Strictness Analysis by Abstract Reduction Using a Tableau Calculus

Manfred Schmidt-Schauß, Sven Eric Panitz, Marko Schütz*

Fachbereich Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt
Germany
e-mail:{schauss,panitz}@informatik.uni-frankfurt.de
schuetz@inf.fu-berlin.de

Abstract. Strictness analysis has been an active field of research since strictness information is of great importance for generating efficient code for lazy functional languages.
One of the most efficient and precise methods for strictness analysis is abstract reduction. Abstract reduction is able to find strictness information in a language with higher-order functions and structured data types. We present a tableau-based calculus for abstract reduction on a lazy functional core language. This approach permits us to prove soundness in an easy and transparent way, and also provides a soundness proof for existing implementations of abstract reduction.

1 Introduction

The compilation of lazy functional languages and their optimization is an active field of research. One of the optimization methods is strictness analysis that tries to find out, which arguments of a function call are sure to be evaluated. Once this is known, the compiler can use this knowledge to arrange the sequence of evaluations in optimal order. This knowledge would also permit to find expressions that can be evaluated in parallel. Some techniques how a compiler may exploit strictness information is e.g. shown in [PJS94, HB93, PJL91a]. A function f is strict, if it evaluates its argument. The rigorous definition is that f is strict, iff $f\ \bot = \bot$, where $\bot$ is the semantical value of all expressions without weak head normal form. A generalization is to consider more than one argument and to say a function f is strict in its i^{th} argument if $f\ x_1\ \ldots\ x_{i-1}\ \bot\ x_{i+1}\ \ldots x_n = \bot$ for all possible arguments x_j with $i \neq j$.
An even more general type of information is so-called context information. Suppose a function f yields some algebraic data type like lists. Then one might be

* We would like to thank Florian Panitz for correcting some of the spelling errors and his advice on language use.

interested in whether evaluating an application $f\ x$ to a certain degree evaluates x to a certain degree, e.g. whether the complete list structure of x will be evaluated when the head normal form of the application is needed.

During the last decade quite a number of different methods for strictness analysis have been proposed. The most common is abstract interpretation [CC77], which was originally used for strictness analysis in [Myc80]. Abstract interpretation puts well-known denotational semantics in a more general form. It is well understood from the theoretical point of view, this is also an active field of research, and some successful implementations for strictness analysis have been built [JHR94].

Another currently available method for strictness analysis is abstract reduction, which abstracts the operational semantics.

In this paper we make a contribution to the investigation of abstract reduction. Abstract reduction for strictness analysis was originally proposed by Nöcker in [Nöc93]. It is successfully incorporated into the Clean [NSvP91, PvE93] compiler. Another, freely available implementation of abstract reduction in Haskell is introduced in [Sch95].

The underlying idea of abstract reduction is, intuitively spoken, quite simple: Rewriting steps are performed on expressions that have the special symbols `Bot` and `Top`. The constant `Bot` stands for expressions that are interpreted as $\perp$ in the standard interpretation and `Top` stands for any expression. In order to check for a supercombinator f with n arguments whether it is strict in its i^{th} argument, the abstract expression f `Top`...`Bot`...`Top` will be reduced, where all arguments are `Top`, except the i^{th} one which is `Bot`. If it can be proved (using reduction) that the result must be `Bot`, we have the desired strictness information.

When we detect a loop in an abstract reduction sequence, the expression that gives rise to this loop is rewritten by `Bot`. The semantical justification for this introduction of `Bot` is that the abstract reduction sequence mimics only looping concrete reduction sequences so that no weak head normal form of the reduced expression exists.

Nöcker defines an abstract reduction relation $\rightarrow_\alpha$ which is restricted to perform so-called "needed reductions". Comparing this with our approach it turns out that our relation gives a decidable sufficient criterion for a reduction to be a "needed reduction", whereas "needed reduction" is undecidable. Furthermore we can also permit arbitrary reductions, if at least one reduction step is necessary.

Up to now no satisfying soundness proof for abstract reduction has been presented. This might be one reason for abstract reduction not yet being incorporated in any Haskell compiler and for being regarded with some reservations in the functional programming community.

There are mainly two reasons for there being still no satisfying soundness proof for abstract reduction. One is that formulations of abstract reduction have dealt with quite a lot of implementational overhead. The other one is that abstract reduction has been seen as a special form of abstract interpretation. This made soundness proofs very difficult. Results in investigating the soundness of abstract reduction can be found in [Nöc92] and [vEGHN93].

We present a reformulation of abstract reduction in terms of a tableau calculus and give simple soundness proofs. Our formulation is quite simple and self contained.

The paper is structured as follows: In section 2 the language we intend to analyse will be defined. In section 3 abstract values will be introduced. In section 4 we will define abstract reduction steps in terms of deduction rules on strictness tableaux and we will prove them to be sound. In section 5, extensions and improvements of the calculus are discussed and in section 6 we present some figures for a running implementation of the method.

2 The Language

We define Λ_c, a simple polymorphic typed functional core language. It closely resembles the language used in [PJL91b] which is the source language for our implementation of abstract reduction [Sch95]. The main difference lies in the fact that we require the alternatives of a case expression to be complete.

2.1 Expressions and Supercombinator Definitions

The type system for Λ_c consists of given basic types including *Num*, function types and a finite number of algebraic type names. Polymorphic types are allowed. The following constants are in Λ_c:

- Numerical constants.
- Built-in functions like $+, -, *, /$.
- For every algebraic type A 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 $\tau_{j_1} \to \ldots \tau_{j_k} \to A$.
- one case constant $\mathbf{case}_A$ for every algebraic type name, which takes $n+1$ arguments, where n is the number of construtor 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 $1 \le k \le n$, where m_k is the arity of the kth constructor of A.
- supercombinators that may be defined in a recursive way (see below).

An example of an algebraic type is *Bool* with the constants **True**, **False** and the case-constant **if**. Another example is (*List* α) with the constants **Nil**, **Cons** and the case-constant $\mathbf{case}_{List}$, which requires three arguments: the list expression, the expression for the **Nil**-case and a function taking the head and tail as arguments in the non-nil case.

2.2 Λ_c Expressions

Λ_c expressions are composed in the standard way, i.e., expressions may be variables x, constants c, and applications $(d\ e)$, where we require that Λ_c-expressions

are well-typed. We assume that application is left associating, i.e., $e_1\ e_2\ e_3$ is equivalent to $(e_1\ e_2)\ e_3$.
Recursions are defined via supercombinators. An expressions e with n free variables $x_1, \dots, x_n$ may be defined as a supercombinator: $sc\ x_1\ \dots\ x_n = e$. If e has type σ then sc is treated as constant with type $\tau_1 \to \dots \tau_n \to \sigma$.
To make life easier, we will assume that there is a non-terminating supercombinator defined by `sc-bot = sc-bot`.
As notational convention $C[t]$ will denote an expression which has at some position the subexpression t.

2.3 Operational Semantics

The operational semantics consists of the $\delta-$reductions. If $sc\ x_1\ \dots\ x_n = e$ is a supercombinator definition, then $sc\ t_1\ \dots\ t_n \to_\delta e[x_1/t_1, \dots, x_n/t_n]$.
$\delta-$rules for built-in primitive functions are the usual ones, i.e., applications of built-in functions to arguments can only be evaluated, if all arguments are basic values. Furthermore, they are assumed to be strict in all arguments.
For every case constant of an algebraic type there is a $\delta-$rule of the following form:

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

An expression is in *weak head normal form* (WHNF), if it is in one of the following forms:

- $c\ e_1\ \dots\ e_n$, where c is a constructor
- $c\ e_1\ \dots\ e_n$, where c is a built-in function, a supercombinator or a **case** constant that requires more than n arguments.

3 Abstract Expressions

In this section, we will define $\Lambda_c^\#$, the language of abstract expressions. $\Lambda_c^\#$ is the same as Λ_c, except that the two constants `Bot` and `Top` are added and that we do not worry about well-typedness of $\Lambda_c^\#$-expressions.
$\Lambda_c^\#$ expressions represent sets of expressions of Λ_c. We require a concretisation map γ :

Definition 1. The function $\gamma : \Lambda_c^\# \to \mathcal{P}(\Lambda_c)$ is recursively defined as follows:
- $\gamma(\texttt{Bot}) = \{$all non-terminating expressions in $\Lambda_c\}$
- $\gamma(\texttt{Top}) = \{$all expressions in $\Lambda_c\}$
- $\gamma(c) = \{c\}$, for constants c
- $\gamma(d\ e) = \{(d'\ e') | (d'\ e')$ is a expression in $\Lambda_c, d' \in \gamma(d), e' \in \gamma(e)\}$

Note that Λ_c-expressions are *per definition* well-typed.

Informally, $\gamma(e)$ is the set of expression in Λ_c that can be constructed from e by replacing every `Bot` with some expression without WHNF and replacing every `Top` by an arbitrary expression in Λ_c. Note that we only permit well-typed expressions in $\gamma(e)$. We exclude expressions $e^\# \in \Lambda_c^\#$ with $\gamma(e^\#) = \emptyset$ from the

language $\Lambda_c^\#$. Since we have a non-terminating expression of any type in the language, we have enough expressions that can be replaced for `Bot`.
A $\Lambda_c^\#$–expression is in WHNF, if it is an expression started by a constructor, or if it is either an application of a built-in function, a supercombinator or a **case** constant of arity n to less than n arguments. `Top` is also considered to be in WHNF. As a special case, basic values are in WHNF. The term `Bot` is not considered to be in WHNF. A *redex* is a position (also a subexpression) in an expression where a reduction can be performed.
A *necessary redex* r in a Λ_c-term t can be recursively defined as follows:

- A term in WHNF does not have any necessary redeces.
- If t is a redex, then t is a necessary redex of itself.
- A necessary redex of e_i is also a necessary redex of $(f\ e_1 \dots e_n)$, if f is a built-in function.
- A necessary redex of e_0 is also a necessary redex of $(e_0\ e_1\ \dots\ e_n)$, if e_0 is a proper expression
- A necessary redex of e_1 is also a necessary redex of $(case_A\ e_1\ \dots\ e_n)$

This definition is also applicable to $\Lambda_c^\#$, if we consider the following expressions as reducible: (`Top` x), (`Bot` x), $(f\ x_1 \dots x_n)$, if f is a built-in function and all x_i's are equal to `Top` or `Bot` or a basic value, ($\mathbf{case}_A$ `Top` $e_1\ \dots\ e_n$) and ($\mathbf{case}_A$ `Bot` $e_1\ \dots\ e_n$).
Note that our notion of abstract reduction is different from Nöcker's definition.
A *necessary reduction* is an abstract reduction where a necessary redex is reduced. The notation will be $t \to s$, if t can be reduced to s, $t \xrightarrow{*} s$, if t can be reduced to s using several reductions, $t \to_N s$, if t can be reduced to s by a necessary reduction, $t \xrightarrow{*}_N s$, if t can be reduced to s by zero or more necessary reductions and $\xrightarrow{m}_N$ to denote m reductions $\to_N$.
We have the following relation between necessary redexes in Λ_c and $\Lambda_c^\#$:

Lemma 2. *Let $e^\#$ be an expression in $\Lambda_c^\#$ which is not in WHNF and which is not equal to* `Bot`. *For every necessary redex $r^\#$ in $e^\#$ and for every $e \in \gamma(e^\#)$ there is also a necessary redex r in e that is equal to or below r.*
Conversely, for every necessary redex r in e there is a necessary redex $r^\#$ in $e^\#$ that is equal to or above r.

Proof. The proof can be done by a simple case analysis

4 Strictness Tableaux

In this section we will define abstract reduction as a deduction calculus, which we will call **STAR** (strictness tableau for abstract reduction). This calculus follows the ideas of tableaux calculus for predicate logics [Smu71].

Definition 3. A *tableau* is a finite tree whose nodes are marked with $\Lambda_c^\#$-expressions and whose edges are either unmarked or marked with "N" (for necessary reduction). Leaves may have as a second label "loop".
A tableau is *sound* if for every inner node n marked with $e^\#$ we have:

For every $e \in \gamma(e^{\#})$ there exists a direct son $e'^{\#}$ of $e^{\#}$ and a term $e' \in \gamma(e'^{\#})$, such that $e = e'$ or $e \to e'$.

Now, we will develop a calculus which derives new sound tableaux with the same root node from a sound tableau. The new tableaux always have to originate from an old one by extending a path with new leaves.
In the following we will describe the expansion rules for tableaux. An expansion rule is called *sound*, if it transforms sound tableaux into sound ones.

Definition 4. A tableau where all leaves are either marked with `Bot` or have a label "loop" is called *closed*.

In the following subsections we will present the expansion rules for strictness tableaux. Note that we never extend leaves that are marked with "loop".

4.1 $\delta-$rules

The simplest extension of tableaux is by reduction on a leaf. Abstract expressions can almost be regarded as concrete expressions. Thus the same δ-reduction as in the concrete case can be used. We define the first deduction rule for strictness tableaux:

> Suppose in a given tableau we have a leaf marked with $C[\mathbf{c}\ e_1\ \ldots\ e_n]$. $\delta-$reduction extends this tableau with the new leaf $C[e']$, if there is a $\delta-$rule: $\mathbf{c}\ e_1 \ldots e_n \to_\delta e'$ for $\mathbf{c}$. If a necessary redex is reduced, then the new edge between the old and the new leaf is marked with "N".

Lemma 5. *The δ-expansion rule is sound.*

Proof. Let $e^{\#}$ be the label of the old leaf and $e'^{\#}$ be the label at the new leaf introduced by $\delta-$reduction. For $e \in \gamma(e^{\#})$ we can perform the same reduction as in $e^{\#}$. We obtain some $e' \in \gamma(e'^{\#})$ with $e \to e'$.
Note that $\gamma(e')$ may also contain expressions that are not derived from expressions in $\gamma(e)$.

4.2 `Bot`-Application

The next rule is designed to perform an application of `Bot` to some argument. This can be regarded as introducing a $\delta-$rule for the constant `Bot`.
We state the following rule:

> Suppose we have a leaf marked with the expression $C[(\texttt{Bot}\ e)]$ in a given tableau. Then we can extend this tableau with the new leaf $C[\texttt{Bot}]$ at this path. The edge will be marked with N, if a necessary redex was reduced.

The soundness of `Bot`-application is obvious. All instances of an expression (`Bot` e) are non-terminating.

4.3 Special δ–rules for Λ_c

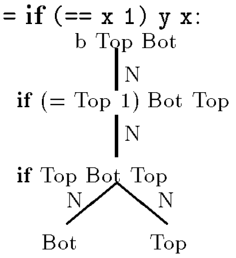
We will now enlarge the set of δ-rules for such cases where a built-in function constant includes one of the two constants `Bot` or `Top` as argument:

- $(f\ e_1\ \dots\ e_{i-1}$ `Bot` $e_{i+1}\ \dots\ e_n) \to_\delta$ `Bot`
  if f is a built-in function of arity n.
- $(f\ x_1\ \dots\ x_n) \to_\delta$ `Top`
  if f is a built-in function of arity n and it is applied to n arguments that are either basic values or equal to `Top` and at least one argument is equal to `Top`.
- (`Top` e) $\to_\delta$ `Top`
- (**case**$_A$ `Bot` $e_1\ \dots\ e_n) \to_\delta$ `Bot`

If we have reduced a necessary redex, then the edge will be marked with N. Finally, rules for **case** constants applied to `Top` as first argument have to be provided. These cannot be stated as further δ–rules, because in these cases the tableau will be branching.

> Suppose we have a leaf marked with $C[($**case**$_A$ `Top` $e_1 \dots e_n)]$ in a given tableau and the algebraic type **A** has n different constructors. Then a new tableau is derived by extending the tableau at this path with the n new leaves $C[(e_i$ `Top`...`Top`$)], 1 \le i \le n$ with n_i times the argument `Top`, where the type of e_i is $\tau_1^i \to \dots \to \tau_{n_i}^i \to \sigma$.
> If the case expression is a necessary redex, then the new edges are all marked with N.

In the following example we assume that `if` is the name for **case**$_{Bool}$ and that equality is the built-in equality on numbers. We analyse the supercombinator `b x y = if (== x 1) y x`:



This tableau is not closed. This is plausible, since `b 2 sc-bot` $\to$ `2`.

Lemma 6. *All the tableau expansion rules in this subsection are sound.*

Proof. This is an easy task.

4.4 `Top`-Approximation

> Suppose we have in a given tableau a leaf that is marked with $C[t]$. A new tableau can be derived by extending the path with the leaf $C[$`Top`$]$. The edge will not be marked.

The soundness of `Top`-approximation is obvious, since all instances of the old leaf are instances of the new leaf.

4.5 Loop-Detection

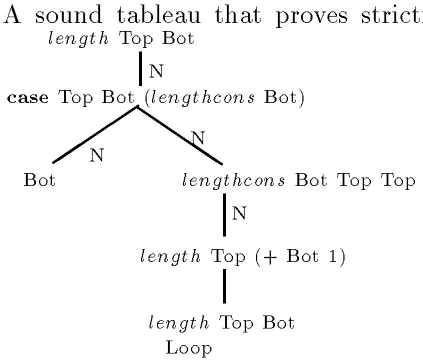
Finally, we need a rule for discovering loops in the evaluation. This rule will be called `Loop`-detection.

> Suppose we have a leaf marked with a term t, which is not in WHNF. If there exists some different node also marked with t on the path from the root to this leaf, and on the path from that node to the leaf there exists an edge that is marked with N, then the leaf will be marked `loop`.

We give an example before we consider the soundness. By now, we are in a position to prove the strictness of the iterative *length* function which is defined as the following supercombinator:

$$length\ xs\ n = \mathbf{case}_{List}\ xs\ n\ (lengthcons\ n)$$
$$lengthcons\ n\ x\ xs = length\ xs\ (+\ n\ 1)$$

A sound tableau that proves strictness of *length* in the second argument is:



Note that using only necessary reductions in this example prevents the detection of loops, since the subterm `(+ Bot 1)` will not be reduced.

4.6 Soundness of STAR

In this section we will prove the soundness of the calculus to compute strictness information, in other words, we will prove that if a tableau is closed, the term at the root has no WHNF. First, we will show that a derived tableau can be regarded as mimicking a concrete reduction.

Lemma 7. *Assume given a tableau generated by* **STAR**. *Let* $t^{\#}$ *be a term in the tableau and let* $t \in \gamma(t^{\#})$.
Then there exists a leaf $s^{\#}$ *that can be reached from* $t^{\#}$ *and some* $s \in \gamma(s^{\#})$, *such that* $t \xrightarrow{*} s$. *Furthermore, if there are* n *edges from* $t^{\#}$ *to* $s^{\#}$ *that are labelled* N, *then the reduction from* t *to* s *performs at least* n *necessary reductions.*

Proof. The proof can be done by carrying it out for a single expansion and by considering all possible cases. All the expansion rules of **STAR** are designed such that this lemma holds.

The proof of the soundness of `Loop`-detection turns out to be the hardest part. Since it is not a local rule, we have to prove its soundness in a global way for the whole tableau. Before we come to this proof, we need a preparatory lemma on the lengths of necessary reductions.
Consider the language Λ_c. For every term t with a WHNF, let $nr\#(t)$ be the minimal number of reductions from t to its WHNF using only necessary reduction steps. Note that this number of reductions is only meant as a technical one that serves as a basis for soundness proofs. It is not practical, since it is almost the worst method to reduce the term to WHNF.

Lemma 8. *We consider reductions of Λ_c-expressions.*

- *For every reduction of t to WHNF, the number of necessary reduction steps is not greater than $nr\#(t)$.*
- *If s is a proper subterm of t, such that t is not in WHNF and s is not below a necessary redex. Then $nr\#(s) \leq nr\#(t)$.*

Proof. Let t be a Λ_c-expression.
Claim 1): (see fig. 1 a)

If $s \rightarrow_N s'$ and $s \rightarrow s''$, and the reductions are different, then there exists an expression s''', such that $s'' \rightarrow_N s'''$ and $s' \xrightarrow{*} s'''$:

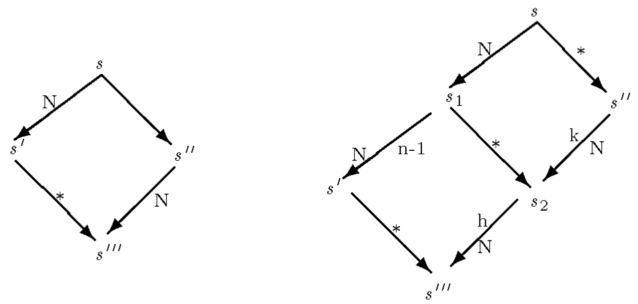


Fig. 1.

If the redeces for the reductions $s \rightarrow_N s'$ and $s \rightarrow s''$ are independent, then the reductions can be commuted, and the claim holds. In the other case, the only possibility is that the redex for the reduction $s \rightarrow_N s'$ is above the redex for the reduction $s \rightarrow s''$. The reductions may either be a supercombinator application or a case-application. In any case, the term s''' is obtained from s'' by performing the reduction corresponding to $s \rightarrow s'$. If this reduction eliminates the reduced subterm in s'', then we have the case that $s' = s'''$ and we are done. Otherwise, the subterm to be reduced may occur several times in s', and we have to make several reductions starting with s' to reach s'''.
Claim 2):

If $s \rightarrow_N s'$ and $s \xrightarrow{*} s''$, then there exists an expression s''', such that $s' \xrightarrow{*} s'''$ and $s'' \xrightarrow{m}_N s'''$ and $m \leq 1$.

This can be proved by induction on the number of reductions in $s \xrightarrow{*} s''$: In the case of zero reductions, we have $s' = s'''$. For one reduction, we use claim 1 in the case that $s' \neq s''$. If $s' = s''$, then the claim holds for $m = 0$.
Claim 3):

If $s \xrightarrow{n}_N s'$ and $s \xrightarrow{*} s''$, then there exists an expression s''', such that $s' \xrightarrow{*} s'''$ and $s'' \xrightarrow{m}_N s'''$ and $m \leq n$:

In the case $n = 1$, this is exactly claim 1. If $n > 1$, (sse fig. 1 b) then let s_1 be the first term in the reduction $s \rightarrow_N s_1 \xrightarrow{n-1}_N s'$. Since $nr\#(s_1) < nr\#(s)$, we can use induction for s_1. By claim 2, there is a term s_2, such that $s'' \xrightarrow{k}_N s_2$ and $s_1 \xrightarrow{*} s_2$ where $k \leq 1$. Using the induction hypothesis, there exists a term s''', such that $s' \xrightarrow{*} s'''$ and $s_2 \xrightarrow{h} s'''$ and $h \leq n-1$. Putting this together, the claim and thus the first part of the lemma is proved.
The second part of the lemma is obvious, since first reducing s to its WHNF requires $nr\#(s)$ necessary reductions. If s is a proper subterm of t which is not below a necesary redex, then at least some (necessary) top reduction of t is not counted in the reductions of s.

Theorem 9. *Let $t \in \Lambda_c^{\#}$. If* **STAR** *generates a closed tableau from t, then $\gamma(t) \subseteq \gamma(\texttt{Bot})$.*

Proof. Assume by contradiction that there is some term $t \in \Lambda_c^{\#}$, such that we can construct a closed tableau for t, but there is some term $t_0 \in \gamma(t)$, which has a WHNF. Let $d_N(n)$ be the number of N-marked edges from the root to node n. There is some node n_{min} marked by a term t_{min} that has a WHNF, such that $d_N(n_{min}) + nr\#(t_{min})$ is minimal. Now consider the path in the tableau that mimics the reduction of t_{min} that exists by Lemma 7. Since the tableau is closed, and t_{min} has a WHNF, there is a looping leaf n_1 marked s, and furthermore there is some other node n_2 above n_1 such that n_2 is also marked with s and there is an N-marked edge between n_1 and n_2. By minimality there is some term $s_1 \in \gamma(s)$ such that $d_N(n_{min}) + nr\#(t_{min}) = d_N(n_1) + nr\#(s_1)$. Since there is some N-marked edge between n_1 and n_2, we have that $d_N(n_2) + nr\#(s_1) < d_N(n_1) + nr\#(s_1)$, which contradicts the minimality of the node n_{min}.

Now we obtain soundness of abstract reduction:

Corollary 10. *Let sc be a supercombinator of arity n. If the tableau calculus* **STAR** *can derive a closed tableau from the root marked with $(sc\, e_1 \ldots e_n)$ with $e_j = \texttt{Top}$ for all $j \neq i$ and $e_i = \texttt{Bot}$ where $1 \leq i \leq n$, then sc is strict in its i^{th} argument.*

5 Extensions and Optimizations

Since implementations of abstract reduction exploit a lot of optimization possibilities to make analyses efficient, we have to consider these optimizations and argue for their soundness.

5.1 Loop-Detection Using an Ordering

One strategy common to the known implementations is to enhance `Loop`-detection by using an approximation ordering. Instead of checking equivalence of terms, usually a check is performed whether two abstract terms are in a less-equal relation.

The ordering $\leq_\alpha$ is defined as follows:

- e $\leq_\alpha$`Top` for all expressions $e \in \Lambda_c^{\#}$.
- `Bot` $\leq_\alpha e$ for all expressions $e \in \Lambda_c^{\#}$.
- $e \leq_\alpha e$ for all expressions $e \in \Lambda_c^{\#}$.
- $(s_1\ s_2) \leq_\alpha (t_1\ t_2)$ if $s_1 \leq_\alpha t_1$ and $s_2 \leq_\alpha t_2$.

Consider the following extended rule for loop-detection

> Suppose there is a leaf marked with a term t, such that t is not in WHNF, and there is some subexpression s of t that is not below a necessary redex. If on the path from the root to this leaf there is some different node marked with t' and $s \leq_\alpha t'$, and either on the path from that node to the leaf there is an edge that is marked with N, or s is a proper subexpression of t, then the leaf may be marked `loop`.

The calculus extended by this form of `Loop`-detection is called **STAR-XL**. In order to prove its correctness, some tools are required. Let $d \leq_I e$ be defined as follows:

- $e \leq_I e$ for all expressions e in $\Lambda_c^{\#}$
- `Bot` $\leq_I e$ for all expressions e in $\Lambda_c^{\#}$
- $(d_1\ d_2) \leq_I (e_1\ e_2)$ if $d_1 \leq_I e_1$ and $d_2 \leq_I e_2$

Based on this ordering we define a new version of concretisation:
$\gamma_I(e^{\#}) = \bigcup\{\gamma(d^{\#}) \mid d^{\#} \leq_I e^{\#}\}$
Based on this concretisation Lemma 7 can be proved for γ_I instead of γ.

Lemma 11. *Let $t^{\#}$ be a term in a tableau generated by* **STAR-XL***, such that $t^{\#}$ has a WHNF, and let $t \in \gamma_I(t^{\#})$.*
Then there exists a leaf $s^{\#}$ that can be reached from $t^{\#}$ and some $s \in \gamma_I(s^{\#})$, such that $t \xrightarrow{*} s$. Furthermore, if there are n edges on a path from $t^{\#}$ to $s^{\#}$ that are labelled N, then the reduction from t to s performs at least n necessary reductions.

Lemma 12. *$d \leq_\alpha e$ implies that $\gamma_I(d) \subseteq \gamma_I(e)$.*

Based on this, we can prove the soundness of the extended loop-detection.

Proposition 13. *The calculus* **STAR-XL** *is correct.*

Proof. (Sketch) Can be obtained by a modification of the proof of Theorem 9, the essential modification being the different measure for minimal nodes, which is $d_N(n_{min}) + nr\#(t_{min})$.

5.2 Strategies

The construction of strictness tableaux is non-deterministic. Given a non-closed tableau, we can select a leaf, and for this leaf we have several rules to choose from: for example `Top`-approximation or reduction of one of its redexes. Pragmatically, a good strategy is to test for `Loop`-detection as early as possible and to perform `Top`-approximation only if forced by resource limitations. A depth-first strategy or a variant thereof are good strategies to deal with non-determinism.
Furthermore, the algorithm has to give up if a leaf is generated that is marked with a WHNF or with `Top`.

5.3 Using directed acylcic graphs for tableaux

Implementations [Nöc93, Sch95] of abstract reduction have *UNIONs* as a representation mechanism. One use of *UNIONs* is to represent branching as a *UNION* of the leaves. It is tempting to do further simplifications on these *UNIONs*, for example to remove duplicates. It would be even better to eliminate a leaf d in a *UNION*, if there is some other leaf e in this union, and $d \leq_\alpha e$.
In the following we will argue under which condition the application of this method results in a sound algorithm. Therefore, we use labelled directed acyclic graphs instead of a tree, the corresponding objects are called *dag-tableaux*.
Besides the δ-rules, we have the following rules:

- Merging nodes:
  If there is a leaf n_1 marked s, where s is not in WHNF, and there is another node n_2 marked with t, such that $s \leq_\alpha t$, and neither n_2 is a predecessor of n_1, nor n_1 is a predecessor of n_2, then merge the two nodes and mark the new node with t.
- `Loop`-detection:
  Suppose a leaf n is marked with a term t, such that t is not in WHNF. Let s be a subexpression of t that is not below a necessary redex of t. If there is some predecessor node n' of n marked with t' and $s \leq_\alpha t'$ for some subexpression, and either s is a proper subexpression of t or on every path from n to a leaf there is an edge that is marked with N, then the leaf may be marked `loop`.

Interestingly, the merge of nodes would permit a cycle-detection between inner nodes after every merge, which is, however, not yet implemented.

The calculus **STAR-XLD** uses dag-tableaux, adapted version of the expansion rules, the Merge rule and the `Loop`-detection mentioned above. Closedness is defined as for tableaux. We have the following soundness claim.

Theorem 14. **STAR-XLD** *is a correct calculus for strictness analysis.*

Proof. Assume by contradiction that there is some term $t \in \Lambda_c^{\#}$, such that we can construct a closed dag-tableau for t using **STAR-XLD**, but there is some term $t_0 \in \gamma_I(t)$, which has a WHNF. Then there is some node n_{min} marked with t_{min}, such that there is a term $t_{min,0} \in \gamma_I(t_{min})$ with WHNF and $nr\#(t_{min,0})$ is minimal. By following the reduction, we can assume that n_{min} is a leaf. Since the dag-tableau is closed, n_{min} must be a leaf labelled "loop". Then we can find a node with a shorter reduction of a concretisation to WHNF: In the subexpression case, this is obvious. In the other case, $t_{min,0}$ is also a concretisation of an inner node, from which at least one necessary reduction can be made to find a further concretisation in a leaf.
We have reached a contradiction, showing that the term t cannot have a WHNF.

5.4 Pattern Matching and Local Definitions

In general, the language to be analysed may permit pattern matching and also local definitions. These syntactical constructions can be translated into the syntax of a core-language like Λ_c. This translation preserves the semantics of every function definition, hence the results of the strictness analysis on the translated version can be lifted to the high-level version. This justifies the correctness of abstract reduction adopted in a functional language with a richer syntax. Different variants and strategies of pattern-matching could all be translated. Sometimes it may be appropriate to translate into $\Lambda_c^{\#}$, for example omitted cases in a case-expression can be translated as `Bot`.

5.5 Context Analysis

Abstract reduction can be used to analyse context strictness. In the abstract interpretation framework for such an analysis a new abstract domain has to be defined, e.g. the four point list domain in [Wad87]. E. Nöcker in his paper [Nöc93] represents these abstract values by recursive unions of expressions in his abstract language. We can represent such recursive unions in $\Lambda_c^{\#}$. For example `botmem = if Top (Cons Top botmem) (Cons Bot Top)`.
The concretisations of such recursive $\Lambda_c^{\#}$-expressions can be defined via an infinite union.
The evaluator function e defined below recognizes the context $\bot\in$. In order to show that a function f evaluates a list-of-integers argument to normal form, it is sufficient to show that `e (f botmem)` is equal to `Bot`, therefore it is sufficient that a closed tableau can be generated for this expression.
We derive the following tableaux[2] in figure 2 for `(e (append Top botmem))`, where we have the following supercombinator definitions:

[2] We made some abbreviations in this tableau to save space.

```
append x y = case x y (append2 y)
append2 ys x xs = Cons x (append xs ys)
e xs = case xs Top econs
econs x xs = if x (e xs) (e xs)
```

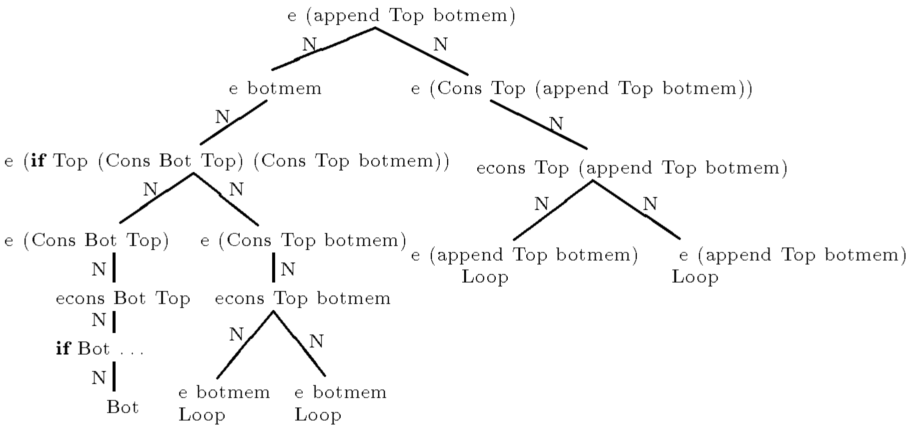


Fig. 2. A closed tableau for (e (append Top botmem))

It is possible to extend the calculus such that it is able to analyse whether a function f evaluates its argument to normal form. We assume that such arguments are data objects, i.e., that they have a type without a function subtype. This can be done by describing the set of all data that are infinite or contain a non-terminating expression somewhere. We write the expression as a union:
botdata $=$ Bot $\cup$ ($\bigcup$ **cons** Top ... botdata ... Top)
where the union is over all non-constant constructors and botdata may be at every argument position of constructors. If f botdata yields a closed tableau, then f evaluates its argument to normal form.
The calculus has to be extended in such a way that unions are handled as branching. However, using these $\Lambda_c^{\#}$-expression, we may run the risk of evaluating ill-typed expressions. This can easily be handled by extending the calculus with a new rule that marks leaves with ill-typed subexpressions of the form $(case_A\ (cons_B\ \ldots)\ \ldots)$, where A and B are different algebraic type names, or of the form $(f\ e_1\ \ldots\ e_n)$, where f is a built-in function, as ill-tyed. The definition of a closed tableau is then extended by permitting also ill-typed leaves.

6 Implementation of Abstract Reduction

An implementation of abstract reduction based on lazy evaluation rather than on eager evaluation has been accomplished ([Sch94, Sch95]). The implementation is done completely in Haskell and analyses a functional core language.

It is realised via a newly developed abstract machine, the $G^{\#}$-machine, which extends the G-machine by the handling of abstract values: `Top`, `Bot` and `Union`. These unions can be viewed as a representation of the different possibilities that a reduction can take. The control is adapted to this insofar as the different possibilities are further reduced in a (breadth first) fair fashion. Similar to the G-machine, the source gets compiled into $G^{\#}$-code. The subsequent interpretation of $G^{\#}$-code then yields strictness information.
The implementation, currently version 0.1.2, is freely available by anonymous ftp from ftp.uni-frankfurt.de in the directory /pub/dist/kist/functional/abs-g.

6.1 Results

Several tests were performed on a Sun SPARCstation 10/41 with 64MB of RAM running Solaris 2.4. The analyser itself was compiled by the Glasgow Haskell Compiler version 0.24. The Core language files for the tests were obtained by running the Gofer compiler `gofc` with a `-D` option followed by some manual touch-up to meet our syntax. A first test was analysing a program script (`t10.core`) entailing Core language versions of parts taken from the analyser itself, namely from the compiler to $G^{\#}$-code, the lexer and some auxiliary functions plus the Core language version of the Gofer [Jon94] `standard.prelude` with exception of floating point functions. In this test 287 functions with a total of 526 arguments are analysed for their strictness. To our knowledge the analyser found all strictness information in this script. This was achieved in 11 seconds of user time. This time includes the times for parsing the Core language script which a real compiler will already have done.
A second test involved a Core language version of the pseudoknot benchmark program [HtFeA+94] plus the Core language version of the `standard.prelude`. This program constitutes a widely used benchmark for functional language compiler technology. It is a real world application from molecular biology which does not depend on laziness for correct behaviour. For this test all floating point numbers were approximated with $\top^{\#}$, as the $G^{\#}$-machine does not support floating point numbers. In this test 248 functions are analysed with a total of 493 arguments. To our knowledge all strict arguments of functions were found.
For a third test we analysed all of the contexts which can arise in the `standard.prelude` with respect to Wadler's 4-point-domain except for the `zip` functions with arity above 2. That is for every list valued argument we tested all the combinations of the values $\top\!\!\in$, $\bot\!\!\in$ and ∞. If the result is a list as well we combine these with the evaluators $\xi_\bot$, $\xi_{\bot\in}$ and ξ_∞. Thus for `append` 27 context analyses are performed. Here 596 functions are analysed with a total of 767 arguments. 435 functions thereof are constructed to perform context analysis.

6.2 Evalutation of the implementation

The main parameter for limiting the resources of the analyser is the number of open redexes. We found that for our input files no additional strict arguments are found once 12 or more open redexes are allowed. From this point on the running

time remains constant. For the input files prepared for context analyses, on the other hand, we see a continuing increase of running time as we increase the number of open redexes allowed. The reason is that the analyser does not detect all of the looping reductions.
Although the implementation favors ease of understanding over efficiency, it proves that abstract reduction with reduction path analysis performs well both in terms of precision and efficency. Thus it is fit for every-day strictness analysis. Furthermore it finds strictness information which implementations of other methods do not find. It is possible to optimize our implementation in several respects, an important optimization may be to compile $G^{\#}$-machine instructions instead of interpreting them. Cautiously estimating, it should not be very difficult to cut down the running time by some factor greater than 2. We are quite positive that it can be tailored to be incorporated in any Haskell compiler and are currently exploring the benefits of this strictness analyser within a compiler.

6.3 Correctness of the implementation

In our implementation, branching is represented as a `Union` node. Furthermore, the expressions are represented as a graph in order to support sharing and graph reduction. The computational history of the leaves is also memorized. The control is roughly as follows:
First the current `Union` is simplified with respect to the ordering $\leq_\alpha$, i.e., only the maximal elements are retained. Then one lazy reduction step is performed for every expression in the union and a new union is created, which may contain more elements. The original terms in the union will be memorized as they stand (without further reduction possibility). Sharing and graph reduction only affect the expressions in the new union. The last step in this cycle is loop detection using the computational history and the ordering.
Now let us analyse the translation of these steps into operations on the dag-tableaux.

- Simplifying the union is the operation of merging nodes, where this operation is restricted to operate only on leaf nodes.
- Translating δ-reduction has to take sharing of expressions into account. Lazily Reducing an expression in a union and updating it amounts to connecting the directly generated sons to their father by an N-edge in a dag-tableaux. Sharing can be simulated by non-necessary reductions on other leaves.
- Loop-detection in the tableau is exactly the same as in the implementation, since comparing the expression in the union with the frozen expressions in the history correspond to comparing the expressions at the leaves with expressions at inner nodes. The implementations guarantees that there is an N-edge between the looping nodes and the leaves.

Thus the calculus **STAR-XLD** corresponds to our implementation, hence the correctness and soundness of this calculus holds for the implementation, too.

The reason for reducing unions breadth-first instead of the depth-first is that lazy depth-first reduction fails to find strictness in such cases as the length example (see section 4.5). Lazy breadth-first reduction together with sharing are able to detect the loop. Nöcker's implementation uses reduction which needs to be strict at times in order to cope with this problem.

7 Conclusion

We have presented a new account of abstract reduction based on tableaux calculus and provide a simple and comprehensive proof for the soundness of different variants. We also demonstrated that an implementation based on this ideas performs well, appears to be comparable to Nöcker's implementation and as far as we can see, is superior to abstract analysis based on abstract interpretation in terms of efficiency and the amount of strictness information obtained.
The new proof method based on tableaux provides a new view on the implementations of abstract reduction and enhances their power. This makes for the time being abstract reduction the tool of choice for strictness analysis.
We hope that implementations of lazy functional languages will benefit from these results in the near future.

References

[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.

[HB93] Denis B Howe and Geoffrey L Burn. Using strictness in the STG machine. In John T O'Donnel and Kevin Hammond, editors, *Functional Programming*, Workshops in Computingt, pages 127–137. Springer, 1993.

[HtFeA+94] P. H. Hartel, M. Feeley, M. Alt, L. Augustsson, P. Baumann, M. Beemster, E. Chailloux, C. H. Flood, W. Grieskamp, J. H. G. van Groningen, K. Hammond, B. Hausman, M.Y. Ivory, P. Lee, X. Leroy, S: Loosemore, N. Röjemo, M. Serrano, J.-P. Talpin, J. Thackray, P. Weiss, and P. Wentworth. Pseudoknot: a float-intensive benchmark for functional compilers. In John Glauert, editor, *Implementation of Functional Languages '94*, 1994. Draft.

[JHR94] Kristian Damm Jensen, Peter Hjæresen, and Mads Rosendahl. Efficient strictness analysis of Haskell. In Baudouin Le Charlier, editor, *Static Analysis*, number 864 in Lecture Notes in Computer Science, pages 346–362. Springer, 1994.

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

[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.

[Nöc92] Eric Nöcker. Strictness analysis by abstract reduction in orthogonal term rewriting systems. Technical Report 92-31, University of Nijmegen, Department of Computer Science, 1992.

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

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

[PJL91a] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 636–666. Springer, 1991.

[PJL91b] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages: a Tutorial.* Prentice-Hall International, London, 1991.

[PJS94] Simon L. Peyton Jones and André Santos. Compilation by transformation in the Glasgow Haskell Compiler. In *Functional Programming, Glasgow 1994*, Workshops in Computing, pages 184–204. Springer, 1994.

[PvE93] Rinus Plasmeijer and Marko van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, Workingham, 1993.

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

[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.

[vEGHN93] M. van Eekelen, E. Goubault, C.L. Hankin, and E. Nöcker. Abstract reduction: Towards a theory via abstract interpretation. In M.R. Sleep, M.J. Plasmeijer, and M.C.J.D. van Eekelen, editors, *Term Graph Rewriting - Theory and Practice*, chapter 9. Wiley, Chichester, 1993.

[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.

This article was processed using the LaTeX macro package with LLNCS style