rules we assume that there are sound and effectively computable predicates to recognize "valid" and "implies":

- If one of the $\Lambda^{\#}$-expressions is dynamically ill-typed, then label the leaf as "closed".
- If the inequation at the leaf is valid, then label the leaf as "closed".
- Let L be a leaf labelled with $p_{L,l}\langle s_{L,1}, \ldots, s_{L,n}\rangle \geq p_{L,r}\langle t_{L,1}, \ldots, t_{L,m}\rangle$. Let there be an ancestor N of L, which is labelled with $p_{N,l}\langle s_{N,1}, \ldots, s_{N,n}\rangle \geq p_{N,r}\langle t_{N,1}, \ldots, t_{N,m}\rangle$, such that the following holds:
  There exists a subsuming substitution σ, such that
  $\sigma(s_{N,i}) \equiv s_{L,i}$, and $\sigma(t_{N,j}) \equiv t_{L,j}$ for $i = 1, \ldots, n; j = 1, \ldots, m$.
  The inequation $\sigma(p_{N,l}\langle s_{N,1}, \ldots, s_{N,n}\rangle) \geq \sigma(p_{N,r}\langle t_{N,1}, \ldots, t_{N,m}\rangle)$ implies $p_{L,l}\langle s_{L,1}, \ldots, s_{L,n}\rangle \geq p_{L,r}\langle t_{L,1}, \ldots, t_{L,m}\rangle$.
  Then label the leaf L by (N, "*closed*").

Note that the implementation is also able to exploit symmetries of the functions in the subsumption check.

Definition 21. An ordering tableau is *closed*, iff every leaf is closed.

Now we can show that a closed ordering tableau is a proof that the ordering $>_\pi$ satisfies a given ordering constraint.

Theorem 22. *Let $s > t$ be an ordering constraint. If a closed ordering tableau for $(\lambda z.z)\langle s\rangle \geq (\lambda z.z + 1)\langle t\rangle$ can be constructed using the rules in Definition 20, then $s > t$ is satisfied by the ordering $>_\pi$.*

Proof. Assume that the theorem is false. Then there is a closed ordering tableau for $(\lambda z.z)\langle s\rangle \geq (\lambda z.z+1)\langle t\rangle$, but $s > t$ is not satisfied by $>_\pi$. There is a ground substitution ρ, such that $\rho(s), \rho(t)$ are nf-terminating ground expressions, and $\pi(\rho(s)) < \pi(rho(t)) + 1$.
It is easy to see that the expansion rules of the tableau permit to find a witness of a wrong instance of an inequation at some son, if the father has a wrong instance of its inequation. This is obvious for the reduction, and also for the branching rule, since it is only used for normal order redexes, and it is assumed that witnesses are well-typed. Hence there is some leaf, such that the inequation has a wrong instance. This leaf must be closed by the subsumption rule.
Among the leaves with non-valid inequations select a leaf and a corresponding instance which is minimal w.r.t. the minimal number of normal order reductions to compute its π-value. This number is defined, since we assumed that the instance is nf-terminating at the root, which is inherited to the corresponding leaf.
This leaf has a subsuming ancestor N, which is labelled by a wrong inequation, and which requires the same number of normal order reductions to normal form. We reduce again until a further leaf L' with a wrong instance is found. Since at least one normal order step was performed on the path from N to L', the instance at the new leaf L' requires a smaller number of normal order reductions for the computation. Hence we have a contradiction to the minimal choice.

5 Extensions

We describe some extensions and modifications of the termination analysis.

5.1 Lazy Termination

There are sensible functions in a lazy functional language, like `repeat`, which are not nf-terminating. In order to apply the termination analysis also to these functions, it will turn out that a simple test on a preclosed tableau is a sufficient condition that a function is lazy terminating.

Definition 23. Every well-typed expression is 0-lazy-terminating. A well-typed expression e is n-lazy-terminating, iff either e has a HNF with a supercombinator as head and e is nf-terminating, or e has HNF $(c\ e_1 \ldots\ e_k)$, c is a constructor and every e_i is $(n-1)$-lazy-terminating.
An expression e is *lazy terminating*, iff for all n, e is n-lazy-terminating.
We lift the notion of lazy termination to $\Lambda^\#$ via concretization maps in the natural way.

It is obvious that nf-termination implies lazy termination.
An example for a lazy terminating expression is (`repeat` 1) with definition `repeat` $x = x$ `:` `repeat` x.
The following simple criterion is sufficient for lazy termination:

Theorem 24. *Let e be an expression, and let there be a preclosed termination tableau with root e, such that for every leaf L with subsuming ancestor N, on the path from N to L the only permitted use of the approximation rule is:* $\frac{c\ e_1 \ldots e_n}{e_1 \mid \ldots \mid e_n}$*, where c is a constructor, there is at least one such application on the path, and there is no argument saturation on the path.*
Then e is lazy terminating

Proof. Assume that the theorem is false. Then there is some e, a preclosed tableau and a concretization $\rho(e)$ that is not lazy terminating. Similarly as in the proof of Theorem 18 we can find a leaf that is not lazy terminating. This leaf is closed using using the criterion for a subsuming ancestor, since nf-termination implies lazy termination. We select a leaf L such that there is an instance that is not n-lazy-terminating with minimal n. We follow a witnessing instance: going from L to its subsuming ancestor N, again reducing down to a new leaf L', we see that L' has a smaller witnessing instance, which is a contradiction.

5.2 Errors

Usually, programs contain error-cases, for example taking the head of a list is not defined for the empty list, or division by zero is undefined. The standard behavior in these cases is that the evaluation stops, and an error message is displayed. For semantic purposes, this is often treated as $\perp$, which is also used for non-terminating object. However, from a pragmatic point of view there is a difference: a programmer does not write non-terminating programs intentionally, whereas error-cases are often useful and sensible. Thus, it makes sense to treat error cases as terminating cases, since a program terminates by displaying something. The implementation of TEA usually treats errors as termination and thus it marks the built-in division and missing cases in a case-expression as nf-terminating exits.

5.3 Strict evaluation

Theorem 18 as it stands is not applicable for a functional language with strict evaluation. However, the construction of termination tableaux can be modified, such that also strict (nf-)termination can be proved. The only necessary modification is to restrict all reductions to applicative order reductions, in the expansion of the termination tableau as well as for the ordering tableau. Since the behavior of normal order and applicative order reduction are different, the heuristics and strategies have to be different: The rule for approximation has to be preferred in several cases in order to make redexes top level terms in the tableau.

6 Implementation and Experiments

TEA is a Haskell-implementation of the termination analysis as described in this paper [Kic97, Pan97]. Optimizations and special rules for treating nested recursive functions are included. Abstract reduction is implemented by a template instantiation machine. The strategy for automatic analysis is as follows. First there is a dependency analysis that determines the sequence of analyzing the functions. Then there are two passes of the script in the determined sequence, where successively for every function there is an attempt to construct a closed termination tableaux. The obtained termination information is used eagerly in the first pass, where the strategy for the application of the approximation is approximate only nf-terminating expressions, such that no branching is necessary. The second pass avoids this eager use of nf-termination knowledge. The generated ordering constraints are then handed over to the ordering tableau-module that tries to construct and verify an ordering based on a linear polynomial interpretation for normal forms. If for a function the detection of nf-termination fails, but there is a preclosed tableau, then the criterion for lazy termination is checked. If a function is neither nf-terminating nor lazy terminating, but there is a preclosed termination tableau, then the program is able to give hints where a possible non-termination may occur by displaying the ordering constraints and the corresponding path.

The implementation incorporates strict built-in functions on integers in the core language and makes use of their termination behavior (see example 2).

To guarantee termination of tableau expansion, there is a maximal possible length of paths in the termination tableau, a practical value being around 40. A restriction for the ordering is that the polynomials are linear, the coefficients may be 0,1, or 2, and there is also a depth bound. Since the rules are non-deterministic, heuristics are used to optimize the probability of constructing a preclosed tableau.

We tested the implementation on three program scripts. The example `t10.core` is taken from the examples for strictness analysis in [Sch94]. It is generated from a script containing almost all the function of the Gofer [Jon94] prelude plus some other functions, the example `Rewrite.core` is one of the modules written for manipulating and reducing expressions for constructing an ordering tableau, and `Misc.core` is a set of standard functions on lists and trees, like different sorting algorithms, a combinator parser, functions generating an infinite lists of primes, and further functions (see [Pan97]). In the following figure the number of functions and the obtained results and the time on a HP C160 is recorded. The last column gives the number of functions where the termination information could not be detected.

Example	#	nf-term.	nf-term + lazy	preclosed	min.	not detected
`t10.core`	212	191 (90%)	200 (94.3%)	205 (96.7%)	0:49	14 (6.6%)
`Rewrite.core`	79	76 (96%)	76 (96%)	78 (98.7%)	7:38	2 (2.5%)
`Misc.core`	121	95 (78,5%)	101 (83,4%)	110 (90,9%)	3:04	13 (10.7%)

Fig. 3. Results on example files.

The results are promising: TEA detects almost all nf-terminating functions in `Rewrite.core`.
It is possible to improve the success rate by modifying the strategy. We analyzed the failure cases in order to learn about possible improvements. Using an interactive mode it is possible to improve the results slightly, which means that adding some passes with different strategies may help a bit. There are some (nf-terminating) functions in the prelude which are not detected as nf-terminating. Their termination behavior depends on the value returned by built-in functions. This could be remedied, but is omitted in our prototype implementation. An example for a function that cannot be proved nf-terminating by TEA is the function `tak`. However, `tak` is an artificial function and it is not easy to prove its termination.
Our prototype implementation and the results justify the hope that it is possible to include powerful tools into compilers which are able to detect a large amount of nf-termination and/or lazy termination in scripts of non-strict functional programs. This information is useful for a programmer as well as for program optimizations.

Acknowledgment

We would like to thank Marko Schütz for his helpful remarks in discussions.

References

[AG97] T. Arts and J. Giesl. Automatically proving termination where simplification orderings fail. In Michel Bidoit & Max Dauchet, editor, *Proceedings of the 7th International Joint Conference on Theory and Practice of Software Development (TAPSOFT '97)*, number 1214 in Lecture Notes in Computer Science, pages 261–272, Lille, France, 1997. Springer–Verlag.

[AH87] S. Abramsky and C. L. Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[AH96] P.H. Andersen and C.K. Holst. Termination analysis for offline partial evaluation of a higher order functional language. In *Static Analysis, third international symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 67–82. Springer-Verlag, 1996.

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

[Bur91] Geoffrey Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation*. Pitman, London, 1991.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 252–252. ACM Press, 1977.

[Der87] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1):69–116, 1987.

[Gie95a] J. Giesl. *Automatisierung von Terminierungsbeweisen für rekursiv definierte Algorithmen*. PhD thesis, Technische Hochschule Darmstadt, 1995.

[Gie95b] J. Giesl. Termination analysis for functional programs using term orderings. In Alan Mycroft, editor, *Static Analysis Symposium '95*, number 984 in Lecture Notes in Computer Science, pages 154–171. Springer, 1995.

[HAB+96] K. Hammond, L. Augustsson, B. Boutel, W. Burton, J. Fairbairn, J. Fasel, A. Gordon, M. Guzmán, J. Hughes, P. Hudak, T. Johnsson, M. Jones, D. Kieburtz, R. Nikhil, W. Partain, J. Peterson, S. Peyton Jones, and P Wadler. Report on the programming language Haskell 1.3. Technical report, Department of Computer Science, University of Glasgow, 1996.

[Jon94] Mark P. Jones. The implementation of the Gofer functional programming system. Research Report YALEU/DCS/RR-1030, Yale University, Department of Computer Science, May 1994.

[Kah95] S. Kahrs. Towards a domain theory for termination proofs. In *Proceedings of the 6th International Conference on Rewriting Techniques and Applications*, volume 914 of *Lecture Notes in Computer Science*, pages 241–255. Springer-Verlag, 1995.

[Kic97] Hubert Kick. Terminierungsanalyse für den Sprachkern einer nicht-strikten funktionalen Programmiersprache unter Verwendung eines Tableaukalküls für abstrakte Reduktion. Master's thesis, Johann Wolfgang Goethe-Universität, Frankfurt, 1997.

[Nöc93] Eric Nöcker. Strictness analysis using abstract reduction. In *Functional Programming Languages and Computer Architecture*, pages 255–265. ACM Press, 1993.

[Pan97] S. E. Panitz. *Generierung statischer Programminformation zur Kompilierung verzögert ausgewerteter funktionaler Programmiersprachen*. (Dissertation), Universität Frankfurt, 1997. preliminary version, in German.

[Sch94] Marko Schütz. Striktheits-Analyse mittels abstrakter Reduktion für den Sprachkern einer nicht-strikten funktionalen Programmiersprache. Master's thesis, Johann Wolfgang Goethe-Universität, Frankfurt, 1994.

[SNN94] Kirsten Lackner Solberg, Hanne Riis Nielson, and Flemming Nielson. Strictness and totality analysis. In Baudouin Le Charlier, editor, *Static Analysis*, number 864 in Lecture Notes in Computer Science, pages 408–422. Springer, 1994.

[SSPS95] M. Schmidt-Schauß, S.E. Panitz, and M. Schütz. Strictness analysis by abstract reduction using a tableau calculus. In Alan Mycroft, editor, *Static Analysis Symposium '95*, number 983 in Lecture Notes in Computer Science, pages 348–365. Springer, 1995.

[Ste95] J. Steinbach. Automatic termination proofs with transformation orderings. In *Proceedings of the 6th International Conference on Rewriting Techniques and Applications*, volume 914 of *Lecture Notes in Computer Science*, pages 11–25. Springer-Verlag, 1995.

[Wal91] Chr. Walther. *Automatisierung von Terminierungsbeweisen*. Künstliche Intelligenz. Vieweg, Braunschweig, 1991.

[Wal94] Chr. Walther. On proving the termination of algorithms by machine. *Artificial Intelligence*, 71:101–157, 1994.