Termination Proofs for a Lazy Functional Language by Abstract Reduction *(draft)*

Sven Eric Panitz
Fachbereich Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt
Germany
e-mail: panitz@informatik.uni-frankfurt.de

June 3, 1996

Contents

I will get Peter Quince to write a ballad of this dream; it shall be called Bottom's Dream, because it hath no bottom;

A Midsummer Nights Dream

Abstract

Automatic termination proofs of functional programming languages are an often challenged problem.

Most work in this area is done on strict languages. Orderings for arguments of recursive calls are generated. In lazily evaluated languages arguments for functions are not necessarily evaluated to a normal-form. It is not a trivial task to define orderings on expressions that are not in normal-form or that do not even have a normal-form.

We propose a method based on an abstract reduction process that reduces up to the point when sufficient ordering relations can be found. The proposed method is able to find termination proofs for lazily evaluated programs that involve non-terminating subexpressions.

Analysis is performed on a higher-order polymorphic typed language and termination of higher-order functions can be proved, too.

The calculus can be used to derive information on a wide range on different notions of termination.

Chapter 1

Introduction

Since the early days of computer science the termination behavior of programs has been of keen interest in research. Termination and dually non-termination as well are elementary features of a computer program. Every programmer designs programs in a way that they will produce a result in finite time. Therefore, termination of the main procedure is of ultimate importance. On the other hand in non-strict languages where a function application does not necessarily demand the evaluation of its arguments, information on non-termination can be of interest for optimizing program transformations which are performed by a compiler. Such non-termination information of the form: a function does not terminate if it is applied to a non-terminating argument is called strictness information. This paper treats automatic termination proofs for a non strict functional language, i.e. a programming language, where there may be non-terminating subexpressions.

To analyze the termination behavior of a program it is necessary to give a proof for all possible values as program arguments and variables; or, to get more detailed information, for a subclass of all possible input values. A good means for such an analysis is by way of abstract interpretation. A set of values becomes represented by some abstract value. The initial formal framework for abstract interpretation is presented in [CC77]. Abstract interpretation is extensively used in strictness analysis [Myc80, BHA85, Bur87].

To analyze the termination behavior of a program it is furthermore necessary to detect potential loops in the program execution. Such potential loops are to be found as recursive calls in a functional language or as repeatedly running through the same sequence of commands in an imperative language. A good means for detecting such potential loops is to make a symbolic execution of a program on the abstracted values. This method of abstract reduction has been applied in strictness analysis [Nöc93, SSPS95]. An early example for executing programs on abstract values is presented in [Bur74].

Finally, to prove termination of programs, one has to show that all potential loops

are harmless, i.e. they will only be executed finitely many times in a concrete program execution. The usual means for this test is to find some Noetherian orderings on the argument and variable values. It has to be shown that when a previous state is reached again, then the values of arguments and variables will have decreased in some Noetherian ordering. Such orderings for termination proofs have extensively been studied in the area of term rewriting systems, see e.g. [Der87].

Automatic termination provers for functional languages often stay in the tradition of term rewriting and try to generate sufficient ordering relations [Wal94, Gie95a, Gie95b]. Such methods can only give termination proofs for applications to arguments that have a finite normal-form, i.e. they fail compppletely to show whether a function can terminate for undefined or infinite arguments. They are furthermore limited to first-order languages and generally cannot handle mutual recursion.

This paper is structured as follows: In section 2.1 the language we want to analyze is defined. In section 2.2 abstract values are introduced. In section 2.3 abstract reduction steps are defined in terms of deduction rules on termination tableaux and are proved to be sound. Section 2.4 treats application of non-terminating arguments for a lazily evaluated language. In section 3.2 a second calculus is presented, which enables us to prove ordering propositions on the result of recursive functions such that termination proofs for functions which have calls to other functions in their recursive call can be made. A collection of example proofs is to be found in the appendix.

Chapter 2

Termination Tableaux

2.1 The Language of Discourse

We define Λ_c, a simple polymorphic typed higher order functional core language. It resembles closely the language used in [PJL91] with the exception that no local function definitions can be made and *case* alternatives have to be complete. This does not restrict the method we present, because every functional language can be translated into Λ_c by lambda-lifting it.

2.1.1 The Type System

The type system for Λ_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. So there is a finite number of constructor constants for each algebraic type. These are numbered.

Types may be polymorphic. Algebraic types may be recursive. Examples of recursive algebraic types will be $(List\ \alpha)$ and $Tree$.

2.1.2 Expressions and Super-combinator Definitions

Λ_c-expressions are build in the standard way. For every type τ there is a set V^τ of variables $x^\tau, y^\tau, \ldots$. Furthermore there are typed constants c_τ. Expressions are build by 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 to be of type σ. Furthermore types may be specialized, i.e. $e : \sigma$ can be specialized to $e : \tau(\sigma)$ where τ substitutes type variables with types.

Functions are defined through super-combinator definitions. Let $e : \tau$ be a Λ_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 constant of Λ_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 Λ_c:

- Numerical constants .
- Built-in functions like $+, -, *, /, =$.
- 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 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 kth constructor of A.
- super-combinators that may be defined in a recursive way.

An example of an algebraic type is *Bool*. Its constructor constants are **true** and **false**. **if** is the case constant of *Bool*. From now on *Bool* will not be treated separately but together with the other algebraic types.

Examples of recursive algebraic data types used in this paper will be:

- $(List\ \alpha)$ for the usual polymorphic lists with the two constructors $\texttt{Nil}_{(List\ \alpha)}$ and $\texttt{Cons}_{\alpha \to (List\ \alpha) \to (List\ \alpha)}$ and
- *Tree* for unlabeled binary trees with the constructor constants $\texttt{Tnil}_{Tree}$ and $\texttt{Tcons}_{Tree \to Tree \to Tree}$.

In most cases type subscripts of variables and constants will be dropped.
Functions can be applied curried, i.e. not saturated with enough arguments.
As notational convention $C[t]$ will denote an expression which has at some position the subexpression t.

2.1.3 Semantics

The operational semantics consists of the usual δ-reductions. The operational semantics is lazy, i.e. the topmost-leftmost redex is reduced.

δ-rules for built-in primitive functions are the usual ones. For every case constant of an algebraic type there is a δ-rule of the following form:

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

Super-combinators simply rewrite with their definition, i.e.
$(sc\ e_1 \ldots e_n) \rightarrow e[x_1 \mapsto e_1, \ldots x_n \mapsto e_n]$, for a super-combinator definition $(sc\ e_1 \ldots e_n) = e$.

An expression that cannot be reduced any further is in normal-form.

2.2 Abstract Expressions

Now we will define $\Lambda_c^\#$, the language of abstract expressions. $\Lambda_c^\#$ is the same as Λ_c, except that it includes a set of special variables $\mathcal{T} = \{\mathbf{T}^{x_1}, \mathbf{T}^{x_2}, \ldots\}$. The variables $\mathbf{T}^x$ are called abstract variables. The abstract variables have the most general type consisting of a type variable. $\Lambda_c^\#$ is then typed in the same type system as Λ_c.

Abstract variables can be substituted by expressions, which are in normal-form. We define the notion of a substitution:

Definition 1. A substitution $\sigma : \mathcal{V} \rightarrow \Lambda_c$ is a mapping from a set $\mathcal{V}$ of variables to Λ_c such that $Dom(\sigma) = \{x \in \mathcal{V} | \sigma(x) \neq x\}$ is finite.

Therefore, we can represent a substitution σ in the following way:

$$\sigma(t) = t[v_1 \mapsto \sigma(v_1), \ldots, v_n \mapsto \sigma(v_n)], \text{ where } Dom(\sigma) = \{v_1, \ldots, v_n\}.$$

A substitution σ can be lifted in the natural way to a mapping from $\Lambda_c^\#$ to Λ_c by $\sigma(e_1\ e_2) = (\sigma(e_1)\ \sigma(e_2))$.

A concretisation of $t \in \Lambda_c^\#$ is a substitution
$\gamma : \mathcal{T} \rightarrow \{e | e \in \Lambda_c$ and e has a normal-form$\}$, such that γ is type consistent, i.e. every abstract variable is substituted with an expression of the type this abstract variable has been derived for the abstract expression. This means that $\gamma(t) \in \Lambda_c$.

$\Lambda_c^\#$-expressions represent sets of Λ_c-expressions. To make things easier we will also call the result of an application of a concretisation to an abstract expression a concretisation.

Note that a concretisation is always well-typed, but there may be differently typed concretisations for a $t \in \Lambda_c^\#$.

Finally there is a special sort of abstract expressions we would like to distinguish syntactically:

Definition 2. An abstract expression $e \in \Lambda_c^\#$ is in abstract normal-form, if it only consists of constructors and abstract variables.

Note that the definition of abstract normal-forms differs from the definition of normal-form for concrete Λ_c-expressions. The notion of abstract normal-form is not defined as abstract expressions which cannot be reduced any further. This makes a significant difference for abstract expressions of function-type. A function that solely consists of a constructor, i.e. a constructor that is not applied to further expressions is in abstract normal-form, whereas an abstract expression which solely consists of a supercombinator is not in abstract normal-form.

2.3 Termination Tableaux

In this section we will define an abstract reduction calculus in terms of termination tableaux. Termination tableaux are defined as a deduction calculus which follows the ideas of tableaux calculus for predicate logics [Smu71]. Tableau-like methods have also been used in the context of program verification [Bac86]. We will define termination tableaux dually to the strictness tableaux presented in [SSPS95].

In order to prove termination we have to define the notion of termination for Λ_c-expressions. In a pure functional language there can be different notions of termination, e.g. an expression can be defined as terminating, if:

- applicative-order of reduction produces a normal-form
- or normal-order of reduction produces a head normal-form.

In fact, our basic notion of termination will be the existence of a normal-form:

Definition 3. A Λ_c-expression nf-terminates, if it can be reduced to a normal-form.

A $\Lambda_c^\#$-expression e nf-terminates, if all concretisations of e nf-terminate.

A super-combinator sc of arity n nf-terminates, if $(sc\ \mathbf{T}^{x_1} \ldots \mathbf{T}^{x_n}) \in \Lambda_c$ nf-terminates.

Note that this notion of nf-termination is a bit overloaded. A super-combinator name is a Λ_c-expression, which cannot be reduced any further, and therefore this expression is nf-terminating; but throughout this paper nf-termination for an expression of a function-type means, nf-termination of its application to nf-terminating arguments. Now it becomes clear why we did not use the alternative definition for the notion of abstract normal-form: we can guarantee that

every concretisation of an abstract normal-form of function-type will be a nf-terminating function in the sense we use this term from now on.

It is our aim to prove nf-termination of a certain super-combinator *sc*. Therefore, we will usually analyze the abstract expression $(sc\ \mathbf{T}^{x_1} \ldots \mathbf{T}^{x_n})$ for nf-termination. Unlike known methods for termination proofs on strict languages we do not prove that any reduction sequence will yield a normal-form, but that there exists a reduction sequence resulting in a normal-form.

We did not provide a special abstract value which represents all non-terminating expressions. Thze reason for this is that we can just define a supercombinator which we will use as an representative for non-terminating expressions:

```
bot = bot
```

Note that just like $\mathbf{T}$ `bot` has the most general type of a type variable.

We can now introduce a syntactical ordering relation on $\Lambda_c^\#$. This relation will allow us to draw conclusion about nf-termination of certain abstract expressions.

Definition 4. We define the relation $\leq$ on $\Lambda_c^\#$:

- $\mathbf{T}^x \leq e$, for all $e \in \Lambda_c^\#$
- $e \leq e$ for all $e \in \Lambda_c^\#$
- $e \leq$`bot` for all $e \in \Lambda_c^\#$
- $(f\ e_1 \ldots e_n) \leq (f\ e'_1 \ldots e'_n)$, iff $e_i \leq e'_i$ for $1 \leq i \leq n$

Lemma 5. *If* $e \leq e'$ *then*
e' *nf-terminates* $\rightarrow$ e *nf-terminates*

Proof. We will prove the contraposition:
e does not nf-terminate $\rightarrow$ e' does not nf-terminate.

We will use structural induction for the proof.
First we consider the base cases:

- $\mathbf{T}^x = e$: e is nf-terminating!
- $e = e'$: e' nf-terminates iff e nf-terminates!
- $e' =$`bot`: e is not nf-terminating!

Now we treat the recursive case:

$$e = (f\ e_1 \dots e_n) \leq (f\ e'_1 \dots e'_n) = e'$$

and $e_i \leq e'_i$ for all i.

We assume that the lemma has already been proved for smaller terms than e, i.e. is true for all e_i. There are 3 cases:

- f is a constructor:
  If e is not nf-terminating, then there exists an i with e_i is not nf-terminating. By the induction hypothesis we can conclude that e'_i and therefore e' is not nf-terminating.

- f is a supercombinator:
  If e is not nf-terminating, then there is a concretisation γ of e which is not nf-terminating, i.e. $\gamma(e)$ is not nf-terminating. γ can be extended to a concretisation γ' of e'. Evaluation of $\gamma(e)$ in normal-order will have to evaluate a redex which does not have a head normal-form. Evaluation of $\gamma'(e')$ in normal-order will have to evaluate the same redex. Therefore $\gamma'(e')$ is not nf-terminating which means that e' is not nf-terminating.

With the notion of nf-termination we can define termination tableaux.

Definition 6. A tableau is a finite tree whose nodes are marked with $\Lambda_c^\#$ expressions.

A tableau is sound if for every node n marked with e we have:
Let $n_1, \dots, n_k$ be the direct sons of n and n_i be marked with e_i then:
if for all $1 \leq i \leq n$ e_i nf-terminates then e nf-terminates.

Now we develop a calculus which derives new sound tableaux with the same root node from a sound tableau. The new tableaux will always originate from an old one by extending a path with new leaves.

Termination proofs with tableaux will be made in the following way. A tableaux with one single node, which represents the termination assumption, will be expanded by adding new leaves with expansion rules. A closing rule will check, if a thus generated tableau can be closed by an ordering relation on terms.

The mark on the root of a closed tableau is sure to terminate for all its concretisations.

2.3.1 Expansion-rules

δ-rules

The most simple form of extending a tableau is to perform a concrete reduction step on a leaf. An abstract expression is almost a concrete expression. This enables us to reduce at positions where there is no variable T^x involved with the same δ-reduction as in the concrete case. Performing reduction on leaves will create new sound tableaux. We define the first deduction rule for termination tableaux:

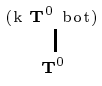
> Suppose we have in a given tableau a leaf marked with $C[\mathbf{c}_\tau\ e_1 \ldots e_n]$. A δ-reduction on this tableau is performed by extending this tableau with the new leaf $C[e']$, if there is a δ-rule: $\mathbf{c}_\tau\ e_1 \ldots e_n \rightarrow_\delta e'$ for $\mathbf{c}_\tau$.

We have to show that δ-reductions on tableaux preserve soundness. Let e be the old leaf and e' the new leaf which is the direct son of e and was introduced by a δ-reduction. Every concretisation of e can be reduced by a concrete δ-reduction with an concretisation of e'. This means that if all concretisations of e' nf-terminate then all concretisations of e will nf-terminate.

In formal notation: $e \rightarrow e'$ implies $\gamma(e) \rightarrow \gamma(e')$ for all concretisations γ.

Let us consider the following super-combinator definitions and a very simple tableau:

```
bot = bot
k x y = x
```



By lemma 5 the tableau above proves that the super-combinator `k` nf-terminates.

Note that if we extend a tableau with the rule above, then every concretisation of the parent rule can be reduced to a concretisation of the new leaf in one step.

Abstract δ-rules

If we have an application of a built-in constant to an abstract variable then we can perform an abstract reduction step. There are two different abstract reduction steps: reduction of built-in arithmetic functions to abstract variables and a *case* on an abstract variable.

Arithmetic functions If an arithmetic function is applied to an abstract variable a new abstract variable is the result:

> $\otimes\ \mathbf{T}^a\ \mathbf{T}^b \rightarrow_\delta \mathbf{T}^c$, $\otimes\ \mathbf{T}^a\ \mathbf{c} \rightarrow_\delta \mathbf{T}^c$, $\otimes\ \mathbf{c}\ \mathbf{T}^a \rightarrow_\delta \mathbf{T}^c$,
> for $\otimes \in \{+, -, *, =, <, >, \leq, \geq\}$
> where $\mathbf{T}^c$ is a new abstract variable in the tableau and $\mathbf{c}$ is some constructor.

It can be seen that no termination can be shown for functions that depend in their termination behavior on basic values. To overcome this deficiency one has to simulate basic values and primitive functions by an algebraic type. So some sort of Peano-arithmetic is needed which is defined with the constructor `Null` and `Succ`. If this is done, analyzing functions on basic types will not cause any special problem.

The only thing that has to be kept in mind is that these functions are hyperstrict. The algebraic type for *Num* would be able to describe infinite numbers. Multiplication of an infinite number with `Null` then would terminate with `Null`, whereas in the strict function it would not terminate. Therefore, δ-rules for an algebraic Num have to be applied in applicative order of reduction in a termination tableau in order to get sound results.

The reader will have noticed that '/' has been left out above. The reason for that is that division by zero is undefined, i.e. it is an non-terminating expression[1]. Therefore, $(/\ e\ \mathbf{T}^c)$ is irreducible, whereas $(/\ \mathbf{T}^a\ \mathbf{c}) \rightarrow_\delta \mathbf{T}^c$ for $c \neq 0$.

We do not allow to draw any conclusion from abstract variables via built-in functions; e.g. a reduction of the form $(\mathbf{T}^a\ =\ \mathbf{T}^a) \rightarrow_\delta \mathbf{True}$ is not allowed.

Note again that every concretisation of the parent node can be reduced to a concretisation of the new leaf.

Decomposition Now we introduce a deduction rule for branching a tableau. This will be necessary when a case expression is applied to an abstract variable. For such an application we know that each instance of the abstract expression will reduce to one of the alternatives of the expression. We define the decomposition rule for termination tableaux:

> Suppose we have in a given tableau a leaf marked with
> $t = C[(\mathbf{case}_\tau\ \mathbf{T}^c\ e_1 \ldots e_n)]$,
> where the algebraic type τ has the set of constructors
> $\{\mathbf{con}_{\tau_1}, \ldots, \mathbf{con}_{\tau_m}\}$ of types: $\tau_{1_i} \rightarrow \ldots \rightarrow \tau_{k_i} \rightarrow \tau,\ 1 \leq i \leq m$.

[1] Non-termination is the only way to express undefined values in our language.

A decomposition of this leaf is performed by extending it with the m new leaves $C[t_i], 1 \leq i \leq m$ such that $t_i = (e_i\ \mathbf{T}^{c_{i_1}} \ldots \mathbf{T}^{c_{i_k}})$.

$\mathbf{T}^{c_{i_j}}$ are new abstract variables in the tableau.

The new edges get marked with the substitutions which produced the new leaf, i.e. $\mathbf{T}^c = (\mathbf{con}_{\tau_i}\ \mathbf{T}^{c_{i_1}} \ldots \mathbf{T}^{c_{i_k}})$.

Although the rule of decomposition looks a bit complex not very much happens here. All that is done is to introduce for a variables $\mathbf{T}^c$ new subexpressions that involve further such variables. This is done in order to enable δ-reduction for the case constant. The substitutions on the edges notify which alternative of the case had been applied. On a path in a tableau there may be several such substitutions $\sigma_1, \ldots, \sigma_n$. To get the right form of the start node of this path, we have to compose all these substitutions to $\sigma_n \circ \ldots \circ \sigma_1$.

The soundness of the decomposition is quite obvious: as well as in the expansion rules before every concretisation of the parent node can be reduced to a concretisation of one of the new leaves.

Note that in this rule we need type information in order to perform reduction in a tableau. It has to be specified of what type the constant **case** is and how many constructors there are for this algebraic type. This will be the only point where the tableau calculus needs type information. Fortunately this is very basic information.

Note that there remain irreducible abstract expressions: applications of the form $(\mathbf{T}^c\ e)$. We will postpone the treatment of such applications to a later section.

Context Skipping and T-Introduction

Now we provide two rules that might require information from other tableaux or prior knowledge of certain expressions. The first enables us to drop parts of which we know that they cannot cause any non-termination. Such parts can be of two kinds: a top-level constructor constant or a top-level super-combinator application of a super-combinator that is known to be nf-terminating. Thus, we define two sub-rules for context skipping:

Suppose we have in a given tableau a leaf marked with $(\mathbf{c}_{\tau_i}\ e_1 \ldots e_n)$. Then a new tableau may be deduced by extending at this leaf with the n new leaves $e_1, \ldots, e_n$.

Suppose we have in a given tableau a leaf marked with $(sc\ e_1 \ldots e_n)$ where sc is a super-combinator that is known to be nf-terminating . Then a new tableau may be deduced by extending at this leaf with the n new leaves $e_1, \ldots, e_n$.

For every node which has been expanded with one of the rules of context skipping we can say: all concretisations of this node can be transformed to concretisations of one of its new leaves by skipping the context of this concretisation.

The second rule we provide in this section allows us to substitute every abstract term, that has only nf-terminating concretisations, with a new variable $\mathbf{T}$:

> Suppose we have in a given tableau a leaf marked with $C[e]$ and e is nf-terminating. Then this path may be extended with the new leaf: $C[\mathbf{T}^c]$, where c is a new label in this path.

Every concretisation of the parent node can be reduced by finitely many reduction steps to a concretisation of the leaf node introduced by $\mathbf{T}$-introduction.

This rule subsumes in a way abstract reduction on arithmetic expressions. An expression$(+\ \mathbf{T}^a\ \mathbf{T}^b)$ can be reduced or approximated with a new abstract variable $\mathbf{T}^c$.

The rules in this subsection preserve soundness.

2.3.2 Closing a Tableau

Now it needs to be defined under which circumstances a given tableau can be closed, because it constitutes a termination proof. There are two different kinds of leaf nodes, which are important for closing a tableau:

- leaf nodes which are in abstract normal-form
- and leaf nodes which have an ancestor node, such that all concretisations of the leaf node are concretisations of the ancestor node.

Leaf nodes which are in abstract normal-form have only nf-terminating concretisations and give directly nf-termination. The rule of $\mathbf{T}$-introduction allows to approximate such nodes with an abstract variable.

For leaf nodes with an ancestor we have to find a Noetherian ordering on the arguments to show that the reduction process on the path did some minimization.

Simple recursive paths

Let us define how such recursive nodes look like:

Definition 7. A leaf node $e' = C[e_1, \ldots, e_n], (n > 0)$ in a tableau is called a recursive node, if all e_i are abstract normal-forms and there is an ancestor node $e = C[\mathbf{T}^{x_1}, \ldots, \mathbf{T}^{x_n}]$, such that there is a substitution σ with $\sigma(e) = e'$.

(e, e') is called a recursive pair and σ the corresponding substitution.

We define the class of tableaux which are candidates for a termination proof:

Definition 8. A tableaux where every leaf is either in abstract normal-form or a recursive node, is called *preclosed.*

Recursive pairs have the desired property:

Lemma 9. *Let (e, e') be a recursive pair. Then every concretisation $\gamma(e')$ of e' is a concretisation of e*

Proof. There exists a substitution σ with $\sigma(e) = e'$. Thus $\gamma(e') = \gamma(\sigma(e))$.
$\gamma' = \gamma \circ \sigma$ is a concretisation map, if σ is consistent with the type system. It remains to show that σ is type-consistent. Assume this to be false: this means that e' is not a well-typed abstract expression and therefore does not have any concretisation, which makes the lemma trivially true.

With the remarks made for every expansion rule we can state a little proposition:

Lemma 10. *Let $\mathcal{T}$ be a preclosed tableau. Then every concretisation of a node in $\mathcal{T}$ is a normal-form or can be transformed by a transformation corresponding to an expansion-rule to a concretisation of at least one further node in $\mathcal{T}$.*

Proof. If the concretisation in question is not a concretisation of a leaf node, then this is true by the remarks about each expansion rule, i.e. a concrete reduction or a context-skipping on a concrete expression can be performed to get the concretisation of a successor node. If it is a concretisation of a leaf node, then this leaf node is either in abstract normal-form and the concretisation is in normal-form, or the node is a recursive node and therefore by lemma 9 the concretisation is also a concretisation of a none-leaf node.

This lemma can be extended a bit by transitivity. This means that every concretisation of a node in such a tableau leaves at least one *trace* in the tableau which may end in a normal-form or go on infinitely. Thus we can follow any non-terminating (normal-order) reduction of a root note concretisation in the finite preclosed tableau. We will call such a sequences of concretisations which follow the edges in a tableau a trace:

Definition 11. A trace of a path in a tableau is a sequence of Λ_c-expressions which correspond via concretisations one-to-one to the nodes in the path and the Λ_c-expressions in the sequence can be obtained by the transformations on concrete expressions corresponding to expansion rules as shown in lemma 10.

Orderings

We have reached the point, where we want to show that the loops we found are harmless, i.e. will not be executed infinitely often. A recursive pair induces a constraint on an ordering relation for the arguments. A Noetherian ordering has to be found which decreases the arguments of the recursive calls. What we need is an ordering on Λ_c-expressions. This ordering can then be lifted to $\Lambda_c^\#$-expressions and to tuples of $\Lambda_c^\#$-expressions.

Definition 12. Let $\geq$ be a reflexive partial ordering relation on Λ_c-expressions.

- for two concretisations γ_1, γ_2 with $\mathcal{DOM}(\gamma_1) = \mathcal{DOM}(\gamma_2)$ we define: $\gamma_1 \geq \gamma_2$ iff $\gamma_1(\mathbf{T}^x) \geq \gamma_2(\mathbf{T}^x)$, for all $\mathbf{T}^x$.
- for $(e_1, \ldots, e_n), (e'_1, \ldots, e'_n) \in \Lambda_c \times \ldots \times \Lambda_c$ we can define orderings $(e_1, \ldots, e_n) \geq_{\text{tup}} (e'_1, \ldots, e'_n)$ such that $>_{\text{tup}}$ is Noetherian whenever $>$ is Noetherian. A simple such ordering of tuples is: $(e_1, \ldots, e_n) \geq_{\text{tup}} (e'_1, \ldots, e'_n)$ iff for all $1 \leq i \leq n$: $e_i \geq e'_i$.
- for a partial ordering $\geq$ we have in the usual way: $e_1 = e_2$ iff. $e_1 \geq e_2$ and $e_2 \geq e_1$. $e_1 > e_2$ iff. $e_1 \geq e_2$ and not $e_2 \geq e_1$.

Let us make a few remarks:

By this definition a tuple-ordering can theoretically be completely independent from the ordering on Λ_c. Throughout this paper we will only use tuple-orderings which are defined by: $(e_1, \ldots, e_n) \geq_{\text{tup}} (e'_1, \ldots, e'_n)$ iff for some $1 \leq i \leq n$: $e_i \geq e'_i$.

We completely neglected types in the definition of orderings. And in fact, due to polymorphic types, it can be useful to compare two expressions of different types. Consider e.g. the algebraic data-types of lists which have elements of alternating types. In Haskell-syntax we can define such a data type as:

```
data AList a b
 = Nil
  |Cons a (AList b a)
```

In order to prove the termination of some sort of length-function[2] on alternating lists it might be necessary to compare two different instances with each other, say a list of type `(AList Int Bool)` with a list of type `(AList Bool Int)`.

But both concretisations are of a type instance of the upper node in the recursive path.

[2] Such a simple length-function cannot be typed with the Milner type system.

We now specify what an ordering has to fulfill in order to serve for a termination proof:

Definition 13. For a recursive pair (e, e') with the corresponding substitution $\sigma = [\mathbf{T}^{x_1} \mapsto t_1, \ldots, \mathbf{T}^{x_n} \mapsto t_n]$ its ordering constraint is defined as:
$(\tau(\mathbf{T}^{x_1}), \ldots, \tau(\mathbf{T}^{x_n})) >_{\text{tup}} (t_1, \ldots, t_n)$,
where τ is the composition of all substitutions on the path from e to e'.

The constraint is to be read as:
there exists partial ordering on Λ_c and a tuple ordering such that for all concretisations γ:
$(\gamma \circ \tau(\mathbf{T}^{x_1}), \ldots, \gamma \circ \tau(\mathbf{T}^{x_n})) >_{\text{tup}} (\gamma(t_1), \ldots, \gamma(t_n))$.

Note that through the definition of recursive pairs the t_i above are in abstract normal-form.

What we are looking for is an ordering on Λ_c-expressions, such that there is a tuple-ordering for the arguments of recursive pairs which is a Noetherian ordering. This ordering then induces a ordering on concretisations of recursive pairs. This means that the path between a recursive pair minimizes an expression. Here we can plug-in modules which try to find one of the numerous different orderings for termination proofs proposed in the literature, e.g. polynomial orderings as proposed in [Ste92, Lan79], one of the orderings presented in [Der87] or some sort of generalized orderings based on multi-sets as presented in [Mar87]. A simple ordering which in many cases is sufficient enough is the ordering which is based on the number of constructors the normal-form of an expression has.

Lemma 14. *Let (e, e') be a recursive pair such that its ordering constraint can be fulfilled. Then for every concretisation $k = \gamma(e)$ which has a direct trace to a concretisation $k' = \gamma(e')$ the tuple of substituted expressions in the context decreases.*

Proof. The composition τ of the substitutions on the path assures that every concretisation $k = \gamma(e)$ with a direct trace to e' is a concretisation of $\tau(e)$, i.e. there is a concretisation γ_1 with: $k = \gamma_1 \circ \tau(e)$. Because k' is derived as a direct trace of the recursive path from $k = \gamma_1 \circ \tau(e)$ we have: $\gamma_1 = \gamma'$. We have found a concretisation to which we can apply the conditions of the fulfilled ordering constraint.

This lemma gives rise to a further definition of orderings:

Definition 15. Let $e = C[\mathbf{T}^{x_1}, \ldots, \mathbf{T}^{x_n}]$ be an abstract expression, γ_1 and γ_2 concretisations of e, $>$ an ordering on Λ_c and $>_{tup}$ a tuple-ordering based on $>$.

Then $\gamma_1(e) >_e \gamma_2(e)$ is defined as:
$(\gamma_1(\mathbf{T}^{x_1}), \ldots, \gamma_1(\mathbf{T}^{x_1})) >_{tup} (\gamma_2(\mathbf{T}^{x_1}), \ldots, \gamma_2(\mathbf{T}^{x_1}))$

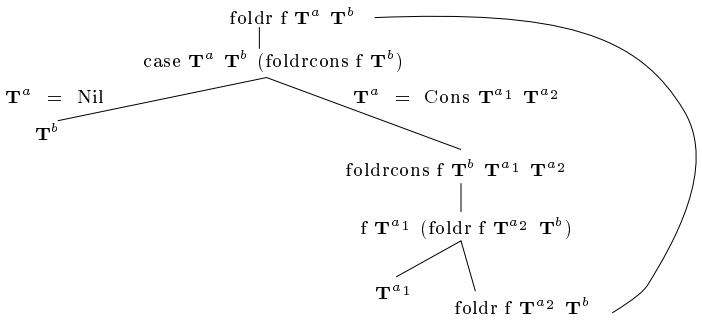


Figure 2.1: A closed tableau for `foldr`.
Recursive pairs are marked with an arc.
The function argument f is supposed to be nf-terminating.

Closing a tableau will now be the task of finding orderings such that every leaf node is either an abstract normal-form or a recursive node, which has an ordering constraint, that can be fulfilled by some ordering.

Before we consider when the closing of a preclosed tableau is sound, we will give an example:

Example 1. We want to analyze the function `foldr` from the Haskell prelude:

```
foldr f xs n = case xs n (foldrcons f n)
foldrcons f n x xs = f x (foldr f xs n)
```

We can deduce the tableau in figure 2.1.

The tableau is preclosed. There are three leaf nodes. Two of them are in abstract normal-form. The third is a recursive node with the ordering constraint: $(\text{cons } \mathbf{T}^{a_1}\ \mathbf{T}^{a_2}) > (\mathbf{T}^{a_2})$. A simple ordering on Λ_c which fulfills this constraint is the textual size of an expression. Note that we only have to consider the second argument of `foldr` and therefore, do not need any tuple-ordering.

For the case that there are no two recursive pairs with paths that have a node in common, soundness of the closing condition for tableaux is quite obvious.

Theorem 16. *Let $\mathcal{T}$ be a preclosed tableau derived with the rules above, such that there are no two recursive pairs with paths that have a node in common.*
If for every recursive path the ordering constraint can be fulfilled by a Noetherian ordering then all concretisations of the root are nf-terminating.

Proof. Assume by contradiction that there is a concretisation of the root which is not nf-terminating. By contraposition of the soundness for the expansion rules, there is a leaf node with a non-nf-terminating concretisation. This leaf has to be a recursive node.

a) Choose such a recursive pair (e, e') such that the distance between root and e' is maximal.

b) Choose a concretisation $k = \gamma(e')$ such that the concretisation of the tuple of abstract variables is minimal in the ordering which was used to close this path, i.e. minimal in $>_{e'}$.

k is by lemma 9 also a concretisation of e. There is by lemma 10 a trace of this concretisation k to a leaf node. This has to be another leaf node than e' because k was a minimal concretisation of e' and the path to e' by lemma 14 minimizes concretisations. Otherwise we would contradict the fact that k is minimal.
The new leaf is part of an recursive pair, but due to the maximal length between the root and e the upper node of the new recursive pair is above e. This contradicts our assumption that there are no two recursive paths which have any nodes in common.

Note that there could possibly be two different recursive paths to the same leaf node. In this case it suffices to consider only one of them. The obvious choice is the nearest one.

In this theorem it is not necessary that the same ordering on Λ_c and the same tuple ordering is used to fulfill the ordering constraints of different recursive pairs. The recursive pairs are independent from each other. Especially the tuple orderings used may discard certain tuple elements, i.e. we do not care for the orderings of these elements.

Example 2. Theorem 16 enables us to prove termination of a function which entails two different minimization processes. Consider e.g. the following function:

```
shiftremove t1 t2 = case t1 (case t2 Tnil shiftremove1)
                            (shiftremove2 t2)
shiftremove1 t1 t2 = shiftremove Tnil t1
shiftremove2 t3 t1 t2
 = shiftremove t1 (TCons (TCons (TCons t1 t2) (TCons t1 t2))
                         (TCons (TCons t1 t2) (TCons t3 t3)))
```

`shiftremove` deconstructs the tree in its first argument and lets the second argument grow and then deconstructs its second argument.

A sound tableau for this function is given in figure 2.2. It can be closed by theorem 16 independently at the two recursive leaves by two simple orderings. At both recursive paths we can use the ordering on Λ_c which orders trees by the number of constructors their normal-forms have.

At the left recursive node the tuple to minimize has just one element. The simple tuple-ordering which demands that this element decreases will do. At the right recursive node this is a tuple ordering which just considers the size of the first argument. Note that on the left branch there has been the recursive node

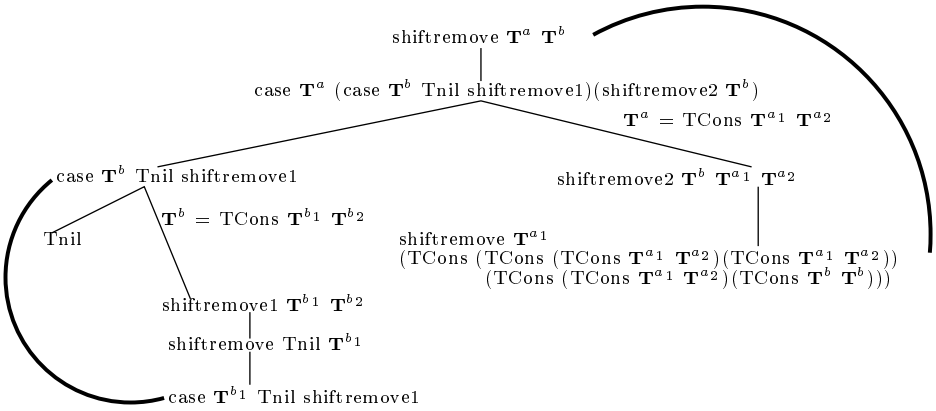


Figure 2.2: A closed tableau for `shiftremove`

(shiftremove TNil $\mathbf{T}^{b_2}$) before, but this was not independent from the recursive node on the right hand side. To close the tableau at that stage requires a more complex ordering, which cannot be generated by the polynom-orderings used in [Gie95b].

Overlapping recursive paths

Finally we have to deal with the situation when a tableau can only be closed, if we have recursive pairs with overlapping paths.

The problems in such situations are that the overlapping recursive pairs have different upper nodes, i.e. they may have a different context. Then we cannot apply the same orderings for these two recursive nodes. (The orderings are dependent on the context of a recursive node.)

Consider the following counterexample where a preclosed tableau with overlapping paths, does not give a termination proof:

Example 3. We define a super-combinator which has two binary trees and one boolean value as arguments.

```
foo t1 t2 b
 = case t1 TNil (foo1 t2 b)

foo1 t2 b t11 t12
 = case t2 Nil (foo2 t11 t12 b)

foo2 t11 t12 b t21 t22
 = if b (foo t11 (TCons (TCons t21 t22) t11) False)
        (foo (TCons (TCons t11 t12) t21) t21 True)
```

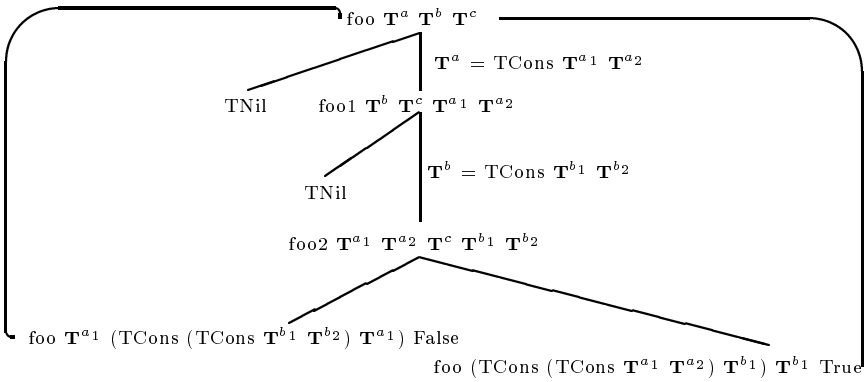


Figure 2.3: A preclosed tableau for `foo`.

`foo` is not nf-terminating. There is the following looping reduction:

```
foo (TCons (TCons TNil TNil)(TCons TNil TNil))
    (TCons TNil TNil) True
→ foo2 (TCons TNil TNil)(TCons TNil TNil) True TNil TNil
→ foo (TCons TNil TNil)
    (TCons (TCons TNil TNil)(TCons TNil TNil)) False
→ foo2 TNil TNil False (TCons TNil TNil)(TCons TNil TNil)
→ foo (TCons (TCons TNil TNil)(TCons TNil TNil))
    (TCons TNil TNil) True
```

We can derive the preclosed tableau in figure. 2.3.

There are two recursive paths:
(Foo $\mathbf{T}^a$ $\mathbf{T}^b$ $\mathbf{T}^c$, foo $\mathbf{T}^{a_1}$ (TCons (TCons $\mathbf{T}^{b_1}$ $\mathbf{T}^{b_2}$) $\mathbf{T}^{a_1}$) False) and
(Foo $\mathbf{T}^a$ $\mathbf{T}^b$ $\mathbf{T}^c$, foo (TCons (TCons $\mathbf{T}^{a_1}$ $\mathbf{T}^{a_2}$) $\mathbf{T}^{b_1}$) $\mathbf{T}^{b_1}$ True)
For both of these recursive paths the ordering constraint can be fulfilled. But it is not allowed to conclude that `foo` is nf-terminating, because the two paths interfere with each other's orderings.

The obvious solution to this problem is to demand that for such overlapping recursive paths the same orderings are being applied. This can only be stated, if the overlapping paths have the same context in the start node, because otherwise they have different tuples of different context to order and it is not clear what a combined ordering looks like.

Lemma 17. *Let $\mathcal{T}$ be a preclosed tableau derived with the rules above, such that overlapping recursive paths have the same start node.*
If for every recursive path the ordering constraint can be fulfilled with some Noetherian ordering such that for overlapping recursive paths the same orderings are used, then all concretisations of the root are nf-terminating.

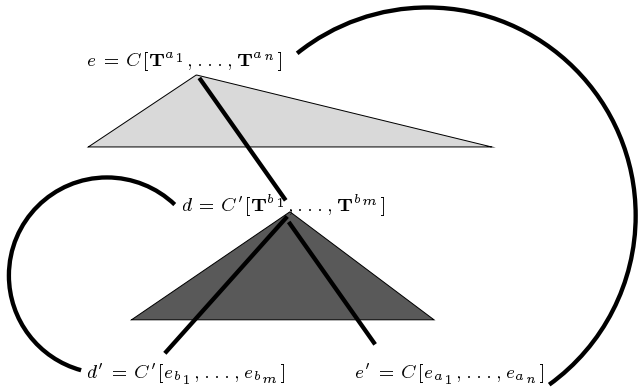


Figure 2.4: A schematic tableau with two overlapping recursive paths

Proof. The situation has not changed very much from the one in theorem 16. We can proceed in the same way as in the proof of theorem 16 up to a). In b) we do not have to consider only one recursive pair, but all recursive pairs which have the same upper node as the pair (e, e') chosen in a), i.e. there are recursive pairs $(e, e'_1), \ldots, (e, e'_n)$. In b) we now can choose the smallest concretisation of all e'_i, because these concretisations depend on the same context. Then proceed in the same way as in the proof of theorem 16.

It remains to deal with overlapping recursive paths which do not have the same start node.

We can try to flatten a preclosed tableau with overlapping recursive paths to a preclosed tableau which does not have overlapping recursive paths. What we may have is a situation as described in figure 2.4.

Here we have two different recursive pairs: (e, e') and (d, d') with different contexts C and C'. We cannot simply define a global ordering which fulfills both ordering constraints of these recursive pairs, because our orderings depend on the context (the tuple of substituted terms in the context). But what we can try to do is, expanding the tableau in such a way that we get two recursive paths with the same start node and the same context, as is shown in figure 2.5.

In this way we can try extend every tableau which has overlapping recursive pairs to a tableau where all overlapping recursive pairs have the same context.

Unfortunately this is not always possible.

Example 4. As a counterexample consider the following function in Haskell syntax:

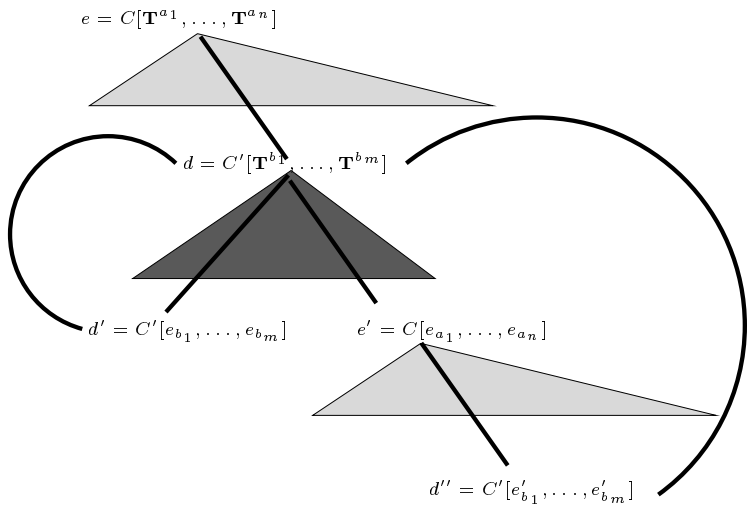


Figure 2.5: Extending tableaux of the form in figure 2.4.

```
data Z = N | S0 Z | S1 Z

f N = N
f (S0 x) = g x
f (S1 x) = f x

g N = N
g (S0 x) = g x
g (S1 x) = f x
```

In Λ_c this looks like:

```
f x = case x N g f
g x = case x N f g
```

As can be seen `f` and `g` have almost identical definitions. Now let us build a termination tableau for one of these functions. We can get the tableau in figure 2.6.

In which way ever we further expand this tableau there is no chance to get a tableau which does not have overlapping recursive pairs. Nevertheless `f` is of course nf-terminating.

In the rest of this section we will see, under which circumstances a preclosed tableau with overlapping recursive paths gives a termination proof. First we need some condition on the orderings on Λ_c.

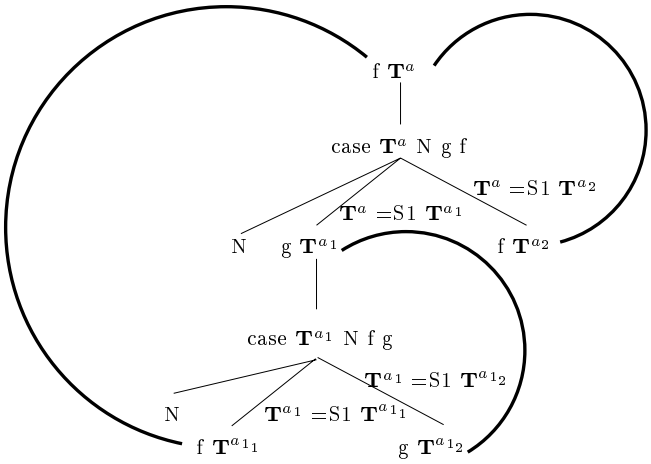


Figure 2.6: A preclosed tableau for `f`. Overlapping paths cannot be eliminated by further expansion.

Definition 18. A partial ordering $\leq$ on Λ_c is called constructor monotonic, iff $e_1 \leq e_1' \ldots e_n \leq e_n'$ implies $(\mathbf{c}\ e_1 \ldots e_n) \leq (\mathbf{c}\ e_1' \ldots e_n')$ for all well-typed expressions.

We can now show that edges in a tableau preserve an ordering relation from successor to predecessor node:

Lemma 19. *Let (e, e') be an edge which does not represent a $\mathbf{T}$-introduction. Let $\leq$ be constructor monotonic on Λ_c and $n_1 \leq n_2$ for all n_1, n_2 of a built in basic type.*
Let $\gamma_1^{e'}(e') = k_1^{e'}$ and $\gamma_2^{e'}(e') = k_2^{e'}$ be concretisations such that $\gamma_1^{e'} \leq \gamma_2^{e'}$. Let $\gamma_1^{e'}(\mathbf{T}^x)$ and $\gamma_2^{e'}(\mathbf{T}^x)$ be of the same type for all variables $\mathbf{T}^x$.
Then for all concretisations $\gamma_2^e(e) = k_2^e$ such that $(k_2^e, k_2^{e'})$ is a trace of (e, e') there exists a concretisation $\gamma_1^e(e) = k_1^e$ such that $(k_1^e, k_1^{e'})$ is a trace of (e, e') and: $\gamma_1^e \leq \gamma_2^e$.

Picture 2.7 gives an illustration of this lemma.

Proof. We have to show that the proposition is true for all sorts of edges in a tableau. We will construct a γ_1^e for the different kinds of edges:

- **concrete δ-reductions**: e and e' have the same set of abstract variables (otherwise we have the case of a non-branching abstract reduction). $\gamma_2^{e'} = \gamma_2^e$ and $\gamma_1^{e'} = \gamma_1^e$.
- **abstract δ-reductions**:

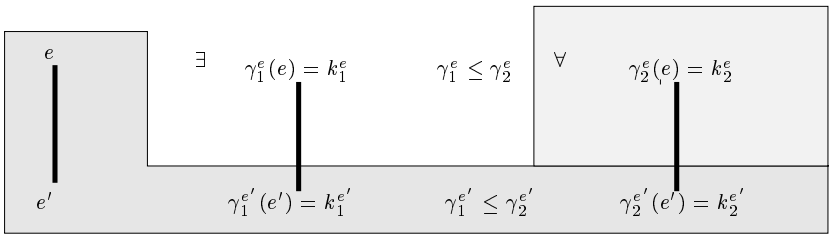


Figure 2.7: An illustration of lemma 19

- **built-in functions**: The abstract variables $\mathbf{T}^x$ occuring in e and not occuring in e' can only be substituted by expressions of a built-in type (because it appears as argument of a built in function). $\gamma_i^e(\mathbf{T}^y) = \gamma_i^{e'}(\mathbf{T}^y)$ for $y \neq x$ and $i \in \{1,2\}$ and $\gamma_1^e(\mathbf{T}^x) \leq \gamma_2^e(\mathbf{T}^x)$.
- **non-branching**: $\{\mathbf{T}^x | \mathbf{T}^x \text{ appears in } e'\} \subseteq \{\mathbf{T}^x | \mathbf{T}^x \text{ appears in } e\}$. Just choose

$$\gamma_1^e(\mathbf{T}^x) = \begin{cases} \gamma_2^e(\mathbf{T}^x), & \text{for } \mathbf{T}^x \text{ not occuring in e'} \\ \gamma_1^{e'}(\mathbf{T}^x), & \text{else} \end{cases}$$

  As will be seen it is essential for this construction of γ_1^e that $\gamma_1^{e'}(\mathbf{T}^x)$ and $\gamma_2^{e'}(\mathbf{T}^x)$ are of the same type for all abstract variables $\mathbf{T}^x$.
- **branching**: Apart from the aspects which are the same as in the non-branching case, there is a abstract variable $\mathbf{T}^x$ in e such that the edge is marked with: $\mathbf{T}^x = (\mathbf{c}\ \mathbf{T}^{x_1} \ldots \mathbf{T}^{x_n})$.
  Therefore: $\gamma_i^e(\mathbf{T}^x) = (\mathbf{c}\ \gamma_i^{e'}(\mathbf{T}^{x_1}) \ldots \gamma_i^{e'}(\mathbf{T}^{x_n}))$.
  With $\gamma_1^{e'}(\mathbf{T}^y) \leq \gamma_2^{e'}(\mathbf{T}^y)$ for all $\mathbf{T}^x$ and constructor monotonicity of $\leq$: $\gamma_1^e(\mathbf{T}^x) \leq \gamma_2^e(\mathbf{T}^x)$.

- **context skipping**: This case is basically the same case as abstract reduction in the non-branching case. Just extend $\gamma_1^{e'}$ for the new abstract variables with $\gamma_2^{e'}$ in order to get γ_1^e. Essential for this is again, that $\gamma_1^{e'}(x)$ and $\gamma_2^{e'}(x)$ are of the same type.

The restrictions to lemma 19 are necessary.

- Lemma 19 is not true for $\mathbf{T}$-introductions. Consider the following counterexample:

  ```
  f x = case x (Cons 1 Nil) Nil
  ```

  Now we approximate (`f` $\mathbf{T}^a = e$ with $\mathbf{T}^b = e'$ and consider the concretisations: $\gamma_1^{e'}(\mathbf{T}^b) =$ `Nil` and $\gamma_2^{e'}(\mathbf{T}^b) =$ `(Cons 1 Nil)`. $\leq$ orders lists by their size. $\gamma_2^e(\mathbf{T}^a) =$ `Nil`. There is no $\gamma_1^{e'}$ as stated in lemma 19.

- For the same reason lemma 19 is not true if we apply some arbitrary ordering on built-in types. Consider a node: $(+\ \mathbf{T}^a\ 1)$ and its abstract reduct: $\mathbf{T}^b$. $\gamma_1^{e'}(\mathbf{T}^b) = 1$ and $\gamma_2^{e'}(\mathbf{T}^b) = 2$. In the ordering on integers with $1 < 2 < 0$: $\gamma_1^e(\mathbf{T}^a) = 0$ and $\gamma_2^e(\mathbf{T}^a) = 1$. Here we have $\gamma_1^e(\mathbf{T}^a) > \gamma_1^{e'}(\mathbf{T}^b)$

- Lemma 19 is not true for concretisations of different types. Consider: $(\texttt{k}\ \mathbf{T}^a\ (+\ 1\ (\texttt{head}\ \mathbf{T}^a)))$, and its reduct $\mathbf{T}^a$.
  $\gamma_1^{e'}(\mathbf{T}^a) = \texttt{(Cons True Nil)}$.
  $\gamma_2^{e'}(\mathbf{T}^a) = \texttt{(Cons 1 Nil)} = \gamma_2^e(\mathbf{T}^a)$.
  The ordering $\leq$ is on the length of lists. There is no γ_1^e with the desired property. Actually there is no trace of the edge e–e' that ends in `(Cons True Nil)`. In such situations we can restrict $\gamma_1^{e'}$ in such a way that there exists a trace of e–e' ending in $\gamma_1^{e'}$.

Let us see how lemma 19 can be used as a basis for proving the correctness of closing tableaux with overlapping paths. We consider a tableau with two overlapping recursive paths as seen in figure 2.4. There are two recursive pairs (e, e') and (d, d').

Let us assume that there is *one* Noetherian ordering relation $>$ which fulfills the ordering constraints of both recursive paths. Assume $>$ to be constructor monotonic. We also demand $>$ to be total on expressions of equal types in Λ_c. Furthermore we must restrict ourselves with the tuple-orderings used[3]. We will only allow tuple-orderings with the following property:
$(e_1, \ldots, e_n) \leq (e'_1, \ldots, e'_n)$ *iff* $e_i \leq e'_i$ for $i = 1 \ldots n$.

By lemma 14 we can conclude that every direct trace of a recursive path minimizes the concretisations of e (or d) in a certain ordering $>_e$ (or $>_d$). But there are possible traces which start at e (or d) and end at e' (or d') which do not follow directly the recursive path. Basically there are two schemes for such non-direct traces:

a) e–d–d'–e'

b) d–e', e–d–d'

We will treat both cases in the following:

a) Let us consider a trace of scheme a). We want to show that every trace of e–d–d'–e' with an arbitrary number of traces of d–d' inside the concretisation gets still minimized in $>_e$. We have to consider a trace of e–$d \xrightarrow{*} d'$–e': l–k–k'–l' with $\gamma_l(e) = l, \gamma_k(d) = k, \gamma_{k'}(d') = k', \gamma_{l'}(e') = l'$.

[3] There are two tuple-orderings involved. One for the context of e and one for d.

where e, d, d', e' are the identifiers used in figure 2.4.

We want to show that: $l >_e l'$. With lemma 19 we can built from k' backwards a trace l''–k' of the path e–d. This is so to speak a shortcut for l–$k \overset{*}{\text{—}} k'$. There is a substitution $\gamma_{l''}$ with $\gamma_{l''}(e) = l''$.

Lemma 14 assures that $k >_d k'$ with the assumptions about the tuple-orderings this means $\gamma_k > \gamma_{k'}$. So we can conclude with lemma 19: $\gamma_l > \gamma_{l''}$. With the assumptions about the tuple-orderings this means $l >_e l''$. Together with $l'' \geq_e l'$ (by lemma 14, because this is a direct trace of e–e'.) we get the desired property $l >_e l'$.

b) Let us consider a trace of scheme b). We want to show that every trace of d–e'–e–d the concretisation gets minimized in $>_d$.

We have to consider a trace of d–e–e'–d:
k–l–l–k' with $\gamma_l(e) = l, \gamma_k(d) = k, \gamma_{k'}(d) = k'$.

We want to show: $k \geq_d k'$. Let us assume that not $k \geq_d k'$. We assumed $<$ to be total, so that this means: $k <_d k'$. Now we can apply lemma 19 again. This time to construct a concretisation l' of e backwards from k. Applying lemma 19 in the same way as in a) gives: $l' <_e l$ which contradicts $l' >_e l$ which is assured by lemma 14 (remember that there is a direct trace from l' to l.).

We have tried to construct a proof for the closing of tableaux with overlapping paths above. Unfortunately we had to make quite a lot of assumptions on the orderings and on the recursive paths. We will summarize them now:

Theorem 20. *The root of preclosed tableau with overlapping recursive paths is nf-terminating, if there is a Noetherian ordering on Λ_c which closes every recursive path such that the following conditions are fulfilled:*

- *the ordering $<$ on Λ_c has to be total for expressions of same type*
- *for all expressions e_1, e_2 of basic value it holds $e_1 = e_2$ in the used ordering on Λ_c*
- *$<$ has to be constructor monotonic*
- *for the used tuple ordering it must hold:*
  $(e_1, \ldots, e_n) \leq (e'_1, \ldots, e'_n)$ iff $e_i \leq e'_i$ for $i = 1 \ldots n$.
- *a trace of a recursive path must not change the type of the concretisation*
- *no $\mathbf{T}$-approximations are allowed on recursive pairs*

Most of these properties are unproblematic. Only the restriction on recursive paths to be type preserving (for all concretisations) is rather problematic. Nevertheless in example 4 we have seen an example were all these requirements are met.

2.3.3 Strategies and further enhancements

Our calculus works so far quite fine, as long as there is only one recursive function involved in the termination behavior we want to prove. If there are more than one recursive function in our program, such that these functions are dependent on each other, we have two possibilities in handling these situations:

- Make a dependency analysis first, start analyzing the functions which are lower in the dependency-hierarchy and use these results for the rule of context-skipping when analyzing functions higher up in the dependency analysis. This strategy cannot be applied for mutual recursive functions as we have seen in example 4.
- Start analyzing some arbitrary function and proof all recursive reduction processes to be terminating in one tableau. This will usually lead to a tableau with overlapping recursive paths and other difficulties. Nevertheless, it can be more accurate, because one of the recursive functions involved may not be terminating for all possible inputs but just the inputs it is called for by another function. The degree of termination for one function which is needed to prove termination of another function may not be known beforehand.

We will see in an example how devastating it can be to choose the wrong strategy in certain cases. This subsection will conclude with a new rule for expanding a tableau which becomes necessary when recursive calls do not appear on the root of an expression.

Example 5. Consider the following function, which is almost the well-known quicksort function with the difference that it does not sort anything, but blows up the input-list:

```
nosort xs = case xs Nil nosort2
nosort2 x xs = append (nosort xs) (Cons x (nosort xs))

append xs ys = case xs ys (append2 ys)
append2 ys x xs = Cons x (append xs ys)
```

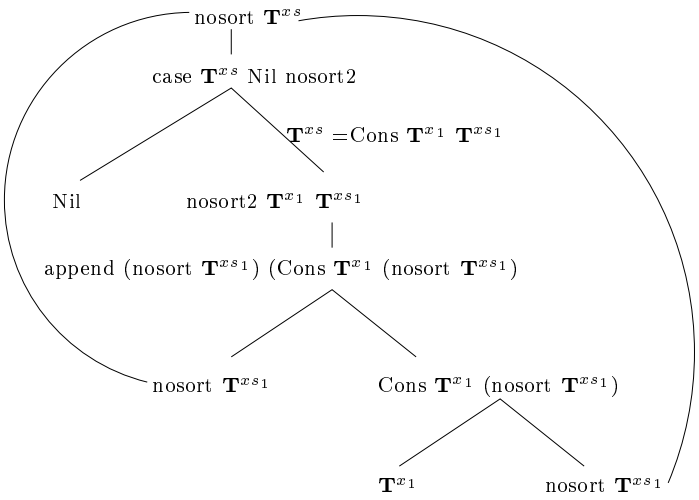


Figure 2.8: A closed tableau for `nosort`. Nf-termination of `append` has been used.

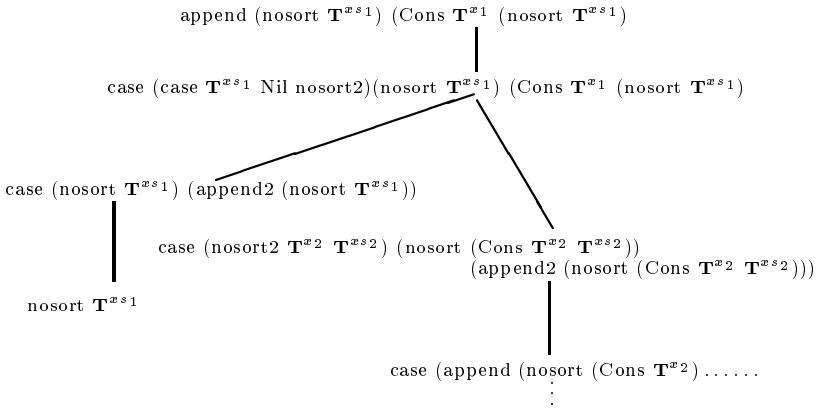


Figure 2.9: Parts of the tableau for `nosort`, where nf-termination of `append` is not used.

A dependency analysis will show that `append` depends in no way on `nosort`. So we can start by building a tableau for (`append` $\mathbf{T}^{xs}$ $\mathbf{T}^{ys}$). This tableau is easy to close. Then we can try to close a tableau for (`nosort` $\mathbf{T}^{xs}$), where we use nf-termination of `append` for a context-skipping. Such a tableau is given in figure 2.8. As can be seen this is a easy to find proof. Now let us see what would have happened if we did not make the dependency analysis and started with the termination analysis of `nosort`. We then could not branch with the rule of context-skipping at the `append`-node but had to proceed with δ-reductions. We ended up in creating an infinite tableau of figure 2.9, being forced to evaluate recursive calls to `nosort` again and again.

The reason that we could not close the tableau in figure 2.9 is that the recursive call of the function `nosort` does not appear on top-level in a leaf. Therefore, we have to reduce applications of `nosort` again and again in the tableau, without being able to perform path-analysis. We can try to overcome this deficiency introducing a new rule:

Approximated context-skipping

The problem that occured in example 5 was that the recursive call of a function appeared inside of a case construct. What we would have liked is to separate the proof for termination of the expression to be cased and the case-expression. We introduce a further expansion rule, the rule of approximated context skipping:

> Suppose we have in a given tableau a leaf marked with $t = C[e]$.
>
> An approximating context-skipping of this leaf is performed by extending it with the 2 new leaves:
>
> - $C[\mathbf{T}^c]$, with $\mathbf{T}^c$ a new abstract variable
> - e

Correctness of this rule is pretty obvious. We approximate an subexpression with a new abstract variable. We may do this as long as we can assure that the approximated expression is nf-terminating. The proof for the nf-terminating expression is not made in a separate tableau but in the same tableau by means of the second new leaf. This rule is of course only sensible, when the new leaf e will become a recursive node, such that we can close the tableau at this leaf. Therefore, we have to reconsider the correctness of path analysis again. Fortunately the proof for correctness of path-analysis in the non-branching case is not effected by this new rule. In the case of overlapping path, we have the same problems with the approximation-edge in lemma 19 as before.

Let us now reconsider example 5: We will again assume that no analysis of `append` has been done before. Now we will use the new rule introduced above to generate a preclosed tableau. The tableau is given in figure 2.10.

We get several recursive nodes. The ones on the left all refer to the root node and can be closed using the simple ordering on the length of lists. The recursive node on the right is the recursive node which proves termination of `append`. The paths are overlapping and there is an approximation-edge on a recursive path. Therefore we cannot apply lemma 19.

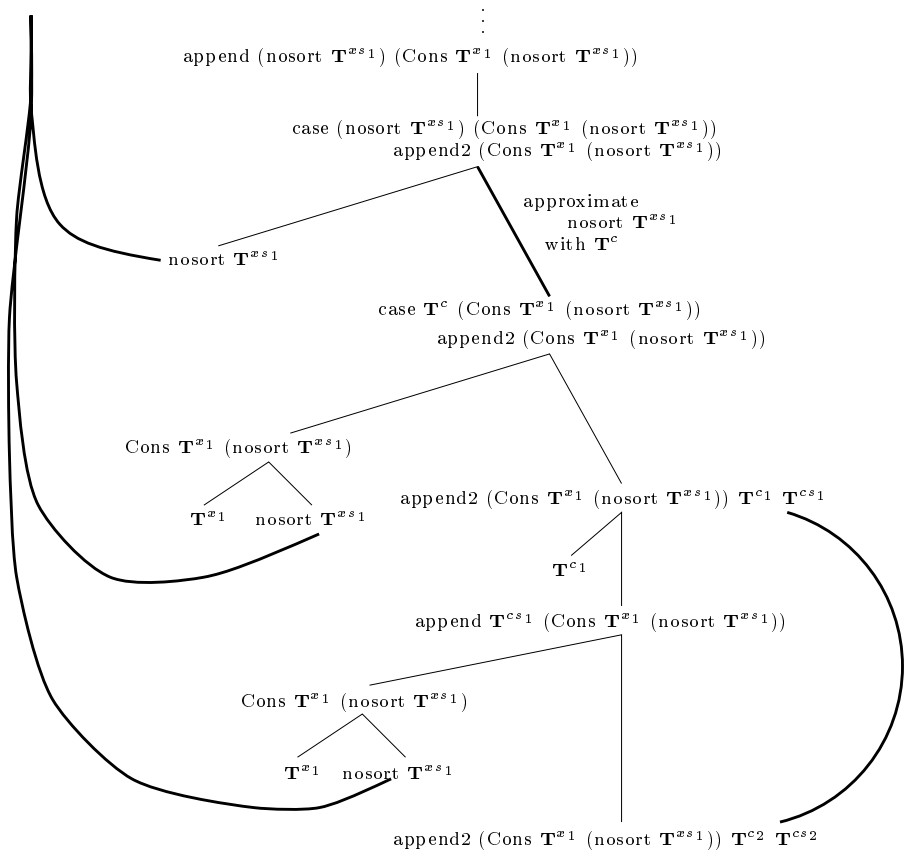


Figure 2.10: Expanding the tableau for `nosort` using approximated context-skipping

2.4 Termination for $\perp$, Infinite Arguments and Lazy-Termination

We already provided an example, where we applied a function to an argument without normal form. This was in the example of the `k` combinator. We can cultivate this a bit more. In a lazy language the question of termination is more than just a question of whether a function yields a normal-form if applied to normal-forms, i.e. it is nf-terminating. It can be of the form: does the function yield at least a head normal-form (or some other form) if it is applied to arguments in some certain form. So we might be interested in the form the function shall deliver and in the form the arguments are required to have. This corresponds to the so called context information in strictness analysis.

2.4.1 Forms for Arguments

As we have seen, we do not need special abstract values that represent certain forms for the argument, but can do with a worst representative of this form. So a representative for expressions which lack a head normal-form is the super-combinator `bot` defined by `bot = bot`.

If we want to know whether a function `f` also terminates for an argument that does not have any normal-form, we try to close the tableau with `(f bot)` at the root.

In the same way we can find representatives for certain forms of lists as they are proposed in [Wad87]. The best defined list are those having a 'finite' normal-form. These are represented by the abstract variable $\mathbf{T}^a$.

Further abstract values that can be defined are:

```
inflist x = Cons x (inflist x)
infbotlist = inflist bot
conshnf = Cons bot bot
map f xs = case xs Nil (consmap f)
resultbot x = bot
```

```
inftoplist = inflist T^b
botelem = map resultbot T^a
hnf = if T conshnf Nil
consmap f x xs = Cons (f x) (map f xs)
```

To get more subtle information of the termination behavior of a list function, a closed tableau with an application of `f` to one of the abstract expressions above has to be derived.

Note that we use free variables on the right hand side of a function definition. We have to assure that we will not use this abstract variables in any other part of the analysis.

Let us prove that the recursive function `length` terminates for all finite lists. The function `length` is defined by:

```
length xs = case xs 0 conslength
conslength x xs = + 1 (length xs)
```

We can deduce the tableau given in figure 2.11.

The tableau can be closed by using again the simple ordering on the size of normal-forms which fulfills the condition: (Cons $\mathbf{T}^{a_1}$ $\mathbf{T}^{a_2}$)>$\mathbf{T}^{a_2}$. The astute reader will have noticed that in this example we do not get an ordering constraint on the arguments of a super-combinator but the context around this argument is a complex expression.

2.4.2 Forms for the Result

Up to now we were only interested whether a functions yielded a result that was a normal-form of a non functional type. But we can get more with termination tableaux. All what we need are dummy evaluator functions, which do nothing

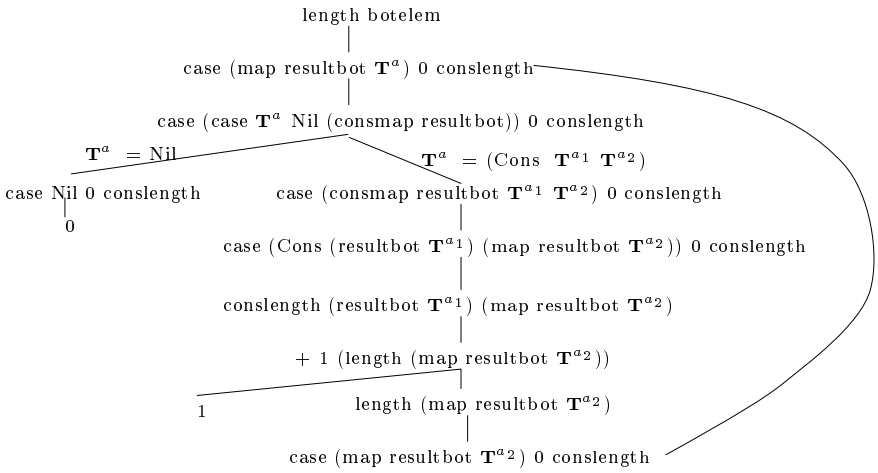


Figure 2.11: A tableau for `length` applied to lists containing undefined elements.

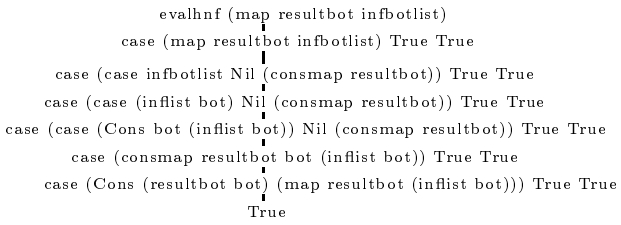


Figure 2.12: A closed tableau for `evalhnf (map resultbot infbotlist)`

more than to evaluate their argument to a certain form and then result with `True` if the argument has this form, e.g. we can define the following evaluators for lists:

```
evalhnf xs = case xs True consevalhnf
consevalhnf x xs = True
evalfinitespine xs = case xs True consevalfspine
consevalfspine x xs = evalfinitespine xs
```

An example of how such termination information can be proved by termination tableaux is the tableau in figure 2.12. It proves that `map` terminates with a head normal-form whenever it is applied to an infinite list that entails undefined elements:

2.4.3 Lazy-Termination

Up to now we always have proved that some abstract expression is nf-terminating or is at least in some subexpressions nf-terminating. But there is a certain class of expression which are not nf-terminating and will not run into an infinite loop; i.e. which will not have any parts that will semantically be $\perp$. A simple example of such an expression is a list which is potentially infinite, but where the

spine can always be reduced to a head-normal form. We have introduced an abstract expression representing such lists with `inftoplist`. `inftoplist` is not nf-terminating, but can be evaluated to a list of a arbitrary length. A speculative compiler might try to reduce such a list as `inftoplist` up to its nth element. It would not risk to run into an infinite loop. It seems to be quite counterintuitive to reduce a proof of a potentially infinite data-object to a proof of nf-termination, but this can be achieved.

First of all let us define the notion of lazy-termination:

Definition 21. An expression is (fully) lazy-terminating if it can be reduced to a form $(\mathbf{c}\ e_1 \ldots e_n)$ and e_i is lazy-terminating, $i = 1, \ldots, n$.

Note, that a data-object which is nf-terminating is also lazy-terminating.

We can refine the notion of lazy-termination by restricting it for certain arguments of constructors in a recursive algebraic type, e.g. for lists we could define the notion of lazy-spine-termination by:

Definition 22. An expression of list-type is lazy-spine-terminating if

- it can be reduced to `Nil` or
- to (`Cons` e es) and es is lazy-spine-terminating.

As can be seen the definition of refined lazy-termination notions are type dependent.

What we need are evaluators which correspond to the definition of lazy-termination. A evaluator, which forces the reduction of an potentially infinite structure will not do. Such an evaluator is a dual one to the definition of lazy-termination, e.g.:

```
lst xs = case xs True lst2
lst2 x xs = lst xs
```

Our calculus is not able to prove lazy-spine-termination with this evaluator, because `lst infbotlist` does not have a normal-form and we can only prove nf-termination.

We are looking for an evaluator which if applied to a potentially infinite list is nf-terminating. Therefore, we define an evaluator with two arguments: a (finite) number and a list. If we can proof that the spine of the list can be evaluated to the length of an arbitrary number, then we know that there is no position in the list that has no head normal-form. We can define such an evaluator in the following way: First of all we need an algebraic data type which denotes natural numbers. We define this data type in Haskell syntax: `data N = Z | S N`

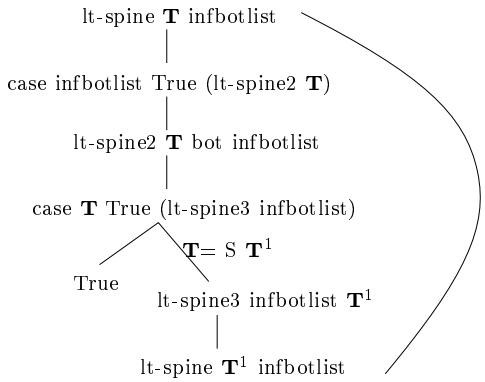


Figure 2.13: A proof of lazy-spine-termination of `infbotlist`

The evaluator for lazy-spine-terminating lists can now be defined as:

```
lt-spine n ls = case ls True (lt-spine2 n)
lt-spine2 n l ls = case n True (lt-spine3 ls)
lt-spine3 ls n = lt-spine n ls
```

Now it is quite easy to prove lazy-spine-termination of `infbotlist`. A proof is given in figure 2.13.

We have introduced the notion of refined lazy-termination by way of an example. We can give a general way how to define an arbitrary notion of lazy-termination for some arbitrary algebraic type. The definition of such a notion also induces the definition of an evaluator, such that we can use the tableau-calculus of nf-termination for lazy-termination proofs. Unfortunately we will get some slight problems with polymorphic types and therefore restrict ourselves to monomorphic types first:

Definition 23. Let an instance of an algebraic type be given: $(A\ \tau_1 \ldots \tau_k)$, such that $(A\ \tau_1 \ldots \tau_k)$ consists of the constructors $\mathbf{c}_1, \ldots, \mathbf{c}_n$ of types:
$\sigma_{1_i} \to \ldots \sigma_{m_i} \to (A\ \tau_1 \ldots \tau_k)$
A lazy-termination definition for $(A\ \tau_1 \ldots \tau_k)$ is of the form:

$LT(e)$:

$$e \to (\mathbf{c}_1\ e_{1_1} \ldots e_{m_1}) \text{ then } \{lt_j(e_j) | \text{for some } j \in \{1_1, \ldots, m_1\}\}$$
$$\vdots$$
$$e \to (\mathbf{c}_n\ e_{1_1} \ldots e_{m_1}) \text{ then } \{lt_j(e_j) \text{ for some } j \in \{1_n, \ldots, m_n\}\}$$

where $lt_j(e_j)$ shall denote that expression e_j is required to be lazy-terminating of kind lt_j.

Note, that this makes also the notion of head normal-form to a special form of

lazy-termination. The set of further lazy-termination conditions in the *then*-part is simply to be left empty.

We give the evaluator which goes along with a definition of lazy-termination and enables us to prove a certain notion of lazy-termination with the calculus of nf-termination.

Definition 24. Let for a monomorphic algebraic type $(A\ \tau_1 \ldots \tau_k)$ a lazy-termination definition LT given as in definition 23. An evaluator for such a definition is the function `lt` defined by:

```
lt x = lt' T x
lt' n x = case n True (lt'' x)
lt'' x n = case c  altfunctions
```

where
altfunctions are functions of the form:

`lti n x1i...xmi`
`= and [`$lt_j(e_j)$ for some $j \in \{1_i, \ldots, m_i\}$`]`

`and` is the usual logical conjunction function over all list elements. The list comprehension represents the set of further lazy-termination conditions in the corresponding definition of the constructor arguments.

The evaluator can now be used to prove lazy-termination within the calculus of nf-termination.

Eventually, let us consider lazy-termination of polymorphical typed algebraic expressions. There are abstract expressions, which can have concretisations of different type, e.g. `inftoplist` has concretisations of type (List α) for all types α. We cannot write an evaluator which forces the reduction of expressions of all possible types. Fortunately, the only way to introduce polymorphism in expressions of an abstract type is by way of an abstract variable **T** (parts of functional type left aside). For these polymorphical typed parts we can therefore show nf-termination, which our calculus can do without an evaluator function. So we can define an evaluator for (full) lazy-termination of lists by:

```
lt-list n ls = case ls True (lt-list2 n)
lt-list2 n l ls = case n True (lt-list3 l ls)
lt-list3 l ls n = Pair l (lt-list n ls)
```

Note that we have to pack the result of forcing evaluation of the spine with the head element into a pair. This avoids type problems. A proof for lazy-termination of `inftoplist` is given in figure 2.14.

Note that this evaluator is not able to prove lazy-termination of all kinds of lazy-terminating lists. Consider infinite lists with infinite list elements. For such abstract expressions another evaluator has to be given. It can be seen how the type of an abstract expression determines the evaluator it needs for proving lazy-termination.

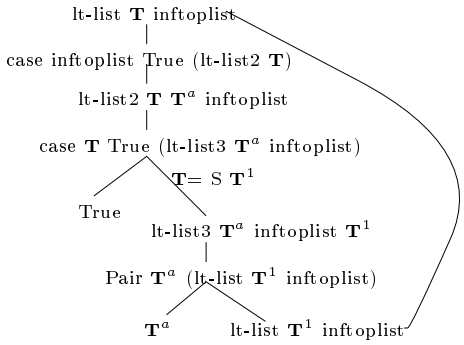


Figure 2.14: A proof of full lazy-termination of `inftoplist`

2.5 Basic Values

Up to this point we neglected basic values completely in our considerations. We could not extract any termination information for functions which rely in their termination behavior on basic values. The only presented solution had been to code basic values into an algebraic type. But there are situation where termination can be seen quite easily to depend on basic types.

Example 6. Consider the following function due to John Hughes:
`h x y = if (x == 0) (h y (y+1)) y`
`h` is nf-termination. We cannot prove this with tableau, because the resulting branch of the case gives us some information on the input values, which we neglected.

If we have a case expression which performs a *case* on an abstract expression where every abstract variable can only be substituted by an expressions of basic type, (and otherwise we do not get a valid concretisation), then the resulting branching will give us at least some information on the abstract variables. Let us try to create a closed tableau for the function `h` in example 6. This will not lead us very far with our calculus. As soon as we perform abstract reductions on the expressions `(`$\mathbf{T}^a$ `== 0)` and `(`$\mathbf{T}^b$ `+ 1)` we will get the same node as the root node (modulo renaming of abstract variables) and will not gain any information. A better thing to do is, not to perform abstract reduction on these expressions, but to leave them untouched and to keep track in the branching of the case of the value these expressions are assumed to have. We can label the edge of a branching with the condition of the branching. This information can later be used to exclude some branches. A tableau which proves termination of `h` is given in figure 2.15.

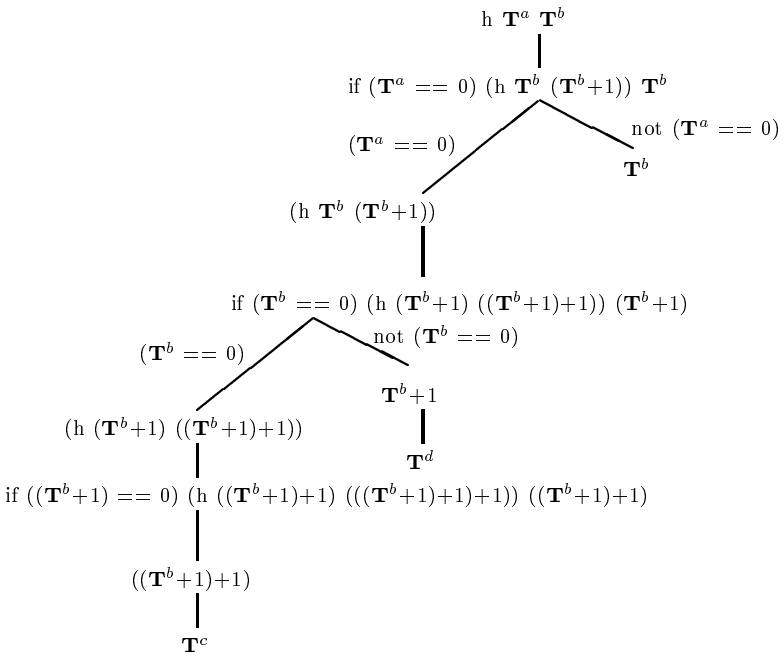


Figure 2.15: A tableau which respects conditionals on basic values.

So what is needed is a deductive component which is able to reason on the built-in functions on basic types. We assume a rather simple deductive component which is limited to conditional expressions which can be represented flat, i.e. which do not involve recursive functions. Such a deductive component could be modeled by a term-rewriting system.

An even simpler approach with only some limited effect, is to include a special rule for the built-in function ==. Whenever we have a leaf marked with:

$$C[\textbf{case } (\mathbf{T}\texttt{==}e)\ e_1\ e_2]$$

we can extend the tableau with the two new leaves:

- $C[e_1][\mathbf{T} \mapsto e]$
- $C[e_2]$

2.6 Functions

Up to now we did not provide any means to represent functions by an abstract value. This was basically for simplicity reasons. There will no *case* be performed on abstract values representing functions. This means that we cannot deconstruct an abstract value for a function into its subparts, the way we have done with

abstract values for algebraic types. This means that an abstract value for a function cannot be used for proving termination, because our termination proofs depend on a Noetherian ordering of the arguments in recursive calls. We cannot prove such orderings for functions. Nevertheless, abstract values for functions can enhance our calculus a bit and, in fact, we already gave an example of an abstract function. When proving termination of `foldr` we simply assumed some property on the function argument and used this for extending the tableau. And this is in fact what we will do with functions. We simply have to generalize the notion of a concretisation:

A concretisation of $t \in \Lambda_c^\#$ is a substitution
$\gamma : \mathcal{T} \to \{e | e \in \Lambda_c \text{ and } e \text{ is nf-terminating}\}$, such that $\gamma(t) \in \Lambda_c$.

There is a small pitfall in this definition. The notion of nf-termination is overloaded for functions with two meanings: it can mean that the function expression has a normal-form and it can mean that the function expression when applied to expressions in normal-form will yield a normal-form again. It is of course the latter meaning we will use for abstract variables of function type. The property of a function expression to have a normal-form is a rather useless one.

Now we have automaticallythree new or at least generalized ways to extend a tableau:

- **T**-application:
  If there is a leaf node $C[(\mathbf{T}^a\ \mathbf{T}^b)]$ in a given tableau, then append the new leaf $C[\mathbf{T}^c]$ to this leaf.
- context-skipping:
  If there is a leaf node $(\mathbf{T}^a\ e_1 \ldots e_n)$, then append the n new leaves $e_1, \ldots, e_n$.
- **T**-approximation:
  If there is a function expression in a leaf, which is known to be nf-terminating, then it may be approximated with a new abstract variable.

Abstract variables for functions have a further consequence for our calculus. They can even be used when closing a recursive path. This is already covered by the extension rules:

Example 7. Consider the following function which applies different functions to the elements of the list argument.

```
addnumber f ls = case ls Nil (addnumber2 (plus1 f))
addnumber2 f l ls = Cons (f l) (addnumber f ls)
plus1 f x = (f x) + 1
```

First of all we can easily prove that `plus1` is nf-terminating, i.e. (`plus1` $\mathbf{T}^a$) yields a nf-terminating function or in other words (`plus1` $\mathbf{T}^a\ \mathbf{T}^b$) is nf-terminating. Now we can derive the tableau for (`addnumber` $\mathbf{T}^a\ \mathbf{T}^b$) which is given in figure 2.16.

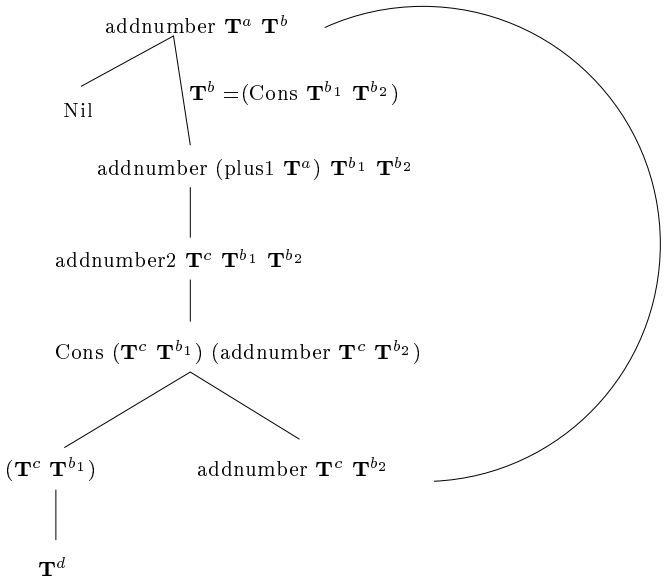


Figure 2.16: A proof for nf-termination of `addnumber`

The ordering constraint for the recursive path is:
$(\mathbf{T}^a, (\texttt{Cons}\ \mathbf{T}^{b_1}\ \mathbf{T}^{b_2})) > (\mathbf{T}^c, \mathbf{T}^{b_2})$.
This constraint can be fulfilled by the ordering on pairs which only depends on the length of the list in the second pair component.

We provided some means to represent certain functions by an abstract variable, namely nf-terminating functions. It arises the question whether this enables us to define any class of functions with a certain termination behavior by some abstract function. We will investigate this question stepwise on the type of functions.

2.6.1 Functions of basic types

Let us start with the simplest function type we can think of, i.e. functions of type $Num \rightarrow Num$. There are three different termination behaviors such functions may have:

- They may constantly yield a normal-form.
- They may constantly yield expressions which do not have a normal-form.
- They may be nf-terminating.

The combinatorial fourth possible function type are functions which yield a normal-form iff the argument does not have a normal-form. This type of functions is impossible. Otherwise this would contradict proposition 5.

We can easily provide representatives for the three types of function on *Num*→*Num*: *i*) n_1 x = **T** *ii*) n_2 x = bot *iii*) n_3 = **T**

2.6.2 Functions of algebraic arguments and basic results

Now let us complicate things a bit. We will now consider functions of types $(A\ \tau_1 \ldots \tau_n) \rightarrow Num$. As a representative for such functions we investigate functions of type $(List\ \alpha) \rightarrow Num$. There are infinite many different types of termination behaviors that can be defined for such functions; e.g. functions which yield a normal-form iff the nth element of the argument list has a normal-form.

All of these termination behaviors are of the form: the function yields a normal-form iff the argument can be evaluated to a certain degree. In section 2.4.2 we have developed evaluator functions which force the evaluation of their argument to a certain degree. Basically these evaluator functions can be used to define abstract functions on algebraic types. We only have to ensure that the evaluator function will not restrict our types too much. This can be done by resulting in some **T** whenever the evaluation was successful (in contrast to the boolean results we gave in section 2.4.2).

Now we can define the following abstract functions for lists:

- l_1 xs = **T**
- l_2 xs = bot
- l_3 = case xs **T** const
  const x xs = **T**
- l_4 xs =case xs **T** l4cons
  l4cons x xs = l_4 xs
- l_5 xs = **T**+(length xs)
- l_6 xs = **T** xs

The abstract function l_4 is in a way interesting. In its definition it can only represent functions of type $(List\ \alpha) \rightarrow Num$. But as a matter of fact we can use it as an representative of all functions of type $(List\ \alpha) \rightarrow \beta$, too. This is because l_4 can only yield an abstract normal form as result if a redex (**T**+e) becomes evaluated. This will result in a abstract variable **T** which is of any type β. The only problem can occur, when context skipping is applied to this redex. But fortunately, context skipping can only be applied to top-level expressions. So that we do not care about the type of this redex after a context skipping. Actually the (**T**+) was not necessary, because we provided the rule of **T**-approximation[4]. If in the expression (**T**+e) e is evaluated to some normal-form, say the number 5, then this can be approximated by some variable **T**. The result is the same as performing the abstract δ-rule: (**T**+e) →**T**).

[4] And in fact this is the first time we make use of this rule

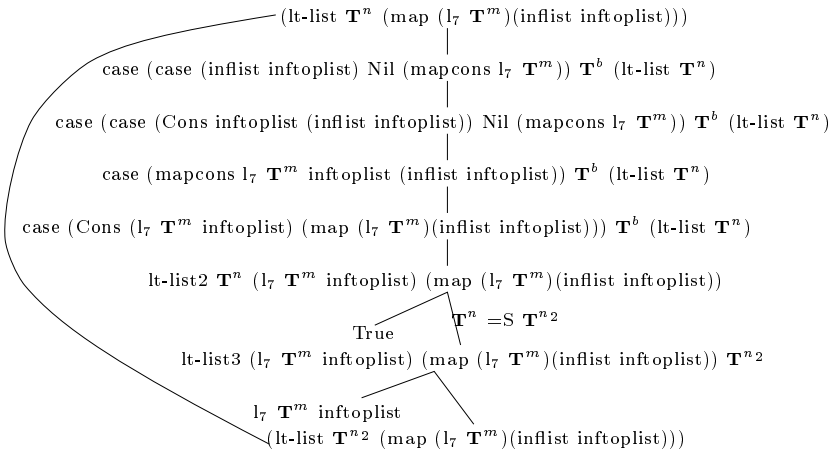


Figure 2.17: A proof for nf-termination involving abstract functions

In the same way we can adopt the evaluator functions for lazy-termination as an abstract function. So we can e.g. define the abstract function (l_7 **T**) in the following way:

- l_7 n xs = case xs **T** ($l_7$2 n)
  $l_7$2 n x xs = case n **T** ($l_7$3 x xs)
  $l_7$3 x xs n = Pair x (l_7 n xs)

l_7 represents all functions, which will yield a normal-form iff their argument is lazy-terminating in its spine and every list element of the argument has a normal-form. l_7 can also be used to represent functions which do not yield a tuple as result. Simply **T**-approximate the pair.

Example 8. Let us prove some example with the abstract functions we have introduced. We want to prove that

(lt-list $\mathbf{T}^n$ (map (l_7 $\mathbf{T}^m$)(inflist inftoplist)))

is nf-terminating.

This means that, if we take a lazy-spine-terminating list of lazy-spine-terminating lists, which have nf-terminating elements and apply a function which transforms such lists into normal-forms to all these elements, then we will get a spine lazy-terminating list with nf-terminating elements.

A prove is given in figure 2.17. At the leaf (l_7 $\mathbf{T}^m$ inftoplist) we can append basically the same tableau as the one in figure 2.16.

2.6.3 Functions of algebraic results

If we want to specify abstract functions which yield some algebraic type as result this is comparatively easy. In section 2.4.1 we have already seen that we can provide abstract values for any type of lists which can be evaluated to some

degree. In order to define abstract functions we simply can use these abstract values. An easy example is the following function which yields a lazy-terminating list, if its argument has a head normal-form:

```
headtolazy xs = case xs inftoplist headtolazy2
headtolazy2 x xs = inftoplist
```

Combining the results of the last two subsections we can specify (almost) any abstract function.

Chapter 3

Ordering Tableaux

3.1 Linear Orderings

In this section we will define the class of Noetherian orderings on Λ_c-expressions which can be used as a basis for termination proofs. Such orderings on Λ_c are defined through a partial function τ which maps Λ_c-expressions to $\mathbb{N}$.

Definition 25. A partial function $\tau : \Lambda_c \to \mathbb{N}$ can be recursively defined by as follows:

- $\tau(e) = n_0^c + n_1^c\tau(e_1) + \ldots + n_m^c\tau(e_m), m_i^c \in \mathbb{N}$
  if e has the head normal-form $(\mathbf{c}\ e_1 \ldots e_m)$ and $\mathbf{c}$ is a constructor of arity greater or equal m.
- $\tau(e) = n_0^f + n_1^f\tau(e_1) + \ldots + n_m^f\tau(e_m), n_i \in \mathbb{N}$
  if e has the head normal-form $(f\ e_1 \ldots e_m)$ and f is a supercombinator of arity greater m.

We will call τ a termination function and demand that for every constructor there are defining equalities of τ. This includes all cases of partial applications of constructors. If the calculation of $\tau(e)$ does not terminate, then $\tau(e)$ is undefined. As a further restriction we will only allow such functions τ that $\tau(e) > 0$, if $\tau(e)$ is defined.
The multiplication is to be understood as bottom-avoiding, i.e. $0\tau(e_i) = 0$ even if $\tau(e_i)$ is undefined.

There are several reasons for $\tau(e)$ being undefined:

- e does not have a head normal-form.

- e is a potentially infinite object (i.e. it does not have a normal-form).
- e has a head normal-form, but no rule has been specified for its supercombinator in the definition of τ.

Note that the two clauses in the definition of termination functions can be subsumed in one clause. We distinguished between expressions of function type and expressions of non-functional type for reasons of clarity only.

3.2 Ordering Tableaux

In this section we will define an abstract reduction calculus in terms of ordering tableaux. Ordering tableaux are defined similarly to termination tableaux and operate on the same source language.

A tableau is a finite tree whose nodes are marked with ordering propositions on abstract expressions.

First we define what an ordering proposition looks like:

Definition 26. A ordering proposition is an expression that can be generated by the following grammar:

$$\begin{array}{lll} \mathcal{S} & ::= & \mathcal{LR} \geq \mathcal{LR} \\ \mathcal{LR} & ::= & (I\!N, \{\mathcal{E}_1, \ldots, \mathcal{E}_n\}), \\ \mathcal{E} & ::= & (I\!N, \mathcal{L}), \end{array}$$

where $\mathcal{L}$ may be any $\Lambda_c^{\#}$-expression.

Definition 27. An ordering proposition:

$$(n_0^l, \{(n_1^l, e_1^l) \ldots (n_n^l, e_n^l)\}) \geq (n_0^r, \{(n_1^r, e_1^r) \ldots (n_m^r, e_m^r)\})$$

is valid for a termination function τ iff:

$$n_0^l + \sum_{i=1}^{n} n_i^l \tau(\gamma(e_i^l)) \geq n_0^r + \sum_{i=1}^{n} n_i^r \tau(\gamma(e_i^r))$$

for all concretisations γ, such that both sides of the inequality are defined.

In this definition we restrict ourselves only to some sort of partial correctness. If the function τ is undefined for parts of the ordering proposition, then this proposition is considered to be valid. That means that we may prove some statements of the form: if the expression *(loop x)* terminates, then $\tau(x) > \tau(loop\ x)$. For a proof of total correctness we then would have to prove that *(loop x)* terminates.

Our calculus will basically only prove partial correctness of ordering propositions, but can be extended to prove total correctness and then will be capable to subsume termination tableaux completely.

For the case that we want to prove total correctness of ordering propositions we will have to use another notion of validity:

Definition 28. An ordering proposition is totally valid iff it is valid and both sides of the inequality are defined for all concretisations.

The notion of validity of ordering proposition gives us a semantics for nodes in a tableau. For this semantics we can specify what it means for a tableau to be sound.

Definition 29. An ordering tableau is sound if for all nodes n we have: Let n be marked with lr. If n has the direct sons $n_1, \ldots, n_k$ such that n_i is marked with lr_i then:
if for all $1 \leq i \leq k : lr_i$ is valid then lr is valid.

Now we develop a calculus which derives new sound tableaux with the same root node from a sound tableau. The new tableau will always originate from an old one by extending a path with new leaves.

We will define deduction rules for ordering tableaux and prove their soundness.

3.2.1 Normalizing ordering propositions

The first thing will be to find a uniform representation for ordering propositions. We define a normal-form for ordering propositions:

Definition 30. An ordering proposition

$$(n_0^l, \{(n_1^l, e_1^l), \ldots, (n_n^l, e_n^l)\}) \geq (n_0^r, \{(n_1^r, e_1^r), \ldots, (n_m^r, e_m^r)\})$$

is in normal-form iff

- there is no e_j^k which is in head normal-form, such that τ can be applied directly to e_j^k.
- there are no two different expressions e_j^k, e_l^l in abstract normal-form such that they are syntactically equal.
- $n_0^l = 0$ or $n_0^r = 0$.

It is easy to calculate a normal-form of an ordering proposition:

i) calculate recursively all tuples of the form $(n, \mathbf{c}\ e_1 \dots e_m)$. Replace them with: $(n * n_1, e_1), \dots, (n * n_m, e_m)$ and increase n_0^k by $(n * n_0)$ ($k = l$ if $(n, \mathbf{c}\ e_1 \dots e_m)$ was on the left, $k = r$ otherwise),
where $\tau(\mathbf{c}\ e_1 \dots e_m)$ is defined as $n_0 + \sum_{i=1}^{m} n_i \tau(e_i)$

ii) delete all tuples $(0, e)$

iii) cancel out n_0^l or n_0^r, i.e. if $n_0^l > n_0^r$ then replace n_0^l by $n_0^l - n_0^r$ and n_0^r by 0 or the other way round.

iv) for all abstract normal-forms e unite all pairs (n, e) to one pair by canceling out in the same way as in iii).

Note that after step i) there are only abstract variables left as expressions in abstract normal-form.

Lemma 31. *The procedure above preserves total validity of ordering propositions.*

Proof. We only cancel out natural numbers or abstract normal-forms. Abstract normal-forms always are defined under τ. Thus, an ordering proposition is totally valid iff its normal-form is totally valid.

If we allow to cancel out arbitrary expressions, i.e. expressions not in abstract normal-form there may be ordering propositions which have a totally valid normal-form but are not totally valid themselves.

Next we specify expansion rules, which allow to extend paths of an ordering tableau.

3.2.2 $\delta-$rules

We can adopt the same rules we applied to termination tableau, i.e. we can reduce any redex at a leaf and we can branch, if there is a *case* on an abstract variable. After having performed a reduction, we will calculate the normal-form of the resulting new leaf node.

The next rules are very general approximations. These rules make the calculus to a very high degree non-deterministic. Strategies when to apply these rules are given later on.

3.2.3 Deletion and Addition

There are several approximations that can be made on an ordering proposition:

- Delete left:
  - replace n_0^l with $n_0^{l\prime}$, where $0 \leq n_0^{l\prime} \leq n_0^l$.
  - replace any n_i^l with $n_i^{l\prime}$, where $0 < n_i^{l\prime} \leq n_i^l$.
  - delete any pair (n_i^l, e_i^l) where e_i^l is in abstract normal-form.
- Add right:
  - replace n_0^r with $n_0^{r\prime}$, where $n_0^r \leq n_0^{r\prime}$.
  - replace any n_i^r with $n_r^{l\prime}$, where $n_i^r \leq n_i^{r\prime}$.
  - add any pair (n_i^r, e_i^r) where e_i^r is in abstract normal-form.
- Delete $\mathbf{T}$:
  - if there is a pair $(n, \mathbf{T})$ on one side of the proposition, then delete this pair and increase n_0 on this side by n.

These approximations are sound in the way that:
if the approximated node is totally valid then the original node is totally valid.

3.2.4 Splitting

Suppose in a given tableau there is a leaf marked with:

$$(n_0^l, \{(n_1^l, e_1^l), \ldots, (n_n^l, e_n^l)\}) \geq (n_0^r, \{(n_1^r, e_1^r), \ldots, (n_m^r, e_m^r)\})$$

Then the tableau may be expanded at this leaf by appending k new leaves

$$(n_0^{l_i}, \{(n_1^{l_i}, e_1^{l_i}) \ldots (n_n^{l_i}, e_n^{l_i})\}) \geq (n_0^{r_i}, \{(n_1^{r_i}, e_1^{r_i}) \ldots (n_m^{r_i}, e_m^{r_i})\})$$

$i = 1, \ldots, k$, iff:

- $\sum_{i=1}^k n_j^{l_i} = n_j^l$, for all $j = 1, \ldots, n$ and
- $\sum_{i=1}^k n_j^{r_i} = n_j^r$, for all $j = 1, \ldots, m$

3.2.5 Tautology nodes

There are nodes which are trivially valid. With the approximations introduced before we can identify tautology nodes with the node marked with:
$(0, \{\}) \geq (0, \{\})$.

3.2.6 Approximation

We have already seen some approximation rules, which were more or less of syntactical nature and can be integrated in a fully automatic prover. In this subsection we will introduce a highly non-deterministic approximation rule which is therefor only suitable for an interactive mode of the calculus:
A leaf $l \geq r$ can be approximated with:

- $l \geq r'$, iff $r' \geq r$ is valid.
- $l' \geq r$, iff $l \geq l$‘ is valid.

An example where this rule will help to find a termination proof is given in the appendix.

3.2.7 Induction Step

We arrived at the rule where a close examination of a path allows to make an induction step. Such an induction step marks a leaf which constitutes a hypotheses which has been on the path from the root to this leaf before.

Definition 32. A leaf node $e' = C[e_1, \ldots, e_n]$ in a tableau is called a recursive node, if there is an ancestor node $e = C[\mathbf{T}^{x_1}, \ldots, \mathbf{T}^{x_n}]$, such that there is a substitution σ with $\sigma(e) = e'$.

(e, e') is called a recursive pair, σ the corresponding substitution and the path from e to e' the recursive path.

Note that the context $C[.]$ describes the frame of some ordering proposition rather than only an abstract expression. We allow that the ordering of set-elements may be changed.

We define the class of tableaux which are candidates for an ordering proof:

Definition 33. A tableau where every leaf is either a tautology node or a recursive node, is called *preclosed*.

3.2.8 Partial Correctness

This section is devoted to the partial correctness of preclosed tableaux. Basically we want to show that the root of a preclosed tableau is valid. This means that every concretisation for which the ordering proposition is defined, makes the proposition true. In order to prove partial correctness we need some measure

on concretisations of ordering propositions. This measure will be based on the progress we have made in a proof. Therefore we will firstly define edges which constitute a progress in the proof of an ordering proposition.

Definition 34. Edges in a tableau which produce an expression in head normal-form $(n, (\mathbf{c}\ e_1 \ldots e_n))$ such that there is a defining rule for the termination function τ in the case $\tau(\mathbf{c}\ e_1 \ldots e_n)$ are called progress edges.

The process of normalizing will calculate a bit of τ for progress edges. How much of the expressions have to be still evaluated in order to evaluate τ will be the main measure in the proof of the theorem for partial correctness. The proof is similar to the one of the main theorem in [SS96], which is a proof on partial correctness only, too.

Theorem 35. *The root of a preclosed tableau where every recursive path encloses a progress edge is valid.*

Proof. We can broaden our notion of concretisations for this theorem. We will use the notion of liberal instances. A liberal instance of an abstract expression (and also of an ordering proposition) is a substitution σ which maps abstract variables to arbitrary expressions such that: $\sigma(e) \in \Lambda^c$.

The difference between concretisations and liberal instances is that abstract variables are no longer restricted to represent expressions which have a normal-form.

For the proof of the theorem, let us assume it is false: There is a concretisation of the root for which the ordering proposition is defined and false. Then there are recursive nodes which have concretisations for which the ordering proposition is defined and false. This means there are liberal instances of these recursive nodes, which make the ordering proposition false. For every liberal instance of an ordering proposition, for which the proposition is defined, the number of head normal-forms on top-level which have to be calculated in order to calculate τ is finite (otherwise there was some essential part in the ordering proposition where τ was undefined). Now choose a liberal instance of one of the recursive nodes such that:

- the ordering proposition of the recursive node is defined for this liberal instance.
- the ordering proposition is false.
- the number of top-level head normal-forms which have to be produced to calculate the ordering proposition is minimal.

- and as a second minimization criterion the upper node of the recursive pair has a minimal distance to the root node.

For a recursive pair (e, e') every liberal instance of e' is also a liberal instance of e. The liberal instance of e we have thus obtained can now be transformed along the tableau from the node e down so that we get another liberal instance of a recursive node, for which the ordering proposition of the node is false. This new liberal instance of a recursive node will have less or equal the number of top-level head normal-forms which have to be produced than the liberal instance which was originally chosen. This number must not have decreased, otherwise this contradicted our choice of the liberal instance to be minimal in this respect. So this number has to have stayed constant. This means that we did not transform our original liberal instance along a complete recursive path, or otherwise along the progress node the number would have decreased. This means that the upper node of the new recursive node is closer to the root than the one of the originally chosen liberal instance. This contradicts our second minimization criterion. The assumption was false and the theorem is true.

We have not been very rigorous with the description of transforming a liberal instance along a tableau. There is a small pitfall in this process. The liberal instance may be undefined at some parts, where the tableau requires it to be defined (because a case is evaluated at this position). If this is a position which is necessary for evaluating the ordering proposition then this cannot be the case, because we have chosen liberal instances for which the ordering proposition is defined. If this is some other redex, then the tableau reduces a redex which is not really needed for the calculation of the ordering proposition. We can replace this undefined subexpression in our liberal instance which has no head normal-form, by some arbitrary expression, which does have a head normal-form. This will not effect the number of still to be calculated head normal-forms which are required to calculate the ordering proposition.

The proof is even a bit more general and we can extend the theorem from concretisations of the root to liberal instances of the root.

Corollary 36. *Every liberal instance of the root in a preclosed tableau where every recursive path encloses a progress edge is valid.*

Example 9. We are by now able to prove the ordering proposition which is needed to close the termination tableau of `quicksort`. We have to show that (`less` x xs) is smaller than (`Cons` x xs) (and the equivalent proposition for `greater`). We choose as termination function the function which maps lists to their length:

$$\tau(\texttt{Nil}) = 1$$
$$\tau(\texttt{Cons}\ x\ xs) = 1 + 0\tau(x) + 1\tau(xs)$$

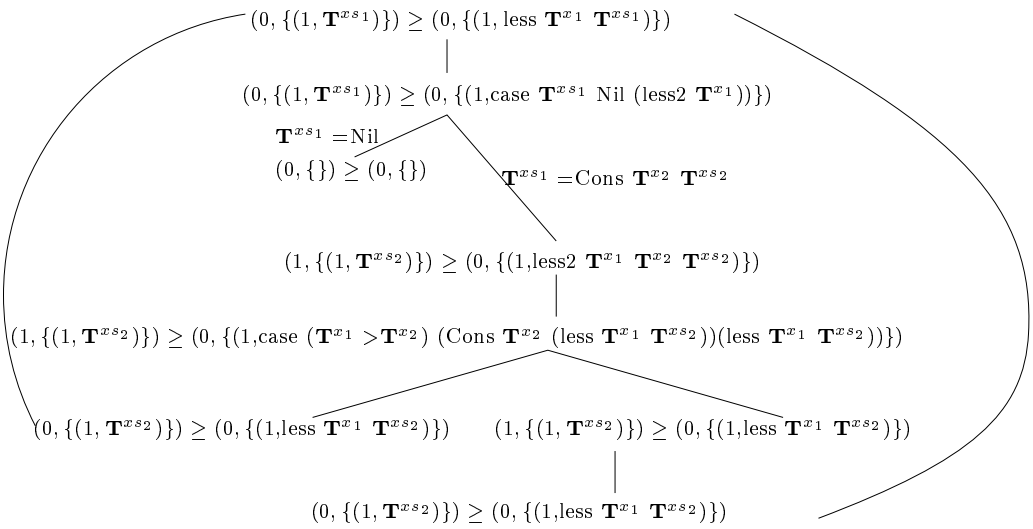


Figure 3.1: A tableau proving an ordering relation for `less`

$\tau(\texttt{Cons}\ x\ ) = 1 + 0\tau(x)$
$\tau(\texttt{Cons}) = 1$

We can derive the simple preclosed tableau in figure 3.1. On the two recursive paths there are progress nodes. Therefore, the original ordering proposition is valid. If we want to prove this proposition to be totally valid then we have to show nf-termination of (`less` x xs), which in fact can be easily done by termination tableau.

3.2.9 Strategies

As can be seen the approximation rules are only performed, if one of the resulting new leaves is a recursive node or a tautology node. Therefore an implementation will combine the path analyzing for the induction with the approximations in one procedure, i.e. there will be a test, whether a approximation can be performed such that one of the resulting nodes is a recursive node.

3.2.10 Functions

We introduced termination functions τ which may be defined for expressions of function type and are able to prove ordering propositions on functions. It is hard to think of an example, where this can be of any help in the context of termination tableau. Let us try to construct such an example. A recursive pair is required, such that an argument of function type appears somewhere in the corresponding tuples of the recursive pair. The only means to express a function

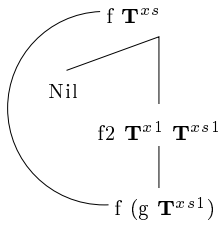


Figure 3.2: A termination tableau for `f`.

is a partially applied supercombinator. So we have to prove somehow that a not saturated function expression is smaller under some termination function than an abstract variable. This abstract variable will represent functions, i.e. it cannot be deconstructed by a case expressions. This makes it impossible to find a useful ordering proposition depending on functions in termination tableau.

We can only think of the following rather obscure example where we abandon well-typedness of Λ_c:

Example 10. We define two supercombinators. They cannot be typed!

```
f x = case x Nil f2
f2 x xs = f (g x)

g x y = g x y
```

Now we try to built a termination tableau for `f`. The resulting tableau is given in figure 3.2.

The following termination function can close the recursive path in the termination tableau for `f`:

$$\begin{aligned}
&\tau(\texttt{Nil}) = 1\\
&\tau(\texttt{Cons } x\ xs) = 1 + 0\tau(x) + 1\tau(xs)\\
&\tau(\texttt{Cons } x\ ) = 1 + 0\tau(x)\\
&\tau(\texttt{Cons}) = 1\\
&\tau(\texttt{g } xs) = 1 + \tau(xs)
\end{aligned}$$

The proof of the required ordering proposition which is required for closing the tableau for `f` is easy to show to be valid.

3.2.11 Total Correctness

Partial correctness gives only conditional proofs. It proves an ordering proposition to be true for concretisations for which the proposition is defined. To show

total correctness we have to prove that an ordering proposition is defined for all of its concretisations. A simple way to assure this is to prove that every abstract expression occuring in this proposition is nf-terminating. This means that for a termination proof of a function `f` we might have to prove some ordering proposition about another function `g`, which includes proving termination of `g` which requires again some ordering proposition etc.

Proving nf-termination of all abstract expressions occuring in an ordering proposition is a bit too strict for proving total validity. A termination function τ can be defined for expressions which are not nf-terminating.

We can simply adapt the path analysis for termination tableau to ordering tableau:

Definition 37. A leaf node $e' = C[e_1, \ldots, e_n]$ in an ordering tableau is called a t-recursive node, if all e_i are abstract normal-forms and there is an ancestor node $e = C[\mathbf{T}^{x_1}, \ldots, \mathbf{T}^{x_n}]$, such that there is a substitution σ with $\sigma(e) = e'$.

(e, e') is called a t-recursive pair and σ the corresponding substitution.

The difference to recursive nodes in ordering tableaux is that the corresponding substitution may only substitute abstract variables with abstract normal-forms as it had been the case in termination tableau. This ensures that every concretisation of a t-recursive node is a concretisation of the upper node in the t-recursive pair. So we can apply lemma 9 again. Now we can close an ordering tableau if we find an ordering which ensures that concretisations of t-recursive paths get decreased.[1]

3.3 Ordering and Termination Tableaux

3.3.1 Ordering propositions for use in termination proofs

Now let us return to the point where we wanted to use ordering propositions in the first place. Assume that in a termination tableau there is a path from a node

$$e = C[\mathbf{T}^{a_1} \ldots \mathbf{T}^{a_n}]$$

to a node

$$e' = C[e_1 \ldots e_n]$$

where the e_i are not necessarily in abstract normal-form.

[1] This means also that we get the same problems with overlapping paths as we had in termination tableau.

This means in the first place that we cannot assure that every concretisation of e' is a concretisation of e, i.e. lemma 9 does not hold anymore. Thus we have to ensure that every concretisation of e' is also a concretisation of e. A simple way to ensure this is to prove nf-termination of all e_i.

The next thing to do is to test whether for some fixed Noetherian ordering the path from e to e' decreases a concretisation of e. As long as the ordering is based on a linear termination function τ we can try to prove the necessary ordering propositions with ordering tableaux.

3.3.2 Termination proofs with ordering tableaux

The astute reader will have noticed that ordering tableaux can subsume termination tableau, if we use them for proving total correctness. A proof for total correctness includes a termination proof of some sort. If we want to prove that an abstract expression e is nf-terminating then we can try to prove total correctness of the following ordering proposition:

$$(0, \{(1, e)\}) \geq (0, \{(1, e)\})$$

Note that this is not a tautology node because the sets on both sides are not empty.

We have to specify the ordering the termination function τ for which we want to close this ordering tableaux. It is our aim to evaluate every part of the concretisations of e. Therefor we have to ensure that τ evaluates every part of the expression it is applied to. This means that τ has to be of the form:

$$\tau(\mathbf{c}\ e_1 \ldots e_n) = 1 + \sum_{i=1}^{n} 1 * \tau(e_i)$$

for every constructor $\mathbf{c}$ or partial application of a supercombinator.

As can be seen τ is a form of evaluator function like the ones in section 2.4. So we can say that proving termination with ordering tableaux uses a termination function which resembles a compiled evaluator function.

3.3.3 Ordering tableaux and ordering tableaux

If we use ordering tableaux to prove nf-termination as proposed in the last subsection or try to prove total correctness of some ordering proposition then we night need information about an ordering relation in order to close the tableau at a recursive but not t-recursive leaf. In such a situation we will need ordering tableaux to close an ordering tableau.

Chapter 4

Conclusion

We have proposed a calculus which performs termination proofs for a lazily evaluated functional programming language. The calculus is based on abstract reduction and formulated in terms of termination tableaux. Termination tableaux generate constraints for an ordering. In order to complete a proof, it can be tested whether some fixed ordering fulfills these constraints or a calculus which generates such orderings can be plugged in.

The presented method is able to handle higher order and polymorphic functions and to give detailed information for the termination behavior of applications to arguments which do not have a normal form.

The method proves termination of recursive functions that have calls to another recursive function g in their recursive calls by proving an ordering proposition about the function in the recursive call. Such proofs can be made by ordering-tableaux, which can also handle higher order and polymorphic functions.

An experimental implementation of the calculus exists which besides other non-trivial examples is able to derive all closed termination tableaux given in this paper. We hope to extend this implementation to a stable and powerful verification tool in the near future.

I would like to thank Marko Schütz for lending me his ear whenever I needed some feedback, Hubert Kick for his work on the implementation and Manfred Schmidt-Schauß for his helpful comments on several draft versions.

Appendix A

Example Proofs

A.1 First Order Functions

First we will see how our calculus compares to a termination calculus solely created for first order strict languages. In the following we will give proofs for the examples given in [Wal94].

All examples will depend on lists. We will for a fixed termination function throughout this section, which ignores lists elements and orders lists simply by their length:

$$\tau(\texttt{Nil}) = 1$$
$$\tau(\texttt{Cons}\ x\ xs) = 1 + 0\tau(x) + 1\tau(xs)$$
$$\tau(\texttt{Cons}\ x\ ) = 1 + 0\tau(x)$$
$$\tau(\texttt{Cons}) = 1$$

Example 11. We start with one of the examples where the calculus of Walther fails to show termination. The bad news is: our calculus fails to fully automatically prove termination in this example, too; but as good news, we can slightly direct our tableaux with the correct approximation such that they are capable to show termination. The function which we want to analyze is `shuffle` with the following definition:

```
shuffle xs = case xs Nil shuffle2
shuffle2 x xs = Cons x (shuffle (reverse xs))

reverse xs = case xs Nil reverse2
reverse2 x xs = append (reverse xs) (Cons x Nil)
```

Nf-termination of `append` and `reverse` is easy to show. The construction of a termination tableau for `shuffle` gives the preclosed tableau of figure A.1.

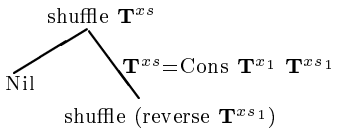


Figure A.1: A termination tableau for `shuffle`.

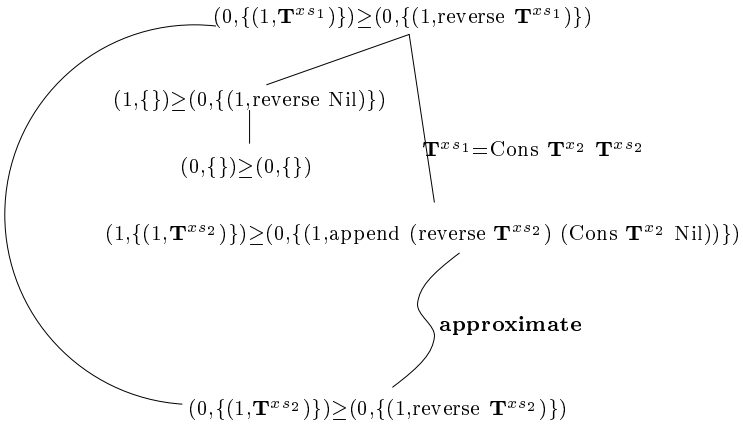


Figure A.2: An ordering tableau for `reverse`.

In order to close the tableau, we have to show for some termination function τ that

$$\tau(\texttt{Cons}\ \mathbf{T}^{x_1}\ \mathbf{T}^{xs_1}) > \tau(\texttt{reverse}\ \mathbf{T}^{xs_1})$$

We can try to built the corresponding ordering tableau as seen in figure A.2. Without any approximation we could not have closed the tableau. An infinite sequence of reduction of the function `reverse` would have occured. Rather than reducing infinitely we stopped after the first reduction of `reverse` and approximated the right hand side of the ordering proposition. The approximation applied here is proved to be sound by another ordering tableau which can be seen in figure A.3. This example shows that it can be a good heuristics avoiding reducing a recursive function several times on one path.

Now let us consider the sorting algorithms given in [Wal94].

Example 12. The first sorting algorithm is a form of `bubble`-sort:

```
bubblesort xs = case xs Nil bubblesort2
bubblesort2 x xs = Cons (last (bubble (Cons x xs)))
                       (bubblesort (but_last (bubble (Cons x xs))))

bubble xs = case xs Nil bubble2
```

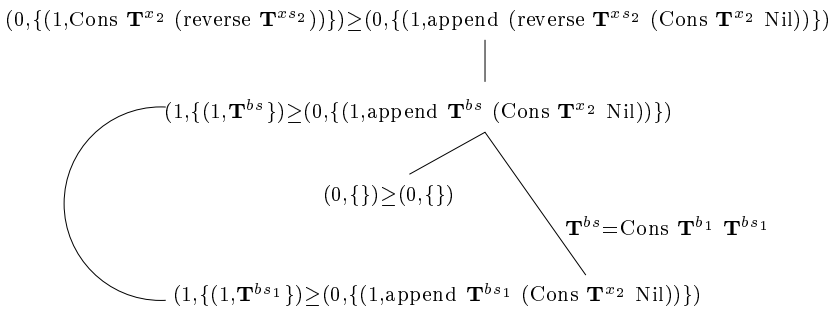


Figure A.3: An ordering tableau which proves the approximation of the ordering tableau for `reverse` to be correct.

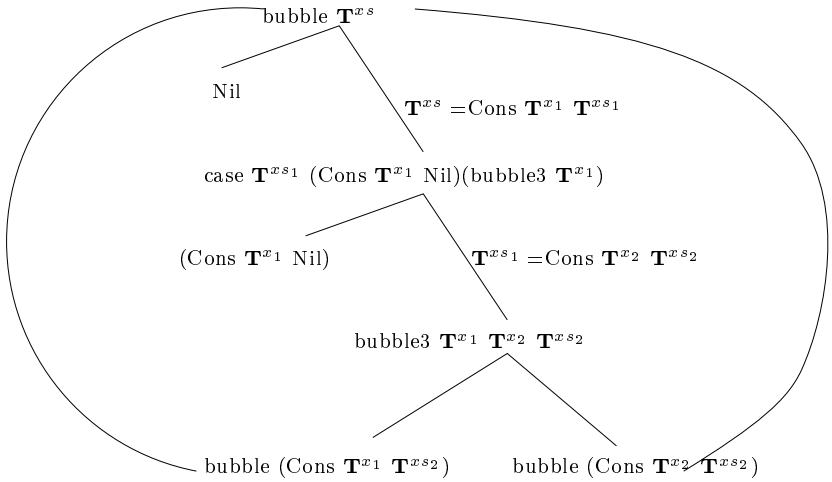


Figure A.4: A termination tableau for `bubble`.

```
bubble2 x xs = case xs (Cons x Nil) (bubble3 x)
bubble3 x1 x2 xs = case x1<=x2 (Cons x2 (bubble (Cons x1 xs)))
                               (Cons x1 (bubble (Cons x2 xs)))

but_last xs = case xs Nil but_last2
but_last2 x xs = case xs Nil (but_last3 x)
but_last3 x1 x2 xs = Cons x1 (but_last (Cons x2 xs))
```

The function `last` of the Haskell standard prelude is easy to show to be nf-terminating.

We start with showing nf-termination of `bubble`. An abbreviated tableau can be found in figure A.4.

Now we can try to build a termination tableau for `bubbblesort`. The tableau is given in figure A.5. In order to close this tableau, we need to prove an ordering proposition involving the functions `but_last` and `bubble`. The ordering tableau for this proposition is given in figure A.6. As can be seen, this example runs

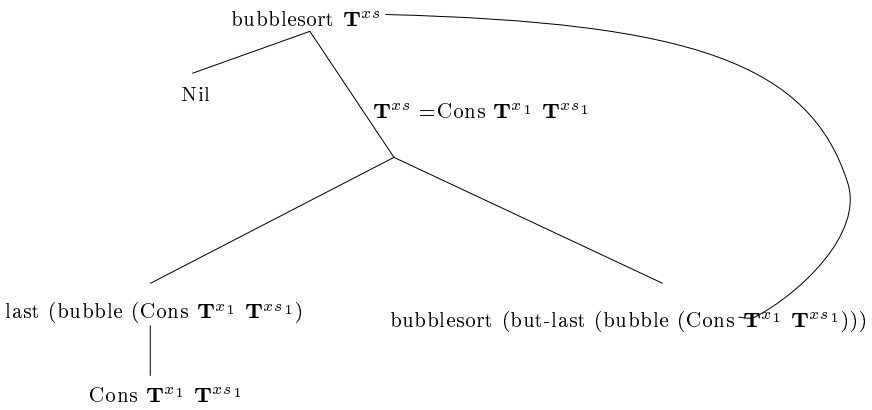


Figure A.5: A termination tableau for `bubblesort`.

trough the calculus smoothly but requires quite a lot of reductions.

Example 13. The second sorting algorithm presented in [Wal94] selects the minimum from a list and appends this to the front of the resulting list.

```
selsort xs = case xs Nil selsort2
selsort 2 x xs = case (x==(minimum (Cons x xs)))
                  (Cons x (selsort xs))
                  (selsort (remove (minimum (Cons x xs)) x xs))

remove xs n m = case xs Nil (remove2 n m)
remove2 n m x xs = case (n==x) (Cons m xs)
                               (Cons x (remove n m xs))
```

The standard prelude function `minimum` is nf-terminating for non-empty lists. Implicitly we have shown this when we proved nf-termination of `fold`. `minimum` is defined through `fold`-functions on lists.

For the function `remove` nf-termination can be shown by a simple termination tableau. The reader can try this for himself. It remains to show nf-termination of `selsort`. First we expand the termination tableau of figure A.7. In order to close this tableau a simple ordering proposition has to be proved. This is done by the ordering tableau of figure A.8.

Example 14. The third sorting algorithm we investigate is `minsort`. Here the smallest element of the list is deleted.

```
minsort xs = case xs Nil minsort2
minsort2 x xs = Cons (minimum (Cons x xs))
                (minsort (deletemin (Cons x xs)))
```

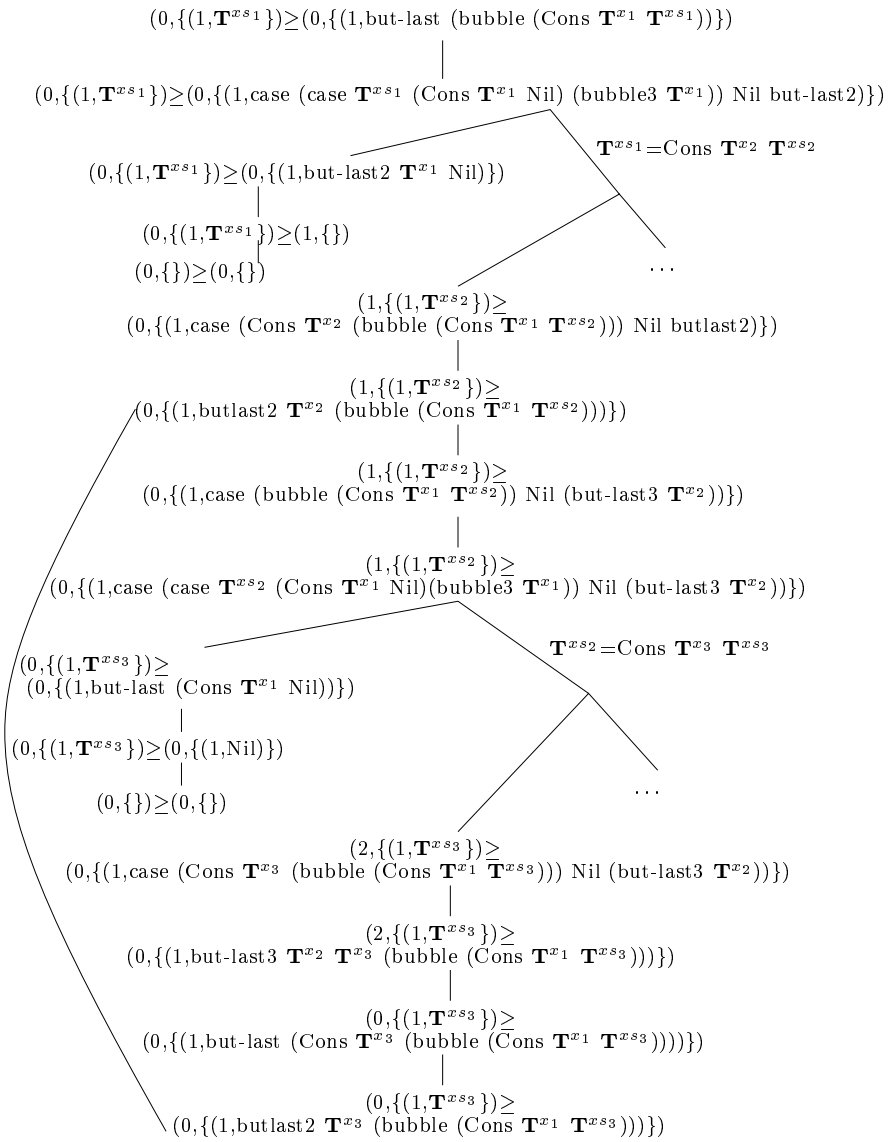


Figure A.6: An ordering tableau for the ordering proposition necessary for closing the termination tableau of `bubblesort`.

```
deletemin xs = case xs Nil deletemin2
deletemin2 x xs = case xs Nil (deletemin3 x)
deletemin3 x1 x2 xs
 = case (x1<=x2) (Cons x2 (deletemin (Cons x1 xs)))
                 (Cons x1 (deletemin (Cons x2 xs)))
```

The termination tableaux for `deletemin` and `minsort` are pretty easy to expand. The only problematic case is the ordering proposition which has to be proved in order to close the tableau for `minsort`. The corresponding ordering tableau can

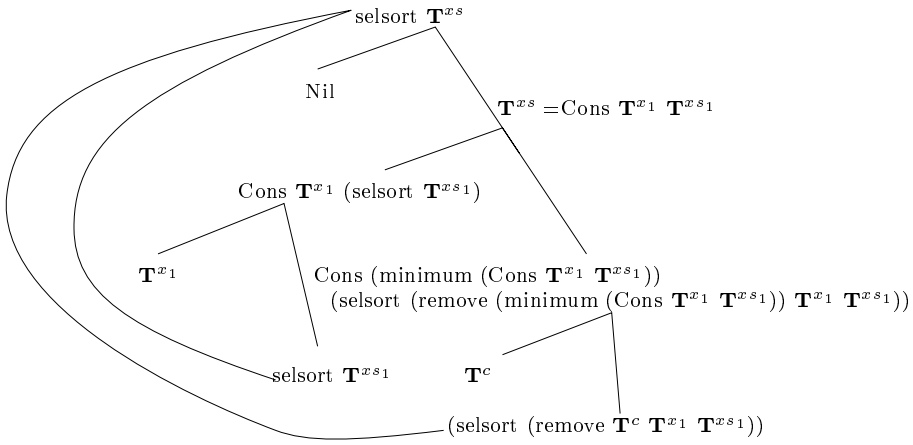


Figure A.7: A termination tableau for `selsort`.

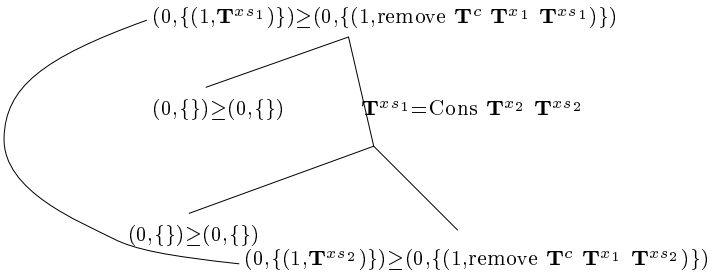


Figure A.8: An ordering tableau involving `remove`.

be seen in figure A.9.

Example 15. The next sorting algorithm we consider is the well-known `quicksort` function which is a typical divide-and-conquer algorithm.

```
quicksort xs = case xs Nil quicksort2
quicksort2 x xs = append (quicksort (smaller x xs))
```

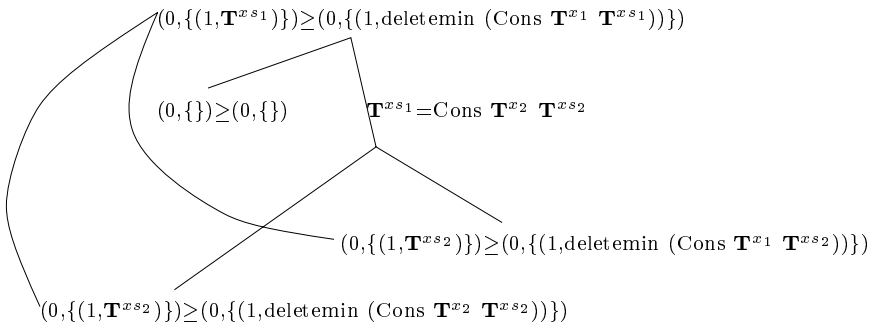


Figure A.9: An ordering tableau involving `deletemin`.

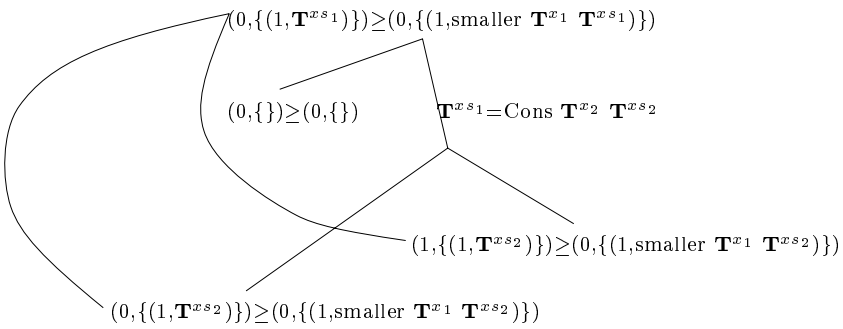


Figure A.10: An ordering tableau involving `smaller`.

```
                        (Cons x (larger x xs))

smaller x xs = case x Nil (smaller2 x)
smaller2 n x xs = case (n<x) (smaller n xs)
                             (Cons x (smaller n xs))

larger x xs = case x Nil (larger2 x)
larger2 n x xs = case (n>x) (larger n xs)
                            (Cons x (larger n xs))
```

The functions `smaller`, `larger` and the standard prelude function `append` are easily shown to be nf-terminating. The expansion of a termination tableau for `quicksort` will then lead pretty soon to two analogous ordering propositions. One of them concerning `smaller` the other concerning `larger`. We give the ordering tableau for `smaller` in figure A.10.

Example 16. The last sorting algorithm we want to investigate is a *merging* function. The list is distributed into two sublists and these are merged to an ordered list again:

```
mergesort xs = case xs Nil mergesort2
mergesort2 x xs = case xs (Cons x Nil) (mergesort3 x)
mergesort3 x1 x2 xs
 = merge (mergesort (distributeodd (Cons x1 (Cons x2 xs)))
         (mergesort (distributeeven (Cons x1 (Cons x2 xs))))

distributeodd xs = case xs Nil distributeodd2
distributeodd2 x xs = case xs (Cons x Nil) (distributeodd3 x)
distributeodd3 x1 x2 xs = Cons x1 (distributeodd xs)

distributeeven xs = case xs Nil distributeeven2
```

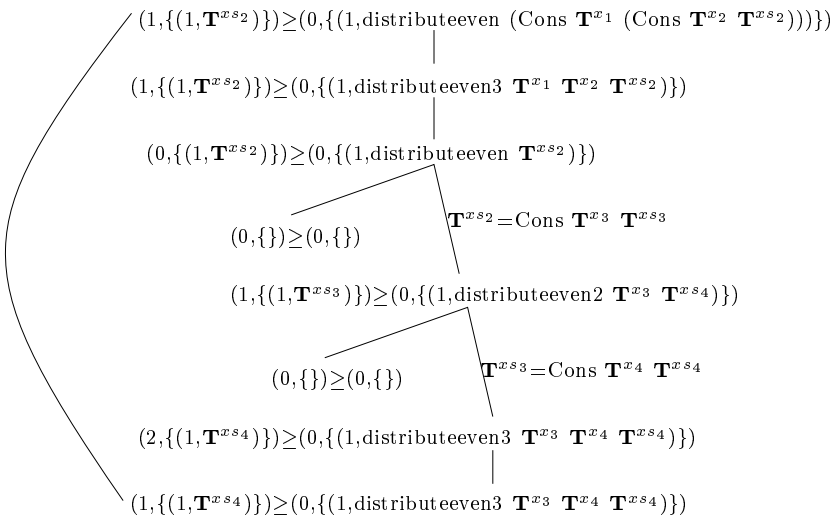


Figure A.11: An ordering tableau involving `distributeeven`.

```
distributeeven2 x xs = case xs Nil (distributeeven3 x)
distributeeven3 x1 x2 xs = Cons x1 (distributeeven xs)

merge xs ys = case xs ys (merge2 ys)
merge2 ys x xs = case ys (Cons x xs) (merge3 x xs)
merge3 x xs y ys = case (x<y) (Cons x (merge xs (Cons y ys)))
                              (Cons y (merge (Cons x xs) ys))
```

The functions `merge`, `distributeodd` and `distributeeven` as well run smoothly through the termination calculus. Nf-termination is easily shown for them. In order to close the termination tableau for `mergesort` we again have to prove two ordering propositions: one for `distributeodd` and one for `distributeeven`. These ordering propositions are quite similar. The ordering tableau for `distributeeven` is given in figure A.11.

A.2 Higher Order Functions

We will give termination proofs for higher-order combinator parsers as they are presented e.g. in [Wad85, FL89]. This example is a typical program using the higher order feature. We will assume that there is a basic type token with an equality function.[1]

The first parser function we need, is a function which checks if the token list starts with a certain token. The next two functions do not involve any recursion and so they trivially terminate. Simple closed tableaux can be derived for them.

[1] These tokens could be of a type *Char* which we did not introduce in our language.

```
lit x ys = case ys Nil (lit2 x)
lit2 x y ys = case (x == y) (Cons ys Nil) Nil
```

Now we have the basic functions that enable us to find tokens in a token list. We provide two combinator functions which can combine two parsers. The first one is the alternative combinator: `alt p q x = append (p x) (q x)`
This function does not cause any problem. `append` can be proved nf-terminating. If `p` and `q` are nf-terminating functions then `alt` will be nf-terminating.

We have shown that any `alt`-combination of parsers constructed with `lit` is nf-terminating. That means that the following parser which parses any alphanumeric-symbol is nf-terminating:

```
alphnum = alt (lit 'a')(alt (lit 'b')(alt (lit 'c')
          ...(alt (lit 'Y')(lit 'Z'))...)
```

The second function we need to define a parser is the sequential combinator:

```
seq p q xs
 = case (p xs) Nil (seq2 q)

seq2 q x xs
 = append (q x) (seq3 q xs)

seq3 q xs = case xs Nil (seq2 q)
```

We can first analyze `seq` under the assumption that its arguments of functional type are nf-terminating functions. This gives rise to the tableau in figure A.12.

We have proved `seq` to be nf-terminating. In a parser for a recursively defined language this will not be sufficient enough. One of the arguments for the function `seq` may be a recursive call to the parser which is constructed by the call of `seq`. In such a situation we cannot assume the arguments of `seq` to be nf-terminating. Actually there is a quite well-known class of `seq`-combinations which are known to be looping: parsers for a grammar with left-recursion. That parsers for a left-recursive grammar are looping can automatically be proved by the strictness analyzer presented in [SSPS95, Sch95].

A simple parser which uses `seq` recursively is the parser for identifiers:

```
ident = alt (seq alphnum ident) alphnum
```

For parsers defined in such a way we have to make a separate termination proof.
The termination tableau for `ident` turns out to be very large. We are only able to print out parts of it in figure A.13. The reason for this is that we have to check out every possible outcome of `alphnum` to see if it decreases the list for the recursive call of `ident`. Fortunately, as large as the tableau might appear, it can nevertheless be closed even without applying any dirty tricks. Note that we did not even have to prove an ordering proposition for the proof. Unfortunately there are nf-terminating parsers which can be constructed with our parser-combinators presented so far, where our calculus fails to prove termina-

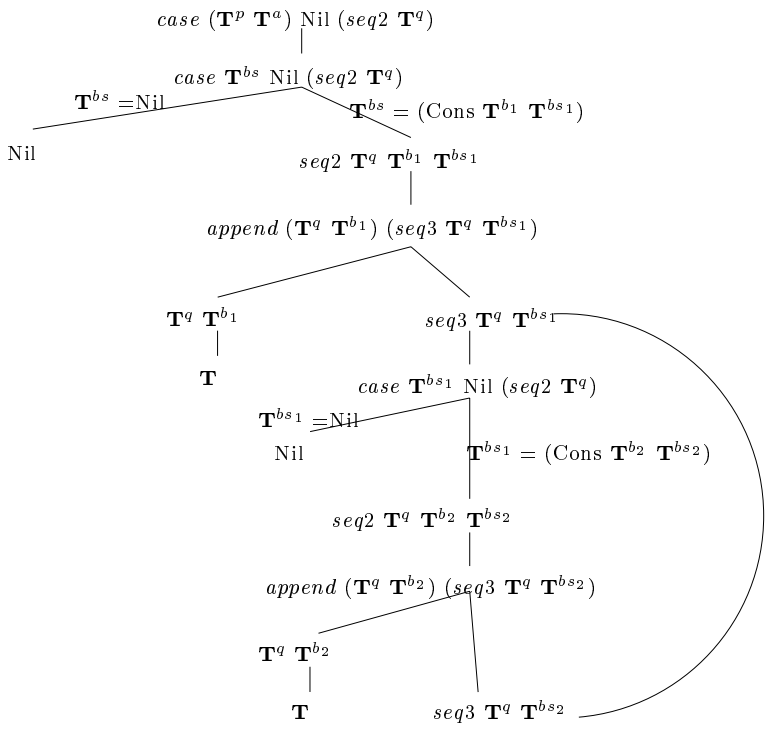


Figure A.12: A closed tableau for (`seq` p q $\mathbf{T}^a$)

tion. Such parser are recursive sequences of a parser with itself. An example is:

```
identtree = alt ident
             (seq (lit '(')
                  (seq identtree
                       (seq (lit ',')
                            (seq identtree (lit ')')))))
```

The problem of such examples is that they basically entail a sequence of the form `p = seq p p`.[2] The sequence operator `seq` demands a sequential evaluation, so basically we have above: to apply the parser `p`, parse the input string with the parser `p` and apply the parser `p` to the result of this parse. This means in a certain way, to evaluate an application `p xs`, we have to evaluate the expression `p (head (p xs))`. To close a termination tableau for `p` would need some knowledge about the result of `p` itself. This is currently beyond the scope of our method.

[2] `p` is of course looping. The non-looping variants of such examples have the same problems.

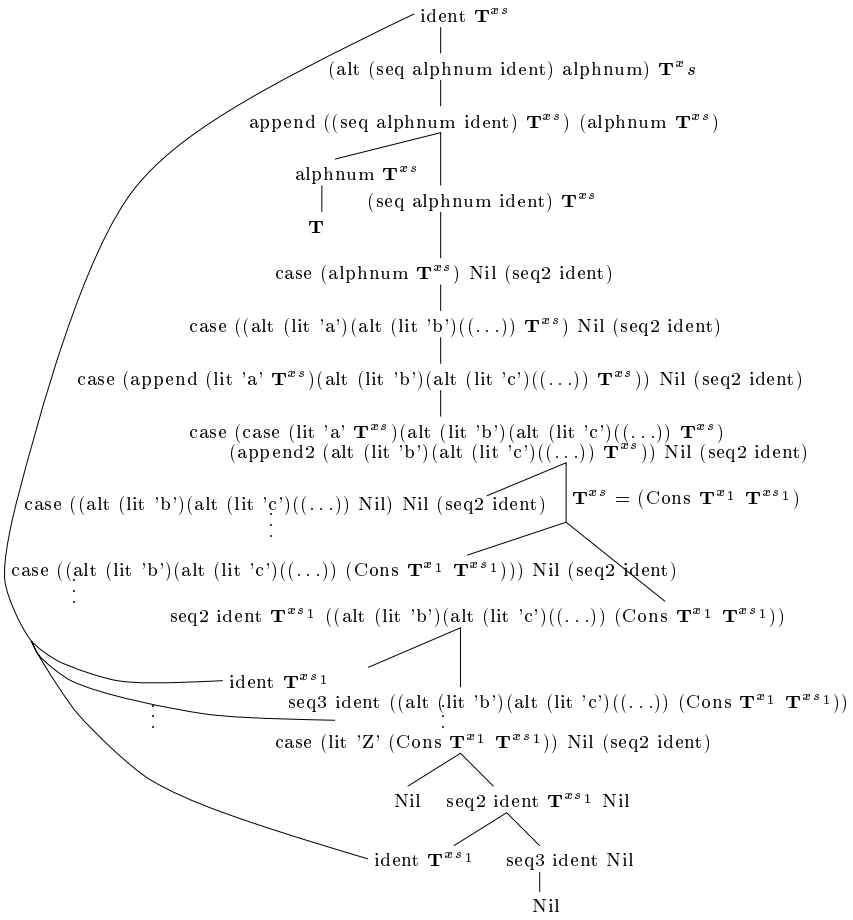


Figure A.13: Parts of a closed tableau for `ident`.

A.3 Infinite Lists

We will give a proof that a program which produces an infinite number of primes will lazy-terminating in its spine. The program which produces the primes will be a bit unusual. We cannot use the usual sieve of Eratosteles program, because this uses the function filter, which cannot guarantee that applied to an infinite list it will produce a head normal-form. Note that, if we could have proved the usual sieve program could be shown to be lazy-terminating in its spine, then we would have proved that there are infinitely many prime numbers. This is a bit to much expected from a fully automatic termination prover.

Instead of the usual sieve program we give a program which assures that if there are only finitely many prime numbers the resulting list will produce an infinite list of prime numbers by repeating some of them infinitely.

So here is the first part of our prime number program:

```
primes = map select (zip prim (fromstep 0 1))
```

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

zip xs ys = case xs Nil (zip2 ys)
zip2 ys x xs = case ys Nil (zip3 x xs)
zip3 x xs y ys = Cons (Pair x y) (zip xs ys)

repeat x = Cons x (repeat x)

fromstep x s = Cons x (fromstep (x+s) s)

prim = map select (zip (repeat primlists) (fromstep 100 100))
```

The second part, where we will see the definition of `select` and `primlists` will be given further down. We will not need them for the first part of our proof.

Let us prove that `primes` is lazy-terminating in its spine. We need an evaluator twhich forces the evaluation of a list to an arbitrary but fixed length:

```
lt-spine n xs = case xs True (lt-spine2 n)
lt-spine2 n x xs = case n True (lt-spine3 x xs)
lt-spine3 x xs n = lt-spine n xs
```

We need to prove that `lt-spine` $\mathbf{T}^n$ `primes` is nf-terminating. We give the main path of the corresponding termination tableau:

`lt-spine primes` $\mathbf{T}^n$
$\rightarrow$`case primes True (lt-spine2` $\mathbf{T}^n$`)`
$\rightarrow C^1$[`map select (zip prim (fromstep` $\mathbf{T}^{a_0}$ $\mathbf{T}^{a_1}$`))`]
$\rightarrow C^1$[`case (zip prim (fromstep` $\mathbf{T}^{a_0}$ $\mathbf{T}^{a_1}$`)) Nil (map2 select)`]
$\rightarrow C^2$[`zip prim (fromstep` $\mathbf{T}^{a_0}$ $\mathbf{T}^{a_1}$`)`]
$\rightarrow C^2$[`case prim Nil (zip2 (fromstep` $\mathbf{T}^{a_0}$ $\mathbf{T}^{a_1}$`))`]
$\rightarrow C^3$[`map select (zip (repeat primlists) (fromstep` $\mathbf{T}^{a_{100}}$ $\mathbf{T}^{a_{101}}$`))`]
$\rightarrow C^3$[`case (zip (repeat primlists) (fromstep` $\mathbf{T}^{a_{100}}$ $\mathbf{T}^{a_{101}}$`))`
`Nil (map2 select)`]
$\rightarrow C^4$[`(zip (repeat primlists) (fromstep` $\mathbf{T}^{a_{100}}$ $\mathbf{T}^{a_{101}}$`))`]
$\rightarrow C^4$[`case (repeat primlists) Nil (zip2 (fromstep` $\mathbf{T}^{a_{100}}$ $\mathbf{T}^{a_{101}}$`))`]
$\rightarrow C^5$[`(repeat primlists)`]
$\rightarrow C^5$[`Cons primlists (repeat primlists)`]
$\rightarrow C^4$[`zip2 (fromstep` $\mathbf{T}^{a_{100}}$ $\mathbf{T}^{a_{101}}$`) primlists (repeat primlists)`]
$\rightarrow C^4$[`Cons (Pair primlists` $\mathbf{T}^{a_{100}}$`)`
`(zip (repeat primlists) (fromstep` $\mathbf{T}^{a_{200}}$ $\mathbf{T}^{a_{101}}$`))`]
$\rightarrow C^3$[`map2 select (Pair primlists` $\mathbf{T}^{a_{100}}$`)`

$$
\begin{array}{l}
\quad \texttt{(zip (repeat primlists) (fromstep } \mathbf{T}^{a_{200}}\ \mathbf{T}^{a_{101}}\texttt{))]} \\
\to C^3[\texttt{Cons (select (Pair primlists } \mathbf{T}^{a_{100}}\texttt{))} \\
\quad \texttt{(map select (zip (repeat primlists) (fromstep } \mathbf{T}^{a_{200}}\ \mathbf{T}^{a_{101}}\texttt{)))]} \\
\to C^2[\texttt{zip2 (fomstep } \mathbf{T}^{a_0}\ \mathbf{T}^{a_1}\texttt{)(select (Pair primlists } \mathbf{T}^{a_{100}}\texttt{))} \\
\quad \texttt{(map select (zip (repeat primlists)(fromstep } \mathbf{T}^{a_{200}}\ \mathbf{T}^{a_{101}}\texttt{))]} \\
\to C^2[\texttt{case (fomstep } \mathbf{T}^{a_0}\ \mathbf{T}^{a_1}\texttt{) Nil)} \\
\quad \texttt{(zip3 (select (Pair primlists } \mathbf{T}^{a_{100}}\texttt{))} \\
\quad \texttt{(map select (zip (repeat primlists)(fromstep } \mathbf{T}^{a_{200}}\ \mathbf{T}^{a_{101}}\texttt{)))]} \\
\to C^2[\texttt{zip3 (select (Pair primlists } \mathbf{T}^{a_{100}}\texttt{))} \\
\quad \texttt{(map select (zip (repeat primlists) (fromstep } \mathbf{T}^{a_{200}}\ \mathbf{T}^{a_{101}}\texttt{)))} \\
\quad \mathbf{T}^{a_0}\ \texttt{(fromstep}\mathbf{T}^{a_2}\ \mathbf{T}^{a_1}\texttt{)]} \\
\to C^1[\texttt{map2 select (Pair (select (Pair primlists } \mathbf{T}^{a_{100}}\texttt{))} \mathbf{T}^{a_0}\ \texttt{)} \\
\quad \texttt{(zip (map select (zip (repeat primlists)} \\
\quad \texttt{(fromstep } \mathbf{T}^{a_{200}}\ \mathbf{T}^{a_{101}}\texttt{)))(fromstep}\mathbf{T}^{a_2}\ \mathbf{T}^{a_1}\texttt{))]} \\
\to \texttt{lt-spine2 } \mathbf{T}^{n} \\
\quad \texttt{(select (Pair (select (Pair primlists } \mathbf{T}^{a_{100}}\texttt{))} \mathbf{T}^{a_0}\texttt{))} \\
\quad \texttt{(map select (zip (map select (zip (repeat primlists)} \\
\quad \texttt{(fromstep } \mathbf{T}^{a_{200}}\ \mathbf{T}^{a_{101}}\texttt{)))(fromstep } \mathbf{T}^{a_2}\ \mathbf{T}^{a_1}\texttt{))))} \\
\to \texttt{lt-spine } \mathbf{T}^{n_1}\texttt{(map select (zip (map select (zip (repeat primlists)} \\
\quad \texttt{(fromstep } \mathbf{T}^{a_{200}}\ \mathbf{T}^{a_{101}}\texttt{)))(fromstep } \mathbf{T}^{a_2}\ \mathbf{T}^{a_1}\texttt{))))} \\
\to C^1[\texttt{(map select (zip (map select (zip (repeat primlists)} \\
\quad \texttt{(fromstep } \mathbf{T}^{a_{200}}\ \mathbf{T}^{a_{101}}\texttt{)))(fromstep } \mathbf{T}^{a_2}\ \mathbf{T}^{a_1}\texttt{))))]} \\
\to C^1[\texttt{case (zip (map select (zip (repeat primlists)} \\
\quad \texttt{(fromstep } \mathbf{T}^{a_{200}}\ \mathbf{T}^{a_{101}}\texttt{)))(fromstep } \mathbf{T}^{a_2}\ \mathbf{T}^{a_1}\texttt{))))} \\
\quad \texttt{Nil (zip2 select)]} \\
\to C^2[\texttt{zip (map select (zip (repeat primlists)} \\
\quad \texttt{(fromstep } \mathbf{T}^{a_{200}}\ \mathbf{T}^{a_{101}}\texttt{)))(fromstep } \mathbf{T}^{a_2}\ \mathbf{T}^{a_1}\texttt{)))]} \\
\to C^2[\texttt{case (map select (zip (repeat primlists)} \\
\quad \texttt{(fromstep } \mathbf{T}^{a_{200}}\ \mathbf{T}^{a_{101}}\texttt{))) Nil (zip2 (fromstep } \mathbf{T}^{a_2}\ \mathbf{T}^{a_1}\texttt{)]} \\
\to C^3[\texttt{map select (zip (repeat primlists)} \\
\quad \texttt{(fromstep } \mathbf{T}^{a_{200}}\ \mathbf{T}^{a_{101}}\texttt{))]} \\
\to C^3[\texttt{case (zip (repeat primlists)} \\
\quad \texttt{(fromstep } \mathbf{T}^{a_{200}}\ \mathbf{T}^{a_{101}}\texttt{)) Nil (map2 select)]} \\
\to C^4[\texttt{case (repeat primlists) Nil (zip2 (fromstep } \mathbf{T}^{a_{200}}\ \mathbf{T}^{a_{101}}\texttt{))]}
\end{array}
$$

The last expression in this reduction sequence has been on the path before (modulo a substitution of abstract variables). We can close the path here, because the Peano-number $\mathbf{T}^n$ has decreased by one.

If we want to prove fully lazy-evaluation of `primes` we also need to show that all lists elements of `primes` are nf-terminating[3]. This means for our example above, that we have to show nf-termination of:

[3] Actually we only have to show them to be lazy-terminating which is the same as nf-termination for basic values

`select (Pair (select (Pair primlists` $\mathbf{T}^{a_{100}}$`)) `$\mathbf{T}^{a_0}$`)`

It is now time to reveal the rest of our prime-number program:

```
primelists = map primelimit (fromstep 1 1)

primelimit limit
 = map hd (iterat (pseudosieve limit) (fromstep 2 1))

iterat f x = map (power f x) (fromstep 0 1)

power f x n = case n x (power2 f x)
power f x n = f (power f x n)

pseudosieve limit xs = case xs bot (pseudosieve2 limit)
pseudosieve2 limit p xs
 = append (filter (modtest p) (take limit xs)) (repeat 2)

modtest p x = ((mod x p)/= 0)

hd xs = case xs 2 hd2
hd2 x xs  = x

select pl = case pl select2
select2 ls n = case n (hd ls) (select3 ls)
select3 ls n = case ls bot (select4 n)
select4 n l ls = select2 ls n
```

Unfortunately a tableau for the remaining proof is far to large to be displayed in any form on paper.

Bibliography

[Bac86] Roland C. Backhouse. *Program Construction and Verification.* Prentice-Hall, Englewood Cliffs,N. J., 1986.

[BHA85] G. L. Burn, C. L. Hankin, and S. Abramsky. The theory for strictness analysis for higher order functions. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Structures*, number 217 in Lecture Notes in Computer Science, pages 42–62. Springer, 1985.

[Bur74] R. M. Burstall. Program proving as hand simulation with a little induction. In *Information Processing, IFIP*, pages 308–312. North-Holland Publishing Company, 1974.

[Bur87] Geoffrey Burn. Evaluation transformers - a model for the parallel evaluation of functional languages (extended abstract). In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 446–470. Springer, 1987.

[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.

[FL89] R. Frost and J. Launchbury. Constructing natural language interpreters in a lazy functional languages. *The Computer Journal*, 32(2):108–121, 1989.

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

[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 ??–?? Springer, 1995.

[Lan79] D. S. Lankford. On proving term-rewriting systems are noetherian. Technical Report MTP-3, Mathematics Department, Louisiana Technical University, 1979.

[Mar87] Ursula Martin. Extension funczions for multiset orderings. *Information Processing Letters*, 26:181–186, 1987.

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

[Myc80] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *4th International Symposium on Programming*, number 83 in Lecture Notes in Computer Science, pages 269–281. Springer, 1980.

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

[PJL91] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages: a Tutorial*. Prentice-Hall International, London, 1991.

[Sch95] Marko Schütz. The $G^{\#}$-machine: Efficient strictness analysis in Haskell. Technical Report 1/95, Johann Wolfgang Goethe-Universität, Fachbereich Informatik, January 1995.

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

[SS96] Schmidt-Schauß. A calculus for proving equivalence of programs in a non-strict functional language. obtainable by request from the author at schauss@informatik.uni-frankfurt.de, 1996.

[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 984 in Lecture Notes in Computer Science, pages 348–365. Springer, 1995.

[Ste92] Joachim Steinbach. Termination proofs of rewriting systems – heuristics for generating polynominal orderings. Technical Report SEKI-Report SR-91-14, Universität Kaiserslautern, 1992.

[Wad85] Phil Wadler. How to replace failure by a list of successes. In *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 113–128. Springer, 1985.

[Wad87] Phil Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declerative Languages*, chapter 12. Ellis Horwood Limited, Chichester, 1987.

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