Strictness Analysis using Abstract Reduction

Eric Nöcker
University of Nijmegen *

November 4, 1993

Abstract

In this paper we present a new and general strictness analysis technique for lazy functional languages. In contrast to many other methods, this method is practically usable. This is shown with results of a real implementation. The key idea is the use of an abstract domain of which the elements represent various kinds of sets of concrete values. Reduction as well as pattern matching in the abstract domain mimics reduction and pattern matching in the concrete domain in a very natural way. With this abstract domain various kinds of strictness analysis are possible. In particular, higher order functions and general data types (tuples and lists, but also user defined data types) fit perfectly well in this mechanism. In contrast to methods based on abstract interpretation, the abstract domain is infinite. Recursive functions are handled by a technique called reduction path analysis. Complicated and expensive fixed point derivation techniques are not needed. The implementation in the Concurrent Clean system shows that the analyser is very fast and that it finds much strictness information.

1. Introduction

During the last decade much research has been done in the area of analysing techniques for the compilation of lazy functional languages. Much of this research concerns strictness analysis. However, despite of the amount of theoretical results, the practical results of strictness analysis techniques were rather disappointing. Most effort has been spent on methods based on abstract interpretation. Based on a finite abstract domain a nonstandard semantics of the language is defined. One of the earliest examples is the work of Mycroft (Mycroft (1981)). He defines an abstract domain consisting of only two elements, representing resp. all nonterminating expressions and all possible expressions. Strictness analysis is applied to first order, flat functions. More complicated examples are strictness analysis for higher order functions (e.g. Abramsky (1985), Burn *et al.* (1985)), and for nonflat domains (e.g. Wadler & Hughes (1987)). These techniques reveal the main disadvantage of such methods. For higher order functions or more complex forms of strictness analysis, the abstract domains grow and can become very complicated. The fixed point analysis, needed for deriving correct results, is of exponential nature, and prohibits usable implementations. In this paper we will present a new strictness analysis method. The interesting point is that this method is implementable, as will be shown with concrete results of a real implementation. As a consequence, the emphasis of this paper will be on the presentation of those practical results. Of the theoretical background we will only give a short overview. The details can be found in Nöcker (1992), and will be published in a separate paper.

The main points of our method are the following:

- We use *infinite* domains. A program, with function and constructor definitions, defines an abstract domain. The elements of this abstract domain are *graphs*, built from a set of abstract symbols that is directly derived from the set of concrete symbols. Abstract graphs can represent various sources of nontermination, for instance the value that represents all infinite lists. Such values can be constructed within the domain itself: no new domain definition is required. This means that it is possible to derive various kinds of strictness information within one and the same system.

*Faculty of Mathematics and Computer Science, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands E-mail: `clean@cs.kun.nl`

- The reduction in the abstract domain is directly related to the corresponding reductions in the concrete domain: *abstract reduction*. We define a general (and undecidable) form of abstract reduction. Several approximations of this general form are possible.
- An abstract reduction sequence does not need to terminate. However, there are several methods to cut off infinite reduction sequences. We will present a technique called *reduction path analysis*. By analysing reduction sequences, it is often possible to derive quite precise information. In other cases, results will have to be approximated.

We note that both abstract reduction and reduction path analysis are approximation techniques. However, they can be made very precise. This can be achieved, for example, by clever implementations, or by giving the analyser more time and memory to run in.

A strictness analyser based on abstract reduction has been implemented as part of the Concurrent Clean system (Plasmeijer *et al.* (1991)). It did turn out that abstract reduction combined with reduction path analysis is well to implement. Furthermore, the results of the strictness analysis are very good. This holds both for the obtained results as for the time and memory behaviour of the strictness analyser.

1.1. The Language

We will assume we have a very simple lazy functional language. Each function has one or more alternatives which are distinguished by patterns on the left-hand side of the definition. Function and constructor symbols start with uppercase characters, this in contrast to variables. Constructor symbols are symbols for which no function definition exists. In this paper we will assume that function symbols cannot occur in patterns (so-called function-constructor systems). Furthermore, we assume some standard predefined types, such as INT and BOOL and their operators. Two examples are:

```
Map f (Cons x xs) = Cons (f x) (Map f xs)
Map f Nil         = Nil

Fib 0             = 1
Fib 1             = 1
Fib n             = + (Fib (- n 1)) (Fib (- n 2))
```

The reduction strategy used is the so-called *functional strategy* (Plasmeijer & Eekelen (1993), Toyama *et al.* (1991)). Basically, the evaluation order is determined by a normal order strategy. It deviates at points where pattern matching is needed: before matching, a value has to be evaluated to head normal form. Patterns in a function definition are matched from left to right, and from top to bottom. Similar strategies are used in many lazy functional languages.

1.2. Strictness Properties

It is a well-known fact that strictness information is important for efficient implementations of lazy functional languages. Of course, it goes beyond the scope of this paper to treat how strictness information can be used. What is important is that an implementation might need several forms of strictness information. The simplest form is head normal form strictness. A function is said to be *strict* in an argument if for any application of that function it is safe to evaluate that argument to head normal form before doing the application itself. That is, a function F of arity n is *strict* in its i^{th} argument iff:

$$F\,x_1, \ldots, x_{i-1}, \perp, x_{i+1}, \ldots, x_n = \perp$$

for all possible terms $x_{j,j \neq i}$, and with $\perp$ representing nontermination or undefinedness. This kind of strictness defines when an argument can be evaluated to head normal form safely. However, there are also other ways to deviate from the standard reduction order. Consider, for instance, the functions:

```
Head (Cons a b)   = a

Length (Cons a b) = ++ (Length b)
Length Nil        = 0
```

Cons Bot Top	$\{\text{Cons} \perp \text{g} \mid \text{g} \in \mathcal{G}\} \cup \{\perp\}$
$\langle$Cons Bot Top,Nil$\rangle$	$\{\text{Cons} \perp \text{g} \mid \text{g} \in \mathcal{G}\} \cup \{\text{Nil}, \perp\}$
inf: Cons Top inf	*all infinite lists*
topmem: $\langle$Cons Top topmem,Nil$\rangle$	*all lists*
botmem: $\langle$Cons Top botmem, Cons Bot topmem$\rangle$	*all lists with at least one $\perp$ element*

Table 1: Examples of abstract graphs and corresponding concrete sets

The function Head is strict in its argument, but also in the (sub)argument a: whenever Head is called with a term that reduces to a Cons node, it is safe to evaluate the head of that Cons node, before applying Head (this might not seem to be a very useful kind of strictness information, however, for other datatypes, like tuples, it might be). Another kind of strictness is shown by the Length function: it is always safe to evaluate the spine of the list argument (which means: all Cons nodes can safely be evaluated, but none of the arguments), before applying Length. This last kind of strictness is called *spine strictness*. Such list evaluations are well-known (Burn (1987)).

To derive such strictness information it is important that it is possible to represent properties (or sets) like "all infinite lists". In the following, we will show that in the abstract reduction framework various kinds of strictness analysis are possible.

2. Principles of Abstract Reduction

In this section we will outline the fundamentals of abstract reduction. In Nöcker (1992) abstract reduction is applied in orthogonal term rewriting systems. Our language is certainly no orthogonal term rewriting system. To some extent however, it has similar properties. In fact, we can consider the reduction strategy as a mean to force a kind of orthogonality. Anyway, we strongly believe that the results of Nöcker (1992) can be extended to functional languages.

The goal of abstract reduction is to mimic concrete reduction as precise as possible. That is, abstractions or approximations are delayed as much as possible. Consider for instance the function definition:

```
F x y = Cons x y
```

If we assume that, as usual, the abstract value Bot corresponds to all nonterminating expressions, and Top to all possible expressions, then we would like that the following abstract reduction step is possible:

$$\mathsf{F}^{\#}\ \mathsf{Bot}\ \mathsf{Top} \rightarrow_{\alpha} \mathsf{Cons}^{\#}\ \mathsf{Bot}\ \mathsf{Top}$$

This decision leads to an infinite domain of abstract values, since the domain must include all possible abstract results. The abstract reduction system must be able to handle all these elements properly. We note that in many systems the new value of the above example would have been approximated.

In the remainder of this section we first describe the construction of the abstract domain. Thereafter, we will give a precise relationship between reduction in the concrete and abstract domain.

2.1. The Abstract Domain

Let $\mathcal{S}$ be the set of symbols associated to a particular set of functions and constructors, and $\mathcal{S}_\perp = \mathcal{S} \cup \{\perp\}$. Let $\mathcal{G}$ be the set of *graphs* over $\mathcal{S}_\perp$. Then, the set of abstract symbols $\mathcal{S}^{\#}$ is defined as follows:

$$\mathcal{S}^{\#} \ = \{F^{\#} \ \mid F \in \mathcal{S}\} \cup \{\text{Bot}, \text{Top}, \text{Union}\}$$

For each symbol in $\mathcal{S}$ an equivalent symbol in $\mathcal{S}^{\#}$ exists. For convenience however, we will omit the #'s if they are of no importance. The symbols Bot, Top and Union are new and will have a special meaning. The symbols Bot and Top have arity zero, whereas Union can be used with an arbitrary number of arguments (in fact, for each arity a special Union symbol can be defined). Elements of the abstract domain $\mathcal{AG}$ are *graphs* over $\mathcal{S}^{\#}$. For convenience, we will use a shorthand notation for Unions:

$$\text{Union } x_1, \ldots, x_n \quad = \quad \langle x_1, \ldots, x_n \rangle$$

The further notation of graphs is straightforward, see table 1 for examples. Sharing is expressed by using node identifiers (the defining occurrence is followed by a colon). Values of the abstract domain $\mathcal{AG}$ correspond with sets of values in $\mathcal{G}$. To some extent, a graph of $\mathcal{AG}$ can be considered as a *denotation* for a subset of $\mathcal{G}$. This relationship is defined by the concretisation function γ: $\mathcal{AG} \to \mathcal{P}(\mathcal{G})$. We first define this function for tree-like graphs (γ_t: $\mathcal{AG}_t \to \mathcal{P}(\mathcal{G}_t)$):

$$\begin{array}{lcl}
\gamma_t[\text{Bot}] & = & \{\perp\} \\
\gamma_t[\text{Top}] & = & \mathcal{G}_t \\
\gamma_t[\text{F } x_1, \ldots, x_n] & = & \{\text{F } t_1, \ldots, t_n \mid t_i \in \gamma_t[x_i], 1 \le i \le n\} \cup \{\perp\} \\
\gamma_t[\langle x_1, \ldots, x_n \rangle] & = & \cup_{1 \le i \le n}(\gamma_t[x_i])
\end{array}$$

Now, we define $\gamma[g] = \{h \in \mathcal{G} \mid U(h) \in \gamma_t[U(g)]\}$, where $U(g)$ is the unraveling of a graph g. This domain is no complete lattice. We refer to Goubault & Hankin (1991) for a more detailed discussion of this domain.

The intuitive meanings of Bot and Top are clear: they represent respectively all nonterminating expressions of $\mathcal{G}$ and all expressions of $\mathcal{G}$. The Union construct is more complex. To some extent, it is similar to the set union operation. However, values in $\mathcal{AG}$ are graphs, thus shared and cyclic structures are allowed (giving possibly cyclic set definitions). We prefer a more operational viewpoint. The Union construct can be used to create several special subsets of $\mathcal{G}$. In the following we will see various examples of this.

The ordering operator $\le$ over $\mathcal{AG}$ is defined as: $t_1 \le t_2 \Leftrightarrow \gamma[t_1] \subseteq \gamma[t_2]$. The Union construct introduces equivalences in $\mathcal{AG}$ ($t_1 \equiv t_2 \Leftrightarrow [t_1 \le t_2 \wedge t_2 \le t_1]$). Some important examples are:

$$\begin{array}{lcl}
\langle x_1, \ldots, x_n \rangle & \equiv & \langle \pi(x_1, \ldots, x_n) \rangle \\
 & & \text{with } \pi \text{ any permutation} \\
\langle x, x_1, \ldots, x_n \rangle & \equiv & \langle x_1, \ldots, x_n \rangle \\
 & & \text{if } x \le x_i \text{ for some } 1 \le i \le n \\
C[\langle x_1, \ldots, x_n \rangle] & \equiv & \langle C[x_1], \ldots, C[x_n] \rangle \\
x : \langle x, x_1, \ldots, x_n \rangle & \equiv & \langle x_1, \ldots, x_n \rangle \\
\langle x \rangle & \equiv & x
\end{array}$$

In table 1 some examples of abstract values for lists are given, and their corresponding sets of the concrete domain. The last three values are similar to the values of Wadler's four point domain (Wadler (1987)). In these examples the possibility of sharing is essential.

2.2. Concrete Reduction

In order to define a relationship between abstract and concrete reductions, we first need a precise definition of what concrete reduction is. Since the abstract reduction mechanism is based on a (graph or term) rewriting mechanism we will also assume a rewriting semantics for the concrete language.

First, we define the concrete reduction relation $\to$ over $\mathcal{G}$. It is directly related to reduction in our simple language: $t \to_r s$ if t rewrites in one step to s. We extend this relation with so-called $\perp$-reductions:

$$C[t] \to_\perp C[\perp] \text{ if t has no head normal form w.r.t. } \to_r$$

where $C[t]$ denotes a context containing a term t. We write $\to = \to_r \cup \to_\perp$ for the extended reduction relation. We note that it is important that this reduction relation is confluent, which is the case for our simple system. The transitive closure of $\to$ will be written as $\twoheadrightarrow$.

Another important notion is *neededness*. A subterm s of t is *needed* if in every reduction sequence leading to a head normal form s or one of its residuals has been reduced. A reduction step is a needed reduction step if a needed redex is rewritten. The reduction relation $\to_n$ is defined as $t \to_n s$ iff $t \twoheadrightarrow s$ and at least one of the reductions is a needed one. We note that $\to_n$ is normalising, i.e. it leads in a finite number of steps to a head normal form (if existing). If a term t has no head normal form then $t \to_n \perp$.

2.3. Abstract Reduction

The main point of abstract reduction is that it should mimic at least some needed reductions of the corresponding concrete reductions. We define a general abstract reduction relation $\to_\alpha$ over $\mathcal{AG}$ as follows (for abstract values t_1 and t_2):

$$t_1 \rightarrow_\alpha t_2 \Leftrightarrow \forall g \in \gamma[t_1].\exists h \in \gamma[t_2] \text{ such that } g \rightarrow_n h$$

The background of the neededness conditions is to ensure safety of bot-introduction, a form of reduction path analysis, see section 3.3. In Nöcker (1992) it is shown that this reduction relation has some very nice properties. Consider for some examples of abstract reduction steps the following function:

```
F (Cons a x) y = F x y
F Nil        y = y
```

Then, we might have the following abstract reduction steps:

F Top Bot	$\rightarrow_\alpha$	Top
F Top Bot	$\rightarrow_\alpha$	⟨Bot, F Top Bot⟩
F inf Top	$\rightarrow_\alpha$	F inf Top
F Top Bot	$\rightarrow_\alpha$	Bot
F (Cons (+ 1 2) Nil) (+ 1 2)	$\rightarrow_\alpha$	F (Cons 3 Nil) 3
F (Cons (+ 1 2) Nil) 3	$\not\rightarrow_\alpha$	F (Cons 3 Nil) 3

The first reduction step is an example of an approximation. The following two can be derived by using the definition of the function. The fourth one requires a deeper analysis. Note the difference between the last two lines. The last reduction step is illegal, since it simulates no needed reductions. That is, nonneeded subterms can be reduced only if also some needed terms are rewritten.

Now, we can formulate the head normal form strictness test in the abstract world. A function F of arity n is strict in its i^{th} argument iff (with the Bot argument on position i):

$$\text{F Top}\ldots\text{Top Bot Top}\ldots\text{Top} \rightarrow_\alpha \text{Bot}$$

Another important consequence is that we have a notion of head normal form in the abstract world. An abstract value cannot reduce if it represents (amongst others) a concrete value that is in head normal form. Thus, an abstract value is in head normal form if:

(i) it is Top or Bottom
(ii) its root symbol is a constructor symbol.
(iii) it is a Union, and one of its arguments is in head normal form

3. Computing Abstract Reductions

The general form of abstract reduction as defined in the previous section is far too general to be implemented. In this section we will discuss how we can approximate the general form. We will do this in two steps. We start by defining an *abstract rewriting* mechanism that is directly related to the concrete function definitions, and the applied reduction strategy (in some sense, this can be considered as a kind of symbolic computation). These abstract rewrites are abstract reductions. Thereafter, in section 3.3 and further, we treat reduction path analysis. By analysing abstract reduction sequences it is possible to derive new abstract reductions.

3.1. Abstract Rewriting

We will start our description by considering an abstract expression $t = F^{\#}\ v_1 \ldots v_n$, where F is a function symbol. This term represents a set of values $\gamma[t]$ in $\mathcal{G}$, *all starting with the same function symbol F*. So, all (top-level) terms will rewrite according to this function F. The abstract rewrite step should simulate all those concrete rewrite steps. Although we know according to what function the value should rewrite, there are two problems:

- different expressions in $\gamma[t]$ can rewrite according to different alternatives of F.
- for different expressions in $\gamma[t]$ the functional strategy can force the reduction of different subexpressions.

Now suppose we have a function definition for a function F consisting of k alternatives:

$$\begin{aligned} &\mathsf{F}\ \mathsf{p}_{11}\ldots\mathsf{p}_{1n} = \mathsf{rhs}_1 \\ &\qquad\quad \ldots \\ &\mathsf{F}\ \mathsf{p}_{k1}\ldots\mathsf{p}_{kn} = \mathsf{rhs}_k \end{aligned}$$

We define a corresponding abstract rewrite rule for $F^{\#}$in $\mathcal{AG}$ as follows:

$$\begin{aligned} &\mathsf{F}^{\#}\ \mathsf{p}_{11}\ldots\mathsf{p}_{1n} = \#[\mathsf{rhs}_1] \\ &\qquad\quad \ldots \\ &\mathsf{F}^{\#}\ \mathsf{p}_{k1}\ldots\mathsf{p}_{kn} = \#[\mathsf{rhs}_k] \end{aligned}$$

where the function # just converts the function symbols of an expression to their abstract variants. Suppose the concrete values of t will rewrite according to the alternatives $i_1 \ldots i_m$ only. Then we can define the abstract reduction step Reduce as follows:

$$\begin{aligned} \mathsf{Reduce}\ [\mathsf{F}^{\#}\ \mathsf{v}_1\ldots\mathsf{v}_n] &= \langle \mathsf{Rewrite}_{i_1},\ldots,\mathsf{Rewrite}_{i_m}\rangle \\ \mathsf{Reduce}\ [\langle \mathsf{x}_1,\ldots,\mathsf{x}_n\rangle] &= \langle \mathsf{Reduce}\ [\mathsf{x}_1],\ldots,\mathsf{Reduce}\ [\mathsf{x}_n]\rangle \end{aligned}$$

where Rewrite$_{i_j}$ is the result of an abstract rewrite step according to alternative i_j. However, since pattern matching might force new reductions, it is possible that subterms have to be rewritten first. For reaching a head normal form, reductions have to be done until that is the case. So, we define:

$$\begin{aligned} \mathsf{ReduceHnf}\ [\mathsf{t}] &= \mathsf{t}, \text{ if t in head normal form} \\ &= \mathsf{ReduceHnf}\ (\mathsf{Reduce}\ [\mathsf{t}]), \text{ otherwise} \end{aligned}$$

Rewriting (i.e. binding the variables of a left-hand side) can be done as usual, the only problem is the binding of the variables in the case that a Top or Union value is matched against a complex pattern. In the case of Top it is simple: then all variables in the pattern should be bound to Top. For Union values it becomes complex. If such a value matches in multiple ways, the variables should be bound in different ways. In fact, this case can be considered as a set of rewrites: for each possible match of the Union one rewrite should be performed. Consider the following example:

F (Cons a b) = Tup a b

then the value F Top would rewrite as follows:

F Top $\rightarrow$ Tup Top Top

and the value F $\langle$Cons Top Bot, Cons Bot Top$\rangle$ would rewrite as follows:

F $\langle$Cons Top Bot, Cons Bot Top$\rangle$ $\rightarrow$ $\langle$Tup Top Bot, Tup Bot Top$\rangle$

The above rewrite of the Union argument can be seen as the Union of two rewrites since F $\langle$Cons Top Bot, Cons Bot Top$\rangle$ $\equiv$ $\langle$F (Cons Top Bot), F (Cons Bot Top)$\rangle$. If a term contains more Unions, the rewrite can become rather complex. Lastly, we notice that the result of the rewrite is Bot if no alternatives are applicable (since $\langle\rangle \equiv$ Bot).

3.2. Pattern Matching and the Functional Strategy

We have to construct a matching mechanism that decides what alternatives of the abstract rewrite rule might match. Intuitively, a value v in $\mathcal{AG}$ matches a pattern p if the set of values in $\mathcal{G}$ which is represented by p is contained in the set of values in $\mathcal{G}$ represented by v. Of course, we have to take account of values that have to be reduced before the matching. A match against a variable always succeeds. A matching algorithm for the abstract reduction should be safe, that is, if an abstract match yields false, then also all concrete matches should yield false:

$$\neg Match_{\alpha}(v, \#[p]) \Rightarrow \forall g \in \gamma[v].\neg Match(g, p)$$

where Match and Match$_{\alpha}$ are the concrete and abstract matching algorithms. A safe abstract matching algorithm (of course, the safest is to choose all possible alternatives) is easy to define, see for example table 2. The definition silently assumes that values are reduced to head normal form if necessary. An

$\mathsf{Match}_\alpha(\mathsf{v},\ \mathsf{p})$	$=$	True, if p is a variable
$\mathsf{Match}_\alpha(\mathsf{Top},\ \mathsf{p})$	$=$	True
$\mathsf{Match}_\alpha(\mathsf{Bot},\ \mathsf{p})$	$=$	False, if p is no variable
$\mathsf{Match}_\alpha(\mathsf{C}\ \mathsf{v}_1 \ldots \mathsf{v}_n,\ \mathsf{C'}\ \mathsf{p}_1 \ldots \mathsf{p}_m)$	$=$	False
$\mathsf{Match}_\alpha(\mathsf{C}\ \mathsf{v}_1 \ldots \mathsf{v}_n,\ \mathsf{C}\ \mathsf{p}_1 \ldots \mathsf{p}_n)$	$=$	CombineMatches ($\mathsf{Match}_\alpha(\mathsf{v}_i,\ \mathsf{p}_i)$) for all $1 \leq \mathsf{i} \leq \mathsf{n}$
$\mathsf{Match}_\alpha(\langle \mathsf{v}_1, \ldots, \mathsf{v}_n \rangle,\ \mathsf{p})$	$=$	CombineUnionMatches ($\mathsf{Match}_\alpha(\mathsf{v}_i,\ \mathsf{p})$) for all $1 \leq \mathsf{i} \leq \mathsf{n}$
CombineMatches ($\mathsf{m}_1 \ldots \mathsf{m}_n$)	$=$	False, if m_i = False for some $1 \leq \mathsf{i} \leq \mathsf{n}$
CombineUnionMatches ($\mathsf{m}_1 \ldots \mathsf{m}_n$)	$=$	True, if m_i = True for some $1 \leq \mathsf{i} \leq \mathsf{n}$

Table 2: A simple matching algorithm

alternative of a function definition for $F^{\#}$ matches if all arguments match the corresponding patterns. This matching algorithm can be used for choosing the alternatives for the rewrites. However, it is rather inaccurate, since it does not take the peculiarities of the functional strategy into account. It takes no account of the order in which alternatives are tried, nor of the fact that matching doesn't need to terminate. This is illustrated by the following example:

```
F Nil = Nil
F x   = Cons x Nil
```

The value F Nil would rewrite as follows (since both alternatives match):

F Nil $\rightarrow$ $\langle$Nil, Cons Nil Nil$\rangle$

And the value F Bot would rewrite as follows (since the second alternative matches):

F Bot $\rightarrow$ Cons Bot Nil

$\mathsf{Match}_f(\mathsf{v},\ \mathsf{p})$	$=$	TotalMatch, if p is a variable
$\mathsf{Match}_f(\mathsf{Top},\ \mathsf{p})$	$=$	PartialMatch, if p is no variable
$\mathsf{Match}_f(\mathsf{Bot},\ \mathsf{p})$	$=$	BotMatch, if p is no variable
$\mathsf{Match}_f(\mathsf{C}\ \mathsf{v}_1 \ldots \mathsf{v}_n,\ \mathsf{C'}\ \mathsf{p}_1 \ldots \mathsf{p}_n)$	$=$	NoMatch
$\mathsf{Match}_f(\mathsf{C}\ \mathsf{v}_1 \ldots \mathsf{v}_n,\ \mathsf{C}\ \mathsf{p}_1 \ldots \mathsf{p}_n)$	$=$	CombineMatches ($\mathsf{Match}_f(\mathsf{v}_i,\ \mathsf{p}_i)$) for all $1 \leq \mathsf{i} \leq \mathsf{n}$
$\mathsf{Match}_f(\langle \mathsf{v}_1, \ldots, \mathsf{v}_n \rangle,\ \mathsf{p})$	$=$	CombineUnionMatches ($\mathsf{Match}_f(\mathsf{v}_i,\ \mathsf{p})$) for all $1 \leq \mathsf{i} \leq \mathsf{n}$
CombineMatches ($\mathsf{m}_1 \ldots \mathsf{m}_n$)	$=$	TotalMatch, if m_i = TotalMatch for all i
	$=$	NoMatch, if m_i = NoMatch for some i & m_j = TotalMatch, for all j < i
	$=$	BotMatch, if m_i = BotMatch for some i & m_j= TotalMatch, for all j < i
	$=$	PartialMatch, otherwise
CombineUnionMatches ($\mathsf{m}_1 \ldots \mathsf{m}_n$)	$=$	TotalMatch, if m_i = TotalMatch for all i
	$=$	NoMatch, if m_i = NoMatch for all i
	$=$	BotMatch, if m_i = BotMatch for all i
	$=$	PartialMatch, otherwise

Table 3: A matching algorithm for the functional strategy

Both rewrites are, though safe, too pessimistic. In the first case in the concrete world always the first alternative would have matched. In the second example the pattern matching would not succeed because the argument has no normal form (the result of F $\perp$ would be $\perp$). This behaviour is the result of the functional strategy. In order to simulate this strategy more precisely, the matching mechanism has to be made smarter. We will start by redefining abstract matching in such a way that it returns more information. We observe that beside the usual two outcomes (Match or NoMatch) two other results are needed. The first, corresponding to the first example, will be the TotalMatch: a value matches a pattern totally. If matching an alternative yields a TotalMatch, then the following ones are of no importance. The second one corresponds to the second example and will be called a BotMatch. Also in this case the following alternatives don't need to be examined. The two possible results of the original Match_α we will call PartialMatch (because it is no TotalMatch), and NoMatch. With the functions CombineMatches and CombineUnionMatches matches of multiple patterns are combined. Also in this case, we can take into account that matching proceeds from left to right. For example, if a match results in a NoMatch then the result of the whole alternative will be NoMatch if all the previous matches resulted in a TotalMatch. The definitions of these functions are derived easily, see table 3. Consider, for example, the pattern:

Cons Nil (Cons a b)

then the following values would match as follows:

Cons Top Top gives PartialMatch
Cons Nil Nil gives NoMatch
Cons Top Bot gives PartialMatch
Cons Nil Bot gives BotMatch

Also the results of the matches of the patterns in an alternative have to be combined in this way. We note the meaning of PartialMatch: it indicates that subsequent alternatives have to be examined. But, an alternative that yields PartialMatch doesn't need to result in a rewrite according to that alternative, since its match might also lead to a nontermination.

The new match function is quite precise. However, it can still be made more accurate. The algorithm becomes much more complex in these case. An example of loss of information is the following. Consider the function:

F Nil (Cons a b) = ...
F x y = ...

Then F Top Bot gives a PartialMatch on the first alternative. So, both alternatives are chosen. However, it is obvious that in this case only the second one is needed: if this value would ever match the first argument pattern (Nil), then it would not terminate.

3.3. Reduction Path Analysis

Both matching and reduction might cause a nontermination. Matching might not terminate if either the pattern or the value contains a cycle (see the definitions for the Match). We don't allow cycles in patterns, so they also will not arise in the abstract world. Cycles in values are possible. They are not dangerous as long as there are no cycles via Union's (then the matching will terminate because patterns are finite, and the only way of not entering the elements of a pattern is via a Union value). For example, Match_α (x: $\langle$x, x$\rangle$, p) will never terminate. This can be solved by requiring that none of the elements of a Union is a Union. This can be assured by applying the equivalences for Unions given earlier. Non terminating reductions will be more common. Fortunately, it is always safe to replace a term (or subterm) by Top in a reduction sequence (since $t \rightarrow_\alpha$ Top for any term t not in head normal form). An implementation might do this because of several reasons, for instance:

- the depth of the recursion due to calls of the Reduce function exceeds some maximum.
- the execution time of the reduction of a term exceeds some maximum.
- the memory requirements of the the reduction of a term exceeds some maximum.

Though reduction in general doesn't need to terminate, we can force termination by applying one or more of these strategies. However, it is not very accurate. That is, it terminates by approximating something with Top, but it throws away more information than is necessary. For example, consider the function:

F x = F x

then reduction would give (some reductions followed by an approximation):

F Bot $\rightarrow \ldots \rightarrow$ F Bot $\rightarrow$ Top

What is the case? The nonterminating sequence of abstract reductions models a set of reduction sequences in the concrete domain which (also) do not terminate. So, one would expect that, if an abstract term reduces to a term in which itself appears on a needed position (and again and again...), the corresponding concrete terms don't terminate either. This is expressed in the following proposition (*bot-introduction*):

$$t \twoheadrightarrow_\alpha C[t] \wedge t \text{ is needed} \Rightarrow t \rightarrow_\alpha \text{Bot}$$

This rule is also an abstract reduction step. For a proof of this proposition for the case of orthogonal term rewriting systems we refer to Nöcker (1992).

3.4. Neededness

The main problem in applying this method is to decide when a subterm of an abstract term is needed. Neededness in abstract terms is directly related to neededness in concrete terms: a subterm s of $t \in \mathcal{AG}$ is needed if *all corresponding* subterms q of s in $\gamma[t]$ are needed. The notion of corresponding subterms is rather difficult to define, and we will omit it here.

Sometimes, it is easy to see whether a subterm of a term of $\mathcal{AG}$ is needed or not. If the term doesn't contain a Union, the needed subterm should be on a *strict* position. Strict positions in an expression can be identified by using strictness information from the functions that are used in that expression. This means that the order in which functions are analysed is of importance (for which we need some kind of dependency analysis).

If the term contains a Union, the subterm should be needed in any part of the Union. So, we can define the predicate IsNeeded:

IsNeeded [t,Top]	$=$	False
IsNeeded [t,Bot]	$=$	True
IsNeeded [t, t]	$=$	True
IsNeeded [t,$\mathsf{F}^{\#}$ $\mathsf{v}_1,\ldots,\mathsf{v}_n$]	$=$	IsNeeded [t,v_i] for a v_i on a strict position
IsNeeded [t,$\langle \mathsf{v}_1,\ldots,\mathsf{v}_n \rangle$]	$=$	IsNeeded [t,v_i] for all $1 \leq i \leq n$

The intuitive meaning of this is as follows. A Union is normally created if multiple alternatives for a rewrite rule are chosen. In fact, each such alternative defines a reduction path. Therefore, such a term should be present in each path. For example (the integer operations are strict in their arguments):

$\langle$+ Top t, - t Top$\rangle$	t is needed
$\langle$+ Top Top, - t Top$\rangle$	t is not needed
$\langle$t, - t Top$\rangle$	t is needed
$\langle$Cons t Bot, - t Top$\rangle$	t is not needed

3.5. Other Forms of Reduction Path Analysis

There are two other interesting forms of the bot-introduction. The first one is a generalisation of bot-introduction:

$$t \twoheadrightarrow_\alpha C[t'] \wedge t' \leq t \wedge t' \text{ is needed} \Rightarrow t \rightarrow_\alpha \text{Bot}$$

Because $t' \leq t$, t' can be replaced by t in $C[]$. This would give the earlier proposition. Finally, there is also a rule, called *cycle-introduction*, if the term t is *not* needed in each path:

$$t \twoheadrightarrow_\alpha C[t'] \wedge t' \leq t \Rightarrow t \rightarrow_\alpha a\colon C[t' := a]$$

That is, t' is replaced by a pointer to the root: a cycle is created (we note that for proving correctness of this rule, we have to extend the reduction relation $\rightarrow_n$ in the concrete domain to reduction sequences of infinite length, see Nöcker (1992)) In section 4, examples of these kinds of reduction path analysis will be given.

Of course, there are still other techniques imaginable. Any analysis that gives us safe abstract reduction steps can be considered as a kind of reduction path analysis. It is, for example, even possible to include other kinds of strictness analysis.

3.6. A Decidable Less-equal

For reduction path analysis a less-equal operator is required. A simple less-equal operator can easily be defined:

$$
\begin{array}{lcll}
\mathsf{Bot} & \leq & \mathsf{t} & \\
\mathsf{t} & \leq & \mathsf{Top} & \\
\mathsf{t} & \leq & \mathsf{t} & \\
\mathsf{F}\ \mathsf{x}_1,\ldots,\mathsf{x}_n & \leq & \mathsf{F}\ \mathsf{y}_1,\ldots,\mathsf{y}_n & \text{if } \mathsf{x}_i \leq \mathsf{y}_i \text{ for all } 1 \leq i \leq n \\
\mathsf{t} & \leq & \langle \mathsf{x}_1,\ldots,\mathsf{x}_n \rangle & \text{if } \mathsf{t} \leq \mathsf{x}_i \text{ for some } 1 \leq i \leq n
\end{array}
$$

The main disadvantage of this operator is that it cannot handle cyclic graphs. Without precautions (e.g. marking) it even would not terminate. Cyclic structures are important for complex forms of strictness analysis, so the operator should be able to cope with them. We can improve its behaviour by adding assumptions. A new argument of $\leq$ is a set of assumptions, containing pairs of values (i.e. graph pointers) under examination:

$$
\begin{array}{ll}
\leq(\mathsf{t},\ \mathsf{t'},\ \mathsf{P}) & \text{if } (\mathsf{t},\ \mathsf{t'}) \text{ in } \mathsf{P} \\
\leq(\mathsf{t}\!:\ \mathsf{F}\ \mathsf{x}_1\ldots\mathsf{x}_n,\ \mathsf{t'}\!:\ \mathsf{F}\ \mathsf{y}_1\ldots\mathsf{y}_n,\ \mathsf{P}) & \text{if } \leq(\mathsf{x}_i,\ \mathsf{y}_i,(\mathsf{t},\ \mathsf{t'})\!:\!\mathsf{P}) \\
 & \text{for all i} \\
\leq(\mathsf{t},\ \mathsf{t'}\!:\ \langle \mathsf{x}_1,\ldots,\mathsf{x}_n \rangle,\ \mathsf{P}) & \text{if } \leq(\mathsf{t},\ \mathsf{x}_i,\ (\mathsf{t},\ \mathsf{t'})\!:\!\mathsf{P}) \\
 & \text{for some i}
\end{array}
$$

We will illustrate this less-equal with an easy example. Consider the following two graphs with root nodes a and x:

```
a:Cons Top b        x: Cons Top x
b:Cons Top b
```

Both graphs represent all infinite lists. It is easy to verify that the algorithm finds that a $\leq$ x and x $\leq$ a (and also that inf $\leq$ botmem $\leq$ topmem).

3.7. Implementing Reduction Path Analysis

If an implementation of reduction path analysis would be directly based on the ideas presented above, the performance would be very bad. Each time a reduction has been done, trees of Union values would have to be traversed in order to find needed terms that can be replaced by Bot. Notice that it is also difficult to derive whether a term is needed in all possible paths. Therefore, we will describe a method with which reduction path analysis can be implemented relatively efficiently.

As in a real reduction machine, also in abstract reduction root updates should be postponed as much as possible. If for a function call several alternatives are applicable, the reduction has to be split up in several paths. Each path is reduced separately. The building of the union of rewrites is postponed until all reduction paths have delivered a head normal form result. Consider the function:

```
F Nil y          = G y
F (Cons a x) y = H x y
```

then F Top Bot would reduce as follows:

$$\mathsf{F\ Top\ Bot} \rightarrow \langle \mathsf{G\ Bot},\ \mathsf{H\ Top\ Bot} \rangle$$

however, in an implementation this union should not be built immediately. First, G Bot and H Top Bot should be reduced. Assuming that they both reduce to Nil, we get:

$$
\begin{array}{rl}
\mathsf{F\ Top\ Bot} & \twoheadrightarrow \langle \mathsf{Nil},\mathsf{Nil} \rangle \\
 & \equiv\ \mathsf{Nil}
\end{array}
$$

So, each reduction can be split up in several subreductions. Each reduction path keeps a list of terms which appeared in the reduction before. If a term has to be reduced that is contained in this list, it is a candidate for a replacement by Bot. Notice that this term is needed in this reduction path, but it doesn't need to be in other ones! Now the term is replaced by an indirection to corresponding term in the list. If the reduction of the term to which the indirection was made is completed, the resulting term has to be scanned for indirections to this term. If all paths contain this indirection, the indirection can be replaced by Bot, otherwise by pointers to the term (to which the indirection points). In order to minimise the search for indirections, terms to which an indirection has been made should be marked as such.

4. Some Elaborated Examples and Practical Results

This section contains a number of examples which illustrate the power and drawbacks of the presented techniques. We will start by giving some preconditions necessary for an implementation. Thereafter, the examples are discussed in increasing complexity. We conclude with a summary of the practical results and the performance figures of the strictness analyser for the examples.

4.1. Approximations of Basic Values

In principle it is possible to take into account all information that is present in a program. For basic values (like integers and booleans) however, this will not be very successful. Because the wide variety and amount of operations on these values the strictness analysis would become very slow if all possible computations would be performed. Therefore, it is generally easier to approximate all basic values by Top.

4.2. Reduction Orders

Generally, abstract terms are evaluated in normal order (or if forced by pattern matching). Sometimes, this is not the best way for finding strictness information. Define:

```
Length (Cons a b) n = Length b (++ n)
Length Nil n        = n
```

Then:

$$\begin{aligned} \mathsf{Length\ Top\ Bot} &\rightarrow \langle \mathsf{Length\ Top\ (++\ Bot)},\ \mathsf{Bot} \rangle \\ &\equiv\ \mathsf{Length\ Top}\ \underline{\mathsf{(++\ Bot)}} \end{aligned}$$

The function Length is strict in its second argument, but this will not be derived. However, if the underlined term would have been reduced, we can infer by reduction path analysis that the result is Bot. As a consequence, it is often better to reduce subterms *eagerly*. This is certainly the case for basic functions.

4.3. Lists and Evaluation Transformers

For lists some special kinds of strictness can be defined (Wadler (1987)). In the current implementation of Clean this would not be useful, because only simple strictness is used in the code generation. However, in implementations of functional languages that use evaluation transformers (Burn (1987)) various kinds of list strictness are used. We will give some examples since it is a good illustration for abstract reduction.

In the evaluation transformer model four kinds of reduction contexts are defined: $\mathcal{E}_0$, which means no reduction, $\mathcal{E}_1$, which means reduction to head normal form, $\mathcal{E}_2$, which means reduction of the spine of a list, and lastly $\mathcal{E}_3$, which means spine reduction and reduction of the elements of the list to head normal form. Given an application which has to be reduced in a certain context $\mathcal{E}$, the arguments of the application can be reduced in contexts which depend on the context $\mathcal{E}$. Consider for example the append function:

```
Append Nil y        = y
Append (Cons a b) y = Cons a (Append b y)
```

If an application of Append has to be reduced in context $\mathcal{E}_2$ then both arguments of Append can be reduced in context $\mathcal{E}_2$. How can such behaviour be derived with our strictness analysis?

Consider first the simple case of the head normal form context $\mathcal{E}_1$. Let F be a function of arity n, of which we are interested in the i^{th} argument. This argument can be reduced in context $\mathcal{E}_1$ if F is strict in that argument. The argument can be reduced in context $\mathcal{E}_2$ or $\mathcal{E}_3$ if respectively:

F Top... Top inf Top ... Top $\twoheadrightarrow$ Bot
F Top... Top botmem Top ... Top $\twoheadrightarrow$ Bot

where inf and botmem are the special abstract values introduced earlier. With the definitions of inf and botmem also the other evaluation contexts can be handled. For example, we might want to derive how the arguments of a function can be reduced if the elements of the resulting list are all needed (thus context $\mathcal{E}_3$):

F Top... Top Bot Top... Top $\twoheadrightarrow$ botmem
F Top... Top botmem Top... Top $\twoheadrightarrow$ botmem

However, such a test, with as right-hand side a non Bot value, is difficult to perform: abstract reduction continues until a head normal form has been reached. The result of the reduction might still contain unevaluated terms, and might thus be incomparable with botmem. A general normal form reduction is possible, but then applications of bot-introductionmight become dangerous. Another solution is to use functions that reduce the spine, and the spine and elements of a list (call them resp. $\mathcal{E}_2$ and $\mathcal{E}_3$). Such functions that simulate the required context are easy to define. In the previous case we would have to check whether:

$\mathcal{E}_2$ (F Top... Top Bot Top... Top) $\twoheadrightarrow$ Bot
$\mathcal{E}_3$ (F Top... Top botmem Top... Top) $\twoheadrightarrow$ Bot

Ordinary strictness can be described easily: an argument is strict, or it is not. However, strictness information for evaluation transformers is more complicated. So, we adopt the following convention. If the result of a function is a list, the strictness information of an argument is denotated as $[a_1, a_2, a_3]$, where a_i defines how the argument can be reduced in the $\mathcal{E}_i$ context. Otherwise, we simply indicate how the arguments can be evaluated. For example, the strictness properties of the Append and Length function are given as:

Append [1,2,3] [0,2,3]
Length 2 1

4.4. Simple Functions

Consider the following simple functions:

Head (Cons a b) = a
F x = Head (Cons x Nil)

Then it is clear that Head and F are strict in their arguments:

Head Bot $\rightarrow$ Bot

F Bot $\rightarrow$ Head (Cons Bot Nil)
$\rightarrow$ Bot

Most strictness analysers do not detect that F is strict! Information is thrown away, for example by replacing Cons Bot Nil by a value like botmem. Notice that Head is also strict in its subargument a, but not (derivable) in b:

Head (Cons Bot Top) $\rightarrow$ Bot
Head (Cons Top Bot) $\rightarrow$ Top

4.5. Recursive Functions

Consider the following recursive functions:

```
F x y = IF (= y 0)
           x
           (F (−− x) (++ y))

G x   = G (Cons 1 x)
```

We assume a standard rewrite rule for the conditional (of course, it is also possible to define F using pattern matching):

```
IF TRUE  y z = y
IF FALSE y z = z
```

First we will derive strictness for the function F. The second argument (y) is easy, because it is used on a strict position at the right-hand side. The first argument is more difficult. It is used in both parts of the conditional, and once it is used in a recursive call to F. It seems that reduction path analysis will not help: each time F is called, the arguments will be different:

```
F Bot Top → IF (= Top Top) Bot (F (−− Bot) (++ Top))
          → IF Top Bot (F (−− Bot) (++ Top))
          → ⟨Bot, F (−− Bot) (++ Top)⟩
          ≡  F (−− Bot) (++ Top)
```

If we would now reduce the arguments of F first, we would get the starting term, so that we can conclude that the result is Bot. However, also if we continue reducing this result will be found, at least, if we use sharing in the implementation:

```
F (−− Bot) (++ Top)   → IF (= y: (++ Top) Top) x: (−− Bot) (F (−− x) (++ y))
                      → IF (= Top Top) (−− Bot) (F (−− x) (++ Top))
                      → ⟨x: (−− Bot), F (−− x) (++ Top)⟩
                      → ⟨Bot, F (−− Bot) (++ Top)⟩
                      ≡  F (−− Bot) (++ Top)
```

this term we don't need to reduce, because it is equal to a term some reductions before. Thus:

```
F Bot Top → Bot
```

So, the analysis derives that F is strict in its first argument. Standard strictness analysers find easily that F is strict in both arguments. Now let's look at the function G. It is strict in its first argument, because G will never terminate. The analysis goes:

```
G Bot → G (Cons Top Bot)
      → G (Cons Top (Cons Top Bot))
      → G (Cons Top (Cons Top (Cons Top Bot)))
      → ...
```

The reduction will not terminate. Also reduction path analysis will not help (terms become bigger). So, it seems as if no information can be derived. However, let's try the following reduction:

```
G Top → G (Cons Top Top)
```

By reduction path analysis we can immediately derive that the result will be Bot. Since G Bot $\leq$ G Top, we can conclude that also G Bot → Bot.

4.6. List Strictness

Consider the following functions over lists:

```
Append Nil y           = y
Append (Cons a b) y = Cons a (Append b y)

Rev2 Nil               = Nil
Rev2 (Cons a b)       = Append (Rev2 b) (Cons a Nil)

Reverse Nil y          = y
Reverse (Cons a b) y = Reverse b (Cons a y)
```

The derivation of strictness for these list functions is rather complicated. We will only show some example reductions; the full results can be found in table 4.

Except for the function Reverse, the results are optimal. We will show now how the results in the $\mathcal{E}_3$ context are derived. Because of the complexity we will not show all steps. Also the context function $\mathcal{E}_3$ will be omitted. First the first argument of Append:

```
Append botmem topmem   → ⟨Cons Top (Append botmem topmem),
                           Cons Bot (Append topmem topmem)⟩
```

Because of the $\mathcal{E}_3$ context the second part of the union reduces immediately to Bot. The first part reduces (via the $\mathcal{E}_3$ context) to Append botmem topmem, which delivers Bot by reduction path analysis. The second argument of Append:

```
Append topmem botmem   → ⟨botmem,
                           Cons Top (Append topmem botmem)⟩
```

Because of the $\mathcal{E}_3$ context the first part of the union reduces immediately to Bot. The second part reduces (via the $\mathcal{E}_3$ context) to Append topmem botmem, which delivers Bot by reduction path analysis. The results of Append can be used in the function Rev2:

```
Rev2 botmem → ⟨Append (Rev2 botmem) (Cons Top Nil),
               Append (Rev2 topmem) (Cons Bot Nil)⟩
```

Because of the $\mathcal{E}_3$ context the arguments of Append will be reduced in that context also. The first part of the union continues with the reduction of Rev2 botmem which results in Bot. The second part also gives Bot, because the reduction Cons Bot Nil in the $\mathcal{E}_3$ context gives Bot. For the function Reverse (which is a tail recursive, and hence much more efficient, variant of Rev2) not all information can be derived. The first argument of Reverse:

```
Reverse botmem topmem   → ⟨Reverse topmem (Cons Bot topmem),
                            Reverse botmem (Cons Top topmem)⟩
```

The second part of the union is less than the original term. The first part reduces further:

```
Reverse topmem (Cons Bot topmem)   → ⟨Cons Bot topmem,
                                      Reverse topmem (Cons Top (Cons Bot topmem))⟩
```

Because of the context the first part gives Bot. However, the second part will reduce further and further. It will not be discovered that the arguments of Reverse are instances of botmem.

4.7. Higher Order Functions

Higher order functions are no problem for our analyser. Of course, we have to allow currying in the abstract domain, but the abstract reduction mechanism stays the same. Consider the following definition and applications of the higher order function Foldr:

```
Foldr op r Nil          = r
Foldr op r (Cons a x) = op a (Foldr op r x)

Catenate l               = Foldr Append Nil l

ConCat a b c             = Catenate (Cons a (Cons b (Cons c Nil)))
```

It is easy to derive that Foldr is strict in its third argument only. The other two functions work on lists, and use the Foldr function. The results of the analysis are given in table 4.

We will show that in the $\mathcal{E}_2$ context the elements of the list with which Catenate is called can be evaluated safely. We have to reduce $\mathcal{E}_2$ (Catenate botmem), so:

```
Catenate botmem   → Foldr Append Nil botmem
                  → ⟨Append Top (Foldr Append Nil botmem),
                     Append Bot (Foldr Append Nil topmem)⟩
```

Because Append is strict in its first argument, the second part of the Union directly reduces to Bot:

```
Catenate botmem   ↠ Append Top (Foldr Append Nil botmem)
                  → ⟨Foldr Append Nil botmem,
                     Cons Top (Append Top (Foldr Append Nil botmem))⟩
```

The first term of this Union appeared earlier in the derivation, and it also is contained on a nonstrict position in the second part. The term can be replaced by a cyclic term, which is equivalent with a simpler one:

```
Catenate botmem ↠ x: ⟨x, Cons Top (Append Top x)⟩
                ≡ x: Cons Top (Append Top x)
```

In the $\mathcal{E}_2$ context the tail of the Cons will be evaluated further:

```
y: Append Top (Cons Top y)   → ⟨Cons Top y: Append Top (Cons Top y),
                                Cons Top (Append Top (Cons Top y))⟩
                             → ⟨z: Cons Top z, Cons Top z⟩
                             ≡  z: Cons Top z
```

The last reduction step is an application of cycle-introduction. Clearly, the resulting term will reduce to Bot in the $\mathcal{E}_2$ context.

Now the function ConCat. We will show that in the spine context the spine of the second argument can be evaluated. We have to reduce $\mathcal{E}_2$ (ConCat Top inf Top), so:

```
ConCat Top inf Top   → Catenate (Cons Top (Cons inf (Cons Top Nil)))
                     → Foldr Append Nil (Cons Top (Cons inf (Cons Top Nil)))
                     → Append Top (Foldr Append Nil (Cons inf (Cons Top Nil)))
                     → ⟨x: (Foldr Append Nil (Cons inf (Cons Top Nil))),
                        Cons Top (Append Top x)⟩
```

The first part of the Union reduces as follows:

```
Foldr Append Nil (Cons inf (Cons Top Nil))
  → Append inf (Foldr Append Nil (Cons Top Nil))
  → Cons Top (Append inf (Foldr Append Nil (Cons Top Nil)))
```

The $\mathcal{E}_2$ function forces evaluation of the tail, which results in Bot. The second part of the Union also gives Bot because of the repeated evaluation of the tail. So, the total result is Bot.

We note that the information which is derived for Catenate is not optimal. The argument of Catenate is a list of lists. However, the notation considers only reduction to head normal form of the arguments of a list. In this particular case, the spines of the elements of the argument list can be reduced, if the spine of the result is needed. This can be derived by reducing:

$\mathcal{E}_2$ (Catenate infmem)

where infmem is defined as:

```
infmem: ⟨Cons inf topmem, Cons Top infmem⟩
```

The value infmem is similar to botmem: it represents lists which contain somewhere an infinite list. So, the analysis can be extended easily.

4.8. Practical Results

The method of strictness analysis described so far has been implemented as part of the Concurrent Clean system (Plasmeijer *et al.* (1991)). The results of the analysis depend on the options that are chosen (e.g. maximum reduction depth, maximum time usage, evaluation order). The most important is the maximum reduction depth (the maximum depth of calls to Reduce, set to 10 in this case).

The functions of this section have been analysed in various ways. Table 4 gives some figures of the analysis of the functions in this section. Per function the maximum memory usage (in Kbyte) and the maximum reduction depth (reached during the analysis) are shown. The whole analysis took approximately 2 seconds of execution time (on a SUN3). With other measurements the following observations have been made:

Function	Results	Memory	Depth
G	3	10	1
F	11	10	3
Append	[1,2,3][0,2,3]	11	4
Foldr	001	10	2
Reverse	[2,2,2][0,2,3]	31	10
Catenate	[1,3,3]	14	10
Rev2	[2,2,3]	13	5
ConCat	[1,2,3][0,2,3][0,2,3]	21	9

Table 4: Results

- with eager evaluation (some) more space and time is consumed, but better results are obtained. Because of eager evaluation terms are reduced that might not be needed. These reductions can continue until the maximum depth is reached. So, some time and memory is wasted. On the other hand, evaluated terms make reduction path analysis easier. This compensates the extra use of space and time.
- without reduction path analysis more time and memory is used. The same holds if just a simple form of the less-equal comparison is done. Furthermore, the results are usually much worse. Because of the bad analysis the reductions will be deeper, so that more time and memory is claimed.
- some functions have a maximum reduction depth. For example, the reduction of function `F` will never exceed a maximum of 3, and that of `ConCat` never 9 (of course, with other settings other values are possible). On the other hand, some functions will always reach the maximum depth (e.g. `Reverse`).
- the analysis is much simpler if no list strictness is derived. For the functions in this section only 16K memory is needed, whereas the analysis takes less than half a second. We note that the Clean compiler only uses ordinary strictness information.
- the analyser is quite fast. A big Clean program, about 5000 lines, was analysed in approximately 75 seconds (without list analysis).

5. Discussion

5.1. Related Work

A lot of excellent research has been done in the area of strictness analysis, and it would go beyond the scope of this paper to give a complete overview and comparison. As a matter of fact, comparing strictness analysis techniques is difficult. Within the context of this paper, we are not only interested in the theoretical properties (e.g. completeness or safety), but also in the practical usability. Most research is of theoretical nature. Often, it is not easy to estimate what information can be derived. Even if the theoretical results are satisfactory, it is possible that in practice, because of severe implementation problems, not the same results can be obtained. It is not clear what is the most powerful method. The examples given in this paper seem to be analysed much more easily with our method.

Most research in the area of strictness analysis is based on abstract interpretation frameworks. This holds both for the basic work of Mycroft (1981) and for methods for higher order functions (e.g. Burn *et al.* (1985)), for polymorphism (e.g. Abramsky (1985)), and for lists (e.g. Wadler (1987), Wadler & Hughes (1987)). These techniques differ in two major points from our work.

Firstly, the abstract domains are finite. Their elements correspond directly to the possible kinds of nontermination that can be analysed. The abstract domains determine how the abstraction of a functional program has to be done (often, the abstraction itself already is an approximation), and what approximations during the analysis must be done. For example:

```
Head (Cons a b) = a
F x             = Head (Cons x Nil)
```

With abstract reduction, it is straightforward to show that F is strict in its argument. Within the abstract interpretation context it is not usual to include an abstract value for Cons Bot Nil. Secondly, a strictness analyser based on those ideas needs fixed point analysis to derive correct results. The advantage is that, within the current context, always the best result can be found. The drawback, however, is the exponential behaviour. For small domains this is no problem. However, it has been shown that even for rather simple cases a domain can contain many thousands of elements (Hunt (1989)). Special implementation techniques, based on the concept of frontiers, have been investigated, but turned out to be no real solution (Clack & Peyton Jones (1985), Martin & Hankin (1987), Hunt (1989)).

Another, new approach is based on a form of type inference (Leung & Mishra (1991)). Strictness information is considered to be part of a type. By using special type inference systems, various forms of strictness analysis can be done. The authors claim that their method can be implemented efficiently, mainly because no fixed point analysis is necessary. However, until now no real practical results have been presented.

5.2. Conclusion

We described a method for doing strictness analysis in Clean, a functional term graph rewrite system. It appeared that this method gives very good results under various circumstances. This has several causes.

First of all, complex abstract values (in fact: graphs) can be constructed during the analysis. This is one of the aspects of the infinite domain. In practice, this means that strictness information is stored as much as possible. Secondly, all information of the program can be used (at least in principle). In fact, the analyser can use the original program. No abstraction that loses information needs to be done. Almost all approximations are done at run-time (for example, if the maximum reduction depth is exceeded). It will be clear that better results can be obtained if more information is used. The most important problems are caused by recursive functions. However, the technique of reduction path analysis can handle this in quite an adequate way. It appeared that an implementation of reduction path analysis can be quite efficient. Furthermore, we note that higher order functions can make the analysis more complicated, but they are no basic problem. The same holds for complex forms of strictness analysis. It appeared to be possible to do strictness analysis for algebraic data types like lists just by defining the right abstract values.

From an implementation point of view, a strictness analyser based on abstract reduction has very nice properties. The method is well to implement and gives good results. As we have shown, much strictness information is found, while the run-time performance is acceptable. Furthermore, various approximations at various levels are possible. Though the analysis can be very complex, the time and memory behaviour can still be controlled. For an optimisation technique this property is very useful.

In this paper we did not present any proofs. Currently, the theoretical foundation of the method is under investigation. It appears to be possible to reformulate things in basic terms of abstract interpretation (Eekelen *et al.* (1991)). However, this turned out to be a tedious task. With this method only some basic forms of abstract reduction and reduction path analysis have been proven to be correct. A better approach seems to be the use a term rewriting framework (Nöcker (1992)). Correctness has been proven for the case of orthogonal term rewriting systems. Further extensions to general functional languages (or other types of computational models) are a topic of further research.

Acknowledgements

I would like to thank Eric Goubault, Chris Hankin and Marko van Eekelen for investing their time in the research of the theoretical foundations of abstract reduction. Thanks are also due to the referees for some useful comments.

6. References

Abramsky S., Strictness Analysis and Polymorphic Invariance, in: *Proc. of Workshop on Programs as Data Objects*, pages 1–23, DIKU, Denmark, Springer Verlag, LNCS 217, 1985.

Burn G.L., Evaluation Transformers-a Model for the Parallel Evaluation of Functional Languages (extended abstract), in: *Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA '87)*, pages 446–470, Portland, Oregon, USA, Springer Verlag, LNCS 274, 1987.

Burn G.L., C.L. Hankin and S. Abramsky, The Theory of Strictness Analysis for Higher Order Functions, in: *Proc. of Workshop on Programs as Data Objects*, pages 42–62, DIKU, Denmark, Springer Verlag, LNCS 217, 1985.

Clack C. and S. Peyton Jones, Strictness Analysis - a Practical Approach, in: *Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA '85)*, pages 35–49, Nancy, France, Springer Verlag, LNCS 201, 1985.

Eekelen M.C.J.D. van, E. Goubault, C. Hankin and E.G.J.M.H. Nöcker, Abstract Reduction: Towards a Theory via Abstract Interpretation, in: Sleep Plasmeijer and Eekelen, editors, *Proc. of Semagraph Symposium*, Nijmegen, The Netherlands, 1991.

Goubault E. and C. Hankin, A Lattice for the Abstract Interpretation of Term Graph Rewrite Systems, in: *Proc. of Semagraph Symposium*, pages 263–273, Nijmegen, The Netherlands, 1991.

Hunt S., Frontiers and Open Sets in Abstract Interpretation, in: *Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*, pages 1–13, London, UK, ACM Press,, 1989.

Leung A. and P. Mishra, Reasoning about Simple and Exhaustive Demand in Higher-Order Lazy Languages, in: *Proc. of International Conference on Functional Programming Languages and Computer Architecture (FPCA '91)*, pages 328–351, Boston, USA, Springer Verlag, LNCS 523, 1991.

Martin C. and C. Hankin, Finding Fixed Points in Finite Lattices, in: *Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA '87)*, pages 426–445, Portland, Oregon, USA, Springer Verlag, LNCS 274, 1987.

Mycroft A., *Abstract interpretation and optimising transformations for applicative programs*, PhD thesis, University of Edinburgh, 1981.

Nöcker E.G.J.M.H., Strictness Analysis by Abstract Reduction in Orthogonal Term Rewriting Systems, Technical Report 92-31, University of Nijmegen, 1992.

Plasmeijer M.J. and M.C.J.D. van Eekelen, *Functional Programming and Parallel Graph Rewriting*, Addison Wesley, 1993.

Plasmeijer M.J., M.C.J.D. van Eekelen, E.G.J.M.H. Nöcker and J.E.W. Smetsers, The Concurrent Clean System - Functional Programming on the Macintosh, in: *Proc. of 7th Int. Conference of the Apple European University Consortium*, Paris, 1991.

Toyama Yoshihito, Sjaak Smetsers, Marko Eekelen and Rinus Plasmeijer, The Functional Strategy and Transitive Term Rewriting Systems, in: *Proc. of Semagraph Symposium*, pages 99–115, Nijmegen, The Netherlands, 1991.

Wadler P. and R. J. M. Hughes, Projections for strictness analysis, in: *Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA '87)*, pages 385–407, Portland, Oregon, USA, Springer Verlag, LNCS 274, 1987.

Wadler P., Strictness Analysis over Non-flat Domains, in: *Abstract Interpretation of Declarative Languages*, Ellis Horwood,, 1987.