# Proving Termination of Higher-Order Functional Programs with a Tableau-like Calculus

Sven Eric Panitz, Manfred Schmidt-Schauß
Fachbereich Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt
Germany
Tel.: +49 69 798 22890
e-mail: {panitz,schauss}@informatik.uni-frankfurt.de

**Abstract**

We apply the techniques from tableau calculi to the termination analysis of programs written in a lazy, higher-order, and polymorphically typed functional programming language like Haskell. The termination analysis uses a higher-order, lazy functional core language. Laziness prevents applicability of termination orderings.

We show that the combination of a tableaux that represents reduction possibilities with a method that computes termination orderings only for particular expressions is a successful method to find out whether a lazy functional program is terminating.

The method has been implemented in Haskell and is able to detect termination of all functions in one of its own modules.

## 1 Introduction

Termination is a fundamental property of programs. Though it is undecidable in general, it is of worth to have a method which can prove termination for a large number of programs. We propose a calculus for proving termination of programs which follows those ideas of tableaux as they have e.g. been developed for first order logics in [Smu71]. Tableau calculi expand trees which represent a systematic development of some logical formula that is to be proved.

A termination proof for a functional language can be given by showing that every recursive call of a function is applied to smaller arguments than the original call in some Noetherian ordering. Orderings on terms for termination proofs have been discussed in the field of term rewriting systems, see e.g. [Der87]. Automatic termination proofs which apply orderings on ground terms in a strict functional first order language are presented in [Wal94] and [Gie95b]. These methods cannot simply be applied to higher-order languages with non-strict semantics. Consider e.g. a function which calculates the length of a list. In a strict language this function will not terminate, if it is applied to a list expression which has non-terminating elements. With non-strict semantics the function `length` terminates for list, as long as they are finite, even if they have non-terminating subexpressions. Therefore, we have to investigate which parts of arguments of a function have to be evaluated, and which can be thrown away at some point.

We will apply the technique of abstract reduction which has been successfully used in the related question of strictness analysis in [Nöc93] and which has been formulated in terms of tableaux in [SSPS95].

Abstract reduction reduces expressions which contain abstract variables until a suitable recursive structure is found, which allows to reason on an inductive statement.
This paper is structured as follows: in section 2 we introduce the language that we will analyze, in section 3 we introduce our calculus with its two sub-calculi, in section 4 we give an example of how the two calculi work together. In the conclusion some practical results are given.

## 2 The language of discourse

We define $\Lambda_c$. It is a simple polymorphic typed higher-order functional core language. The language will be rather close to intermediate languages as they are used in compilers for non-strict functional languages as Haskell [HAB+96] or Clean [NSvP91]. Together with some front-end which desugares Haskell into $\Lambda_c$ we will be able to analyze Haskell programs.

### 2.1 The Type System

The type system for $\Lambda_c$ consists of the basic type *Num*, function types and a finite number of algebraic types. Algebraic types are the sum of finitely many type products. Thus there is a finite number of constructor constants for each algebraic type. These constructor constants are numbered.
Types may be polymorphic. Algebraic types may be recursive. An example of recursive algebraic type will be $(List\ \alpha)$.

### 2.2 Expressions and Super-combinator Definitions

$\Lambda_c$-expressions are built in the standard way. For every type $\tau$ there is a set $V^\tau$ of variables $x^\tau, y^\tau, \ldots$. Furthermore, there are typed constants $c_\tau$. Expressions are made up according to the following rules:

$$x^\sigma : \sigma \;\Big|\; c_\sigma : \sigma \;\Big|\; \frac{e : \sigma_1 \to \ldots \to \sigma_n \to \tau, e_i : \sigma_i}{(e\ e_1 \ldots e_n) : \tau}$$

where $e : \sigma$ denotes expression $e$ of type $\sigma$. Furthermore types may be specialized, i.e. $e : \sigma$ can be specialized to $e : \tau(\sigma)$ where $\tau$ replaces type variables by types.

Functions are defined through super-combinator definitions. Let $e : \tau$ be a $\Lambda_c$-expression that contains no other variables than $v_1^{\sigma_1}, \ldots, v_n^{\sigma_n}$. Then $sc\ v_1 \ldots v_n \;=\; e$ is a super-combinator definition. $sc$ is considered as a constant of $\Lambda_c$ of type: $\sigma_1 \to \ldots \to \sigma_n \to \tau$. The type of a super-combinator can be derived in some type inference algorithm, we assume that of Milner [Mil78].
The following constants are in $\Lambda_c$:

- Numerical constants and built-in functions such as $+, -, *, /, =, 1, 2, 3, \ldots$.
- For every algebraic type $A = A^p(\sigma_1, \ldots, \sigma_m)$ there exist finitely many constructor constants $\mathbf{con}_A^1, \ldots, \mathbf{con}_A^{n_i}$, such that the constructor $\mathbf{con}_A^j$ has a type of the form $\tau_{j_1} \to \ldots \tau_{j_k} \to A, j_k \geq 0$.
- For every algebraic type name $A$ there is one case constant $\mathbf{case}_A$ which takes $n+1$ arguments, where $n$ is the number of constructor constants for $A$. The first argument is the expression to be cased. The other arguments of $\mathbf{case}_A$ are functions taking $m_k$ arguments for $1 \leq k \leq n$, where $m_k$ is the arity of the k$^{th}$ constructor of $A$.

- super-combinators that may be defined in a recursive way.

An example of a recursive algebraic data type used in this paper is $(List\ \alpha)$ for the usual polymorphic lists with the two constructors $\texttt{Nil}_{(List\ \alpha)}$ and $\texttt{Cons}_{\alpha \to (List\ \alpha) \to (List\ \alpha)}$
In most cases type subscripts of variables and constants will be dropped.
Functions can be applied curried, i.e. in a way that they are not saturated with enough arguments. This is the only means to introduce functions as expressions.
As a notational convention $C[t]$ will denote an expression which has at some position the subexpression $t$.
The operational semantics consists of the usual $\delta$-reductions. The operational semantics is non-strict, i.e. the topmost-leftmost (normal-order-) redex is reduced. Reduction is performed in normal-order.
$\delta$-rules for built-in primitive functions are the usual ones. For every case constant of an algebraic type there is a $\delta$-rule of the following form:

$\mathbf{case}_{A_i}\ (\mathbf{con}_j\ t_1 \ldots t_m)\ e_1 \ldots e_n \to_\delta (e_j\ t_1 \ldots t_m)$

Super-combinators simply rewrite with their definition, i.e.
$(sc\ e_1 \ldots e_n) \to e[x_1 \mapsto e_1, \ldots x_n \mapsto e_n]$, for a super-combinator definition $(sc\ x_1 \ldots x_n) = e$.
As an example for $\Lambda_c$, consider a well-known `fold`-function on lists combined with some filtering on the list written in $\Lambda_c$:

```
foldrless f xs a = case xs a (foldrconsless f a)
foldrconsless f a x xs = f x (foldrless f (less x xs) a)

less x xs = case xs Nil (lesscons x)
lesscons y x xs
 = if (x<y) (Cons x (less y xs))(less y xs)
```

In order to have some notions close to the notion of termination, we distinguish expressions in a special form.

**Definition 1. (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 one of the following forms:

- $(\mathbf{con}_A\ e_1 \ldots e_n)$.
- $(\mathbf{sc}\ e_1 \ldots e_n)$ and $arity(\mathbf{sc}) > n$.
- it is a built-in constant $n \in I\!N$.

For an expression of a functional type it is not very interesting whether it has a normal-form or not. We are interested rather in the way a function works for expressions in normal-form. Therefore, we introduce our basic notion of termination. It is similar to the notion of normal-forms but differs for functions:

**Definition 2. (nf-termination)**
$e \in \Lambda_c$ nf-terminates iff:

- it does not have a function-type and can be reduced to an expression in normal form, or
- it is of type $e : \sigma_1 \to \sigma_2$ and can be reduced to some HNF, such that for each nf-terminating $e_1 : \sigma_1$ with $(e\ e_1) \in \Lambda_c$, $(e\ e_1)$ nf-terminates.

$\mathcal{NF} = \{e \in \Lambda_c | e \text{ nf-terminates}\}$

This definition excludes potential infinite objects.

## 3 Termination Analysis

Now we will develop a calculus for proving nf-termination of sets of expressions. The calculus is set up with two sub-calculi: termination tableaux and ordering tableaux.

### 3.1 Termination Tableaux

We want to analyze sets of expressions. These will be represented by abstract expressions:

**Definition 3. (abstraction)**
$\Lambda_c^{\#}$, the language of abstract expressions, is generated by the same rules as $\Lambda_c$ with the additional set of constants $\mathcal{T} = \{\mathbf{T}^{x_1}, \mathbf{T}^{x_2}, \ldots\}$. The constants $\mathbf{T}^{x_i}$ are called abstract variables. They have the most general type of simply a type variable.
$e^{\#} \in \Lambda_c^{\#}$ is called an abstract normal-form if $e^{\#}$ consists entirely of constructor constants and abstract variables.

A $\Lambda_c^{\#}$-expression $e$ with the abstract variables $\{\mathbf{T}^{x_1}, \ldots, \mathbf{T}^{x_n}\}$ can in a way be considered as a formula of the form:

$$\forall \mathbf{T}^{x_1} \ldots \forall \mathbf{T}^{x_n} : (\mathrm{Nf}(\mathbf{T}^{x_1}) \wedge \ldots \mathrm{Nf}(\mathbf{T}^{x_n}) \wedge \mathrm{Ex}(e)) \rightarrow \mathrm{Nf}(e),$$

where Nf is a predicate, which denotes nf-terminating expressions and Ex is a predicate, which denotes well-typed expressions.
As a semantics for $\Lambda_c^{\#}$-expressions we will interpret them as subsets of the universe $\Lambda_c$.

**Definition 4. (concretisation)**
A concretisation $\gamma$ of $e^{\#} \in \Lambda_c^{\#}$ is a substitution $\gamma : \mathcal{T} \rightarrow \Lambda_c$, such that:
$\gamma(\mathbf{T}^x)$ nf-terminates for all abstract variables $\mathbf{T}^x$.
A concretisation is extended in the usual way to a term homomorphism from $\Lambda_c^{\#}$ to $\Lambda_c$.
$\Gamma(e^{\#}) = \{\gamma | \gamma \text{ is concretisation of } e^{\#}\}$
We lift the notion of nf-termination to $\Lambda_c^{\#}$:
$e^{\#} \in \Lambda_c^{\#}$ nf-terminates iff.: for all $\gamma \in \Gamma(e^{\#}) : \gamma(e^{\#})$ nf-terminates.

Concretisations can be compared to models in classical logics. A concretisation $\gamma$ of an abstract expression $e^{\#}$ is valid, if $\gamma(e^{\#})$ nf-terminates. An abstract expression is valid, if all of its concretisations are valid, or in other words, it is nf-terminating.
We now want to prove that some $\Lambda_c^{\#}$-expressions are valid. As in classical tableaux we will prove this by assuming the negated statement and show that this is contradictory by expanding every possible case of the negated statement in a tableau structure. The negation of an abstract expression is a proposition of the form:

$$\exists \mathbf{T}^{x_1} \ldots \exists \mathbf{T}^{x_n} : (\mathrm{Nf}(\mathbf{T}^{x_1}) \wedge \ldots \mathrm{Nf}(\mathbf{T}^{x_n}) \wedge \mathrm{Ex}(e)) \wedge \neg \mathrm{Nf}(e),$$

i.e. we are looking for a concretisation $\gamma$, such that $\gamma(e)$ does not nf-terminate. Termination tableaux use two sorts of rules: expansion rules which systematically try to generate a model for the negated statement, and closing rules, which mark a contradiction at some leaves to the assumption that there is a model which satisfies the negation.
We introduce rules which can expand a tableau. These expansion rules have to be sound in a way that we will not lose a concretisation for which the expanded node is not nf-terminating; i.e. an expansion rule which expands a node $n$ with the nodes $n_1, \ldots, n_k$ is sound iff.: if there is a concretisation $\gamma$ such that $\gamma(n)$ does not nf-terminate then there is a node $n_i$ and there exists a concretisation $\gamma'$ such that $\gamma'(e_i)$ does not nf-terminate.
An overview of the expansion rules is given in figure 1. We discuss them in detail:

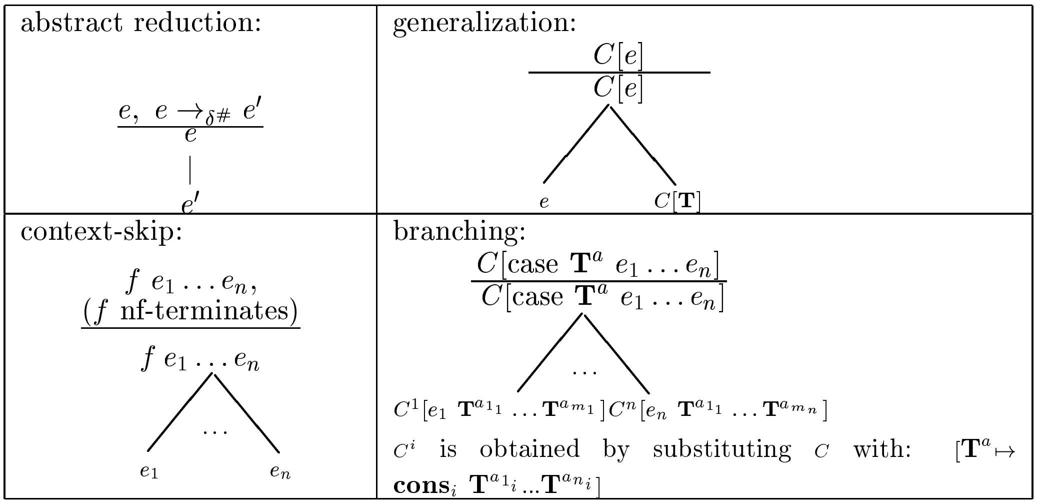


Figure 1: Expansion rules for termination-tableaux.

- Abstract reduction comprises concrete reductions performed on abstract expressions together with some special abstract reduction rules such as: reduction of built-in functions on abstract variables (e.g.: $\mathbf{T}^a + \mathbf{T}^b \rightarrow_{\delta\#} \mathbf{T}^c$ for a new abstract variable $\mathbf{T}^c$) and application of abstract variables (e.g. $(\mathbf{T}^a\ \mathbf{T}^b) \rightarrow_{\delta\#} \mathbf{T}^c$ for a new abstract variable $\mathbf{T}^c$). For an abstract reduction step $e \rightarrow_{\delta\#} e'$ : for all $\gamma \in \Gamma(e)$ there exists a $\gamma' \in \Gamma(e')$ with $\gamma(e) \rightarrow_\delta \gamma'(e')$.

- We can substitute any subexpression with a new abstract variable, if we analyze the subexpression on its own. This is an generalization, because the subexpression may not be nf-terminating but the original expression was nf-terminating. Therefore, this rule has to be applied carefully.

- Applying the rule of context-skipping reduces a top-level application of constants to all the applications arguments. This is only allowed when the top-level function is known to be nf-terminating. Applying this rule will assume that the top-level function is strict in its arguments. Instead of expanding a function application we analyze the arguments for nf-termination. For a non-strict function there may be non nf-terminating arguments, but the application is nf-terminating. This rule has to be applied to constructor constants whenever possible.

- **case**-expressions on abstract variables get reduced to all possible alternatives of the **case**-construct. The new edges in the tableau will be marked by substitutions, in which the abstract variable is substituted by the form it has to have for the alternative. E.g. a leaf node (**case** $\mathbf{T}^x s\ e_1\ e_2$) can be expanded with the two new nodes (**case** `Nil` $e_1\ e_2$) and (**case** (`Cons` $\mathbf{T}^y\ \mathbf{T}^{ys}$) $e_1\ e_2$) and the new edges are marked with the substitutions $[\mathbf{T}^{xs} \mapsto$ `Nil`$]$ and $[\mathbf{T}^{xs} \mapsto$ (`Cons` $\mathbf{T}^y\ \mathbf{T}^{ys}$)$]$. These substitutions are called split-substitutions.

All of these expansion rules are sound.
Now we can detect leaves which do not have a concretisation that is not nf-terminating. Such leaves are nodes, which are marked with an abstract variable:

**Definition 5. (tautology node)**
A node which is marked with an abstract variable is called a tautology node.

A tautology node denotes essentially the following formula:

$$\exists \mathbf{T}^x : (\mathrm{Nf}(\mathbf{T}^x) \wedge \mathrm{Ex}(\mathbf{T}^x)) \wedge \neg \mathrm{Nf}(\mathbf{T}^x),$$

This is not satisfiable! Note that every leaf in abstract normal-form can be extended to a subtableau, which entails only tautology leaves.
We are now in the position to be able to identify nodes in a tableau which describe loops.

**Definition 6. (recursive node)**
A node $e'$ in a tableau is called *recursive node* iff there is a node $e$ above $e'$ with $\sigma(e) = e'$ for some $\sigma : \mathcal{T} \to \Lambda_c^{\#}$, such that $\sigma(\mathbf{T}^{x_i})$ nf-terminates for all abstract variables $\mathbf{T}^{x_i}$. $(e, e')$ is called a recursive pair; the path from $e$ to $e'$ is a recursive path. $\sigma$ is called recursion substitution.

With this definition it is undecidable if a leaf is a recursive node, because this involves knowledge about nf-termination of the $\sigma(\mathbf{T}^{x_i})$. A simple syntactic condition is that $\sigma(\mathbf{T}^{x_i})$ is an abstract normal-form (Def. 3).
The potential loops of a termination tableau have to be proved to be harmless. This is done by showing that the arguments are decreased in some Noetherian ordering.

**Definition 7. (ordering constraint)**
For a recursive pair $(e, e')$ with the recursion substitution
$\sigma = [\mathbf{T}^{x_1} \mapsto e_1, \dots, \mathbf{T}^{x_n} \mapsto e_n]$ the ordering constraint is defined as
$(\delta(\mathbf{T}^{x_1}), \dots, \delta(\mathbf{T}^{x_1})) > (e_1, \dots e_n)$, where $\delta$ is the composition of all split-substitutions on the recursive path.
The meaning of an ordering constraint $(e_1, \dots, e_n) > (e_{n+1}, \dots, e_{2n})$ is: there is a Noetherian ordering $>$ on $n$-tuples of $\Lambda_c$-expressions, such that for any $\gamma \in \bigcap_{1 \leq i \leq 2n} \Gamma(e_i)$: $(\gamma(e_1), \dots, \gamma(e_n)) > (\gamma(e_{n+1}), \dots, \gamma(e_{2n}))$.

**Definition 8. (preclosed)**
A tableau, which only has leaves that are tautologies or that are recursive nodes, is called preclosed.

We can give a simple condition when a tableau constitutes a termination proof.

**Theorem 9. (soundness)**
*Let $\mathcal{T}$ be a preclosed tableau, such that every pair of recursive paths which share some nodes, has the same start node. The root of $\mathcal{T}$ nf-terminates if every ordering-constraint can be fulfilled by some Noetherian ordering, and the ordering constraints of two recursive pairs $(e, e_1)$, $(e, e_2)$ can be fulfilled by the same ordering.*

*Proof.* (Sketch): Assume that there is a concretisation of the root which does not nf-terminate. Then there is at least one leaf, which does not nf-terminate. The leaf has to be a recursive node. Choose a recursive node with the following properties:

- it does not nf-terminate.
- there is no recursive node that does not nf-terminate, which has a recursive path that starts below the start node of the chosen recursive path.

Now choose the smallest concretisation with respect of the ordering which fulfills the ordering constraint of the start node of this recursive path. Because of the soundness of our expansion rules we can follow this concretisation along a path down to one of the recursive nodes and get a concretisation which is smaller in the ordering which closes the tableau. This contradicts the assumption that we have chosen the smallest concretisation.

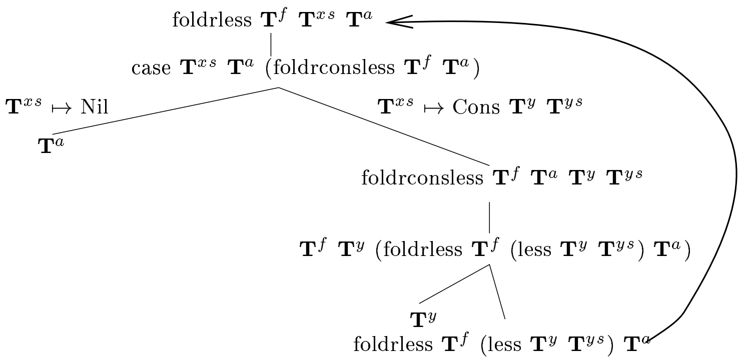


Figure 2: A preclosed tableau for `foldr`.
Recursive pairs are marked with an arrow. The ordering constraint is:
(Cons $\mathbf{T}^y$ $\mathbf{T}^{ys}$) $>$ (less $\mathbf{T}^y$ $\mathbf{T}^{ys}$).

As can be seen, a preclosed tableau may have independent parts for which different orderings can be used in the termination proof. An example for such a situation will be given in section 4
In the case of recursive paths, which have nodes in common but have different start nodes the theorem is not generally true. A counterexample is given in [Pan96].

*Example 1.* As a simple example let us try to prove nf-termination of the function `foldrless` defined in section 2. A preclosed tableau is given in figure 2.
In order to get a proof for nf-termination of `foldrless`, we have to show that the ordering constraint of the recursive path can be fulfilled by some Noetherian ordering. The ordering constraint is: (Cons $\mathbf{T}^y$ $\mathbf{T}^{ys}$) $>$ (less $\mathbf{T}^y$ $\mathbf{T}^{ys}$). This will be proved by using an ordering tableau as introduced in the next section.

## 3.2 Ordering Tableaux

As we have seen, preclosed termination tableaux generate ordering constraints which have to be fulfilled in order to constitute a proof of nf-termination. In this section we will introduce a tableau calculus which generates Noetherian orderings which meet ordering constraints.
To define orderings on $\Lambda_c$ we will interpret $\Lambda_c$-expressions as natural numbers, i.e. we define an interpretation function $\tau : \Lambda_c \to \mathbb{N} \cup \{\bot\}$ and prove for an ordering constraint $l > r$ that for any $\gamma \in \Gamma(l) \cap \Gamma(r)$ if $\tau(l) \neq \bot$ and $\tau(r) \neq \bot$ then $\tau(l) > \tau(r)$. The interpretation $\tau$ will be called termination function. It will be restricted to functions which interpret applications of constructors as linear functions. We will ensure that for a nf-terminating expression $e$ the value of the termination function $\tau(e)$ will be defined, i.e. $\tau(e) \neq \bot$.

**Definition 10. (termination function)**
Termination functions $\tau^\beta : \Lambda_c \to \mathbb{N} \cup \{\bot\}$ are defined through defining equations in the following way:

- $\tau^\beta(e) = 1 + \beta(n_1^c)\tau^\beta(e_1) + \ldots + \beta(n_m^c)\tau^\beta(e_m)$ if $e$ has the HNF $(c\ e_1 \ldots e_m)$

where $\mathcal{C} = \{n_i^c | i \in \{1, \ldots, \text{arity}(c)\}, c \text{ is a constructor}\}$ is a set of variables for coefficients and $\beta$ is a map $\beta : \mathcal{C} \to \mathbb{N}$
This means that a termination function $\tau^\beta$ is completely determined by the function $\beta$.

We assume that there is a defining equation for each constructor $c$. If e does not have a HNF, then $\tau(e) = \perp$. If $\tau(e)$ would yield an infinite number we define $\tau(e) = \perp$. The multiplication is bottom avoiding, i.e. $0 * \tau(e) = 0$ even if $\tau(e) = \perp$. This is unproblematic, because we will never get subexpressions of the form $\tau(e) * 0$ with $\tau(e) = \perp$.

We want to find Noetherian orderings which meet the ordering constraints as generated by termination tableaux. Ordering constraints are defined through tuples of abstract expressions. Fortunately, these abstract expressions are known to be nf-terminating. Therefore, we know that no parts of any termination function will be evaluated to $\perp$.

**Definition 11. (ordering proposition)**
An ordering proposition is of the following form:

$$(n_0^l, \{(n_1^l, e_1^l), \ldots, (n_j^l, e_j^l)\}) \geq (n_0^r, \{(n_1^r, e_1^r), \ldots, (n_k^r, e_k^r)\}),$$

where the $e_i^{lr} \in \Lambda_c^{\#}$ and the $n_i^{lr}$ are arithmetic expressions over $\{+, *\} \cup \mathcal{C} \cup \mathbb{N}$.
An ordering proposition is valid, if there is a map $\beta : \mathcal{C} \to \mathbb{N}$ such that for all concretisations $\gamma$:

$$\beta(n_0^l) + \sum_{1 \leq i \leq j} \beta(n_i^l)\gamma^\beta(e_i^l) \geq \beta(n_0^r) + \sum_{1 \leq i \leq k} \beta(n_i^r)\gamma^\beta(e_i^r),$$

if both sides get evaluated to a value $\neq \perp$.

We now expand ordering tableaux. Ordering tableaux have ordering propositions as nodes. These ordering propositions are simplified in a way, that

- no abstract expression in HNF occurs, and
- no two pairs $(n, e), (m, e)$ occur on one side of the proposition.

We have a set of expansions rules as given in figure 3. We discuss them in detail:

- abstract reduction is the same rule as for termination tableaux. We also proceed in the same way with applications of case constants to abstract variables as we have done in termination tableaux.
- ordering propositions can be syntactically approximated by simply deleting some parts on the left hand side or adding some parts to the right hand side.
- besides the simple syntactic approximation described above, we can make use of the transitivity of orderings and approximate the left hand side with a smaller and the right hand side with a larger value. This rule will not be discussed any further in this paper, but turns out to be the key for a special kind of function where a heuristic will generate lemma propositions which allow the application of this rule. More details can be found in [PSS96].

We can now check for every node, if it is a tautology for some mapping $\beta$ or some sort of recursive node again.

**Definition 12. (tautology node)**
An ordering proposition which for some mapping $\beta : \mathcal{C} \to \mathbb{N}$ can be transformed by simple normalization calculations to a node $(n_l, \{\}) \geq (n_r, \{\})$ with $n_l \geq n_r$ is called a tautology.

**Definition 13. (recursive node)**
A path from a node $n$ to a node $n'$ is called a recursive path if:

| | |
|---|---|
| reduction left:<br>$\dfrac{(n_l,\{\dots,(n_e,e),\dots\})\geq rhs,\ e\to e'}{(n_l,\{\dots,(n_e,e),\dots\})\geq rhs}$<br>$\mid$<br>$(n_l,\{\dots,(n_e,e'),\dots\})\geq rhs$ | reduction right:<br>$\dfrac{lhs\geq(n_r,\{\dots,(n_e,e),\dots\}),\ e\to e'}{lhs\geq(n_r,\{\dots,(n_e,e),\dots\})}$<br>$\mid$<br>$lhs\geq(n_r,\{\dots,(n_e,e'),\dots\})$ |
| in the same way branching reductions on case-expressions | |
| delete pair left:<br>$\dfrac{(n_l,\{\dots,(n_e,e),\dots\})\geq rhs,\ e \text{ nf-terminates}}{(n_l,\{\dots,(n_e,e),\dots\})\geq rhs}$<br>$\mid$<br>$(n_l+n_e),\{\dots,\dots\})\geq rhs$ | add pair right:<br>$\dfrac{lhs\geq(n_r,\{lps\}),\ n_e'\geq n_e,\ e \text{ nf-terminates}}{lhs\geq(n_r,\{lps\})}$<br>$\mid$<br>$lhs\geq(n_r-n_e,\{(n_e,e),lps\})$ |
| approximation left:<br>$\dfrac{lhs\geq mhs, mhs\geq rhs}{mhs\geq rhs}$<br>$\mid$<br>$lhs\geq rhs$ | approximation right<br>$\dfrac{lhs\geq mhs, mhs\geq rhs}{lhs\geq mhs}$<br>$\mid$<br>$lhs\geq rhs$ |

Figure 3: Expansion rules for ordering-tableaux.

- there is a mapping $\beta$ such that there is a substitution $\sigma : \mathcal{T} \to \Lambda_c^{\#}$ with $\sigma(\beta(n)) = \beta(n')$.
- there is at least one reduction of a normal-order redex on the path.

$n$ is called a recursive node.

Thus we are looking for a $\beta$ such that we get a termination function $\tau^{\beta}$ which produces a recursive node. Our implementation right now uses a simple brute force technique to find such a mapping $\beta$. We are simply testing every mapping $\beta : \mathcal{C} \to \{0,1\}$. There are $\mathbf{m}^2$ such $\beta$ where $\mathbf{m} = \sum_{(\mathbf{c} \text{ is constructor})} arity(\mathbf{c})$ is the sum of all constructor arities. These termination functions turn out to be sufficient for most examples.

**Theorem 14.** *A tableau with a global mapping $\beta$ for which every leaf is a tautology or a recursive node proves the ordering proposition at the root to be valid.*

*Proof.* (Sketch): Assume the theorem to be false: then there is a concretisation $\gamma$ of the root, such that for $\gamma$ and $\beta$ the right hand side of the root gets evaluated to a greater integer than the left hand side. In order to evaluate the ordering constraint for $\gamma$ we have to reduce a finite number of normal-order redexes. We ensured that on every recursive path a normal-order redex which is necessary for evaluating the ordering proposition has been reduced. Thus we have a measure which is decreased by recursive paths. (For a definition of necessary reductions see [SSPS95]) If for recursive pairs $(e, e')$ every concretisation of $\gamma(e')$ of $e'$ were a concretisation of $e$, then we were done; but, unfortunately, this is not true. The only thing that can happen is that $\gamma(e')$ at some subexpression has not a nf-terminating expression, where a concretisation of $e$ is supposed to have a nf-terminating expression. Fortunately, this can only cause the ordering constraint to be undefined, which for our purposes is sufficient.

*Example 2.* As a simple example, consider the ordering proposition, which has to be proved in order to show termination of `foldrless`. A closed ordering tableau is given in figure 4. Note that this is the same ordering proposition, which has to be proved for a termination proof of the usual `quicksort` function. In order to find

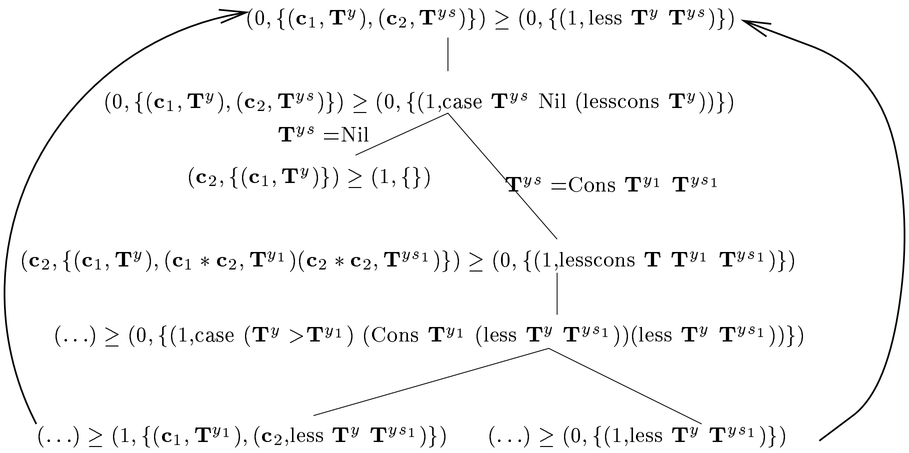


Figure 4: A closed ordering tableau for `less`. Recursive pairs are marked with an arrow.

an ordering which fulfills (`Cons` $\mathbf{T}^x$ $\mathbf{T}^{xs}$)>(`less` $\mathbf{T}^x$ $\mathbf{T}^{xs}$) we create the root node: $(0,\{(1,(\texttt{Cons}\ \mathbf{T}^x\ \mathbf{T}^{xs}))\}) \geq (1,\{(1,(\texttt{less}\ \ \mathbf{T}^x\ \mathbf{T}^{xs}))\})$
For the root we get the two recursive nodes:

- $(\mathbf{c}_2,\{(\mathbf{c}_1,\mathbf{T}^y),(\mathbf{c}_1 * \mathbf{c}_2,\mathbf{T}^{y_1}),(\mathbf{c}_2 * \mathbf{c}_2,\mathbf{T}^{ys_1})\})$
  $\geq (1,\{(\mathbf{c}_1,\mathbf{T}^{y_1}),(\mathbf{c}_2,\text{less } \mathbf{T}^y\ \mathbf{T}^{ys_1})\})$
- $(\mathbf{c}_2,\{(\mathbf{c}_1,\mathbf{T}^y),(\mathbf{c}_1 * \mathbf{c}_2,\mathbf{T}^{y_1})(\mathbf{c}_2 * \mathbf{c}_2,\mathbf{T}^{ys_1})\}) \geq (0,\{(1,\text{less } \mathbf{T}^y\ \mathbf{T}^{ys_1})\})$

The mapping for which the tableau is preclosed is:
$\beta(\mathbf{c}_1) = 0$ and $\beta(\mathbf{c}_2) = 1$.
This mapping makes the second recursive node to:
$(1,\{(1 * \mathbf{c}_2,\mathbf{T}^{ys_1})\}) \geq (0,\{(1,\text{less } \mathbf{T}^y\ \mathbf{T}^{ys_1})\})$
This is only a recursive node, if it is approximated to:
$(1,\{(1 * \mathbf{c}_2,\mathbf{T}^{ys_1})\}) \geq (1,\{(1,\text{less } \mathbf{T}^y\ \mathbf{T}^{ys_1})\})$
Which can be normalized to:
$(0,\{(1 * \mathbf{c}_2,\mathbf{T}^{ys_1})\}) \geq (0,\{(1,\text{less } \mathbf{T}^y\ \mathbf{T}^{ys_1})\})$

# 4 Interaction between Termination and Ordering Tableaux

As a final example we will show that the whole mechanism of termination tableaux is not only necessary for dealing with higher-order functions and non-strict evaluation, but can also be useful for simple first-order examples. We will show, how abstract reduction in termination tableau can help to find proofs with less sophisticated orderings in ordering tableaux. We want to analyze some sort of division function on Peano-numbers, which is taken from [Gie95a].

```
data Num = N|S Num

q:: Num -> Num -> Num -> Num
q x y z = case z N (q2 x y)
q2 x y z1 = case y (S (q x (S z1)(S z1))) (q3 x z1)
q3 x z1 y1 = case x N (q4 y1 z1)
q4 y1 z1 x1 = q x1 y1 (S z1)
```

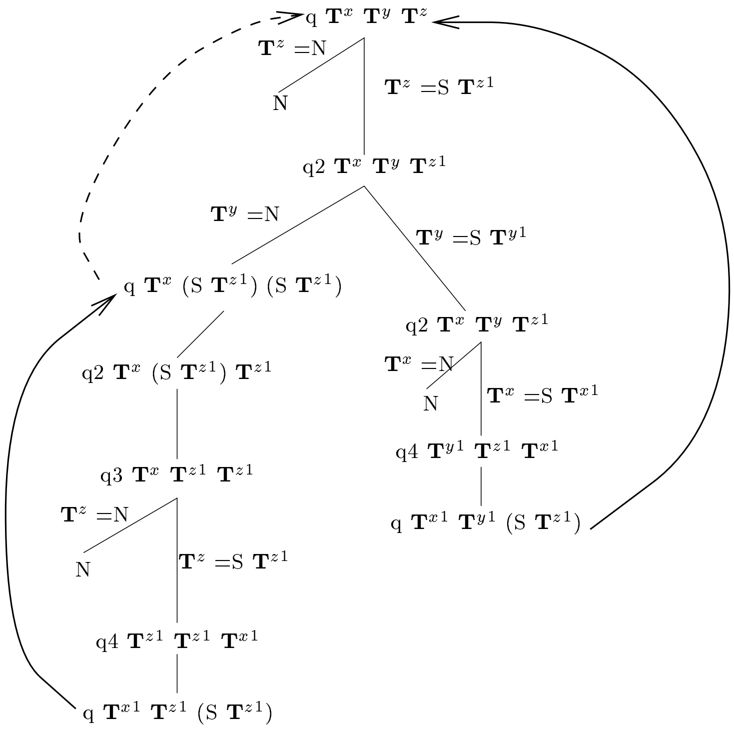


Figure 5: A termination tableau for `q`.
The first recursive path on the left is thrown away in order to find a recursive path further down in the tableau.

Now we expand a termination tableau for `q` as can be seen in figure 5.
If we take a look at the smallest preclosed tableau for (`q` $\mathbf{T}^x$ $\mathbf{T}^y$ $\mathbf{T}^z$), then this generates the following two ordering constraints, which have to be fulfilled with one ordering:

$$(\mathbf{T}^x, \mathtt{N}, \mathrm{S}\ \mathbf{T}^{z1}) > (\mathbf{T}^x, \mathrm{S}\ \mathbf{T}^{z1}, \mathrm{S}\ \mathbf{T}^{z1})$$

$$(\mathrm{S}\ \mathbf{T}^{x1}, \mathrm{S}\ \mathbf{T}^{z1}, \mathrm{S}\ \mathbf{T}^{z1}) > (\mathbf{T}^{x1}, \mathbf{T}^{z1}, \mathbf{T}^{z1})$$

There is no simple interpretation of the constructors as linear polynomials, which fulfills this ordering constraints. The ordering, which is given in [Gie95a] is based on the interpretation $\tau$:

$$\begin{aligned}
\tau((x,y,z)) &= \tau(x) + (\tau(x) - \tau(y) + \tau(z))^2 \\
\tau(\mathtt{N}) &= 1 \\
\tau(\mathtt{S}\ x) &= 1 + x
\end{aligned}$$

Ordering tableaux cannot generate such orderings. However we can ask termination tableaux to find further recursive paths, such that we break up a cluster of recursive paths which share tableau nodes into independent recursive paths. This is what we have done in figure 5. Now we have two independent recursive paths, which generate ordering constraints that can even be fulfilled by different orderings. On the left hand side we get the ordering constraint:

$$(\mathrm{S}\ \mathbf{T}^{x1}, \mathrm{S}\ \mathbf{T}^{z1}, \mathrm{S}\ \mathbf{T}^{z1}) > (\mathbf{T}^{x1}, \mathbf{T}^{z1}, \mathrm{S}\ \mathbf{T}^{z1}),$$

which can e.g. be fulfilled with:

$$\tau((x,y,z)) = \tau(x) + \tau(y) + \tau(z).$$

On the right hand side we get the ordering constraint:

$$(\mathrm{S}\ \mathbf{T}^{x1}, \mathrm{S}\ \mathbf{T}^{z1}, \mathrm{S}\ \mathbf{T}^{z1}) > (\mathbf{T}^{x1}, \mathbf{T}^{z1}, \mathbf{T}^{z1}),$$

which can be fulfilled by the same simple ordering. As can be seen, it is not necessarily so that this example requires an interpretation of a quadratic polynomial with a negative coefficient.

## 5 Conclusion

This paper presents our calculus for proving termination with its basic rules. The two tableau calculi can be extended with further rules that enable us to prove termination of nested recursion. The calculus can be used to prove quite a number of different notions of termination, by programming these into nf-terminating expressions, e.g. we can show that an abstract expression has only concretisations which represent finite lists, by applying the function `length` to it. With small modifications the calculus can also be used to show termination of the strict evaluation of expressions [Pan96].
Our method is implemented in Haskell. We made several tests with our implementation. These tests include analyses of program modules which were not specially created for the test, but simply taken as real-world examples. The code has been desugared from Haskell into $\Lambda_c$ semi-automatically by using the Gofer [Jon94] compiler and applying some further transformation into our syntax by hand.
One of these tests analyzes the abstract reduction module of the analyzer itself. The implementation detects all nf-terminating functions of the 90 functions in this module. This includes some non-trivial functions. As a larger test we analyzed a file consisting of 229 functions, including almost the complete Gofer prelude. This file was analyzed in a bit over 2 minutes on our HPPA 735. The implementation in its current state can easily prove 191 functions to be nf-terminating. Of the remaining 38 functions 16 are not nf-terminating. 13 of these are lazy-terminating, i.e. produce possibly infinite output but never run into a loop which does not produce any output. With slight extensions these functions can be proven to be lazy-terminating. Of the remaining 22 functions nf-termination of 12 functions was not detected because the implementation does not apply all rules for nested recursion (These rules are also not presented in this paper but can be found in [PSS96]). The remaining nf-terminations were not found mostly due to functions which depend on calculations on built-ins which were not modeled as Peano-numbers.
We hope to extend our implementation to a more powerful tool which is able to detect almost all nf-termination in this test file and arbitrary programs.

## Acknowledgment

We would like to thank Hubert Kick for his work on the implementation of termination tableaux and Marko Schütz for his helpful remarks in discussions.

## References

[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] Jürgen 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.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *J.Comp.Sys.Sci*, 17:348–375, 1978.

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

[NSvP91] E. Nöcker, J. E. Smetsers, M. van Eekelen, and M. J. Plasmeijer. Concurrent Clean. In Springer Verlag, editor, *Proc of Parallel Architecture and Languages Europe (PARLE'91)*, number 505 in Lecture Notes in Computer Science, pages 202–219, 1991.

[Pan96] Sven Eric Panitz. Termination poofs for a lazy functional language by abstract reduction. Technical Report Frank 06, J. W. Goethe-Universität, FB 20, Professur KIST,
`http://ki.informatik.uni-frankfurt.de/papers/articles.html`, 5 1996.

[PSS96] Sven Eric Panitz and Manfred Schmidt-Schauß. Tea: A system for proving termination of a non-strict higher-order functional language.
`http://ki.informatik.uni-frankfurt.de/tea/tea.html`, 10 1996.

[Smu71] Raymond M. Smullyan. *First-Order Logic.* Springer, 1971.

[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.

[Wal94] Chr. Walther. Mathematical induction. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 127–228. Oxford university press, 1994.