Efficient Strictness Analysis of Haskell

Kristian Damm Jensen Peter Hjæresen Mads Rosendahl

DIKU, University of Copenhagen,
Universitetsparken 1, DK-2100 København Ø, Denmark
email: {damm,rose}@diku.dk pen@brother.dde.dk

Abstract. Strictness analysis has been a living field of investigation since Mycroft's original work in 1980, and is getting increasingly significant with the still wider use of lazy functional programming languages.
This paper focuses on an actual implementation of a strictness analyser for Haskell. The analyser uses abstract interpretation with chaotic fixpoint iteration. The demand-driven nature of this iteration technique allows us to use large domains including function domains in the style of Burn et al. [BHA86] and Wadler [Wad87] and retain reasonable efficiency.
The implementation, furthermore, allows us to introduce a new way of handling polymorphism by parameterizing the strictness functions with domain-variables.
Finally we present some results of efficiency and precision and compare them to other works.

1 Introduction

During the past years lazy functional programming languages, i.e. languages that uses a call-by-need evaluation, have increased in popularity. Unfortunately the usual methods for implementing this evaluation-strategy are rather costly, since the compiler generates a closure for every parameter passed. If on the other hand the compiler could decide that it is safe to evaluate a parameter before it is passed to the function, it could do so, and simply pass the parameter using call-by-value. The analysis to decide when this is safe is called strictness analysis.

The earliest works in this area were published by Mycroft in 1980 and since then many authors have proposed techniques to improve the precision of the analysis. Few attempts, however, have been made to implement the technique for a large scale programming language. Two of the latest implementations have been made by Seward [Sew91] and the GRASP team [GHC92]. Both implement a strictness analyser for Haskell [Hud92], which is a large, polymorphic, higher order, lazy programming language. They both have one major drawback: they assign the 2-point domain to recursive functional arguments, which means that any analysis on a higher order function will yield very imprecise results.

This paper describes an implementation of an experimental strictness analyser that is able to do full higher order analysis of Haskell. The technique for implementing this analyser is chaotic (demand-driven) fixpoint iteration as described by Rosendahl in [Ros93a] and [Ros93b]. Chaotic fixpoint iteration also allows us to implement a new way of handling polymorphism.

The implemented system is a prototype version and has not been incorporated into a Haskell compiler. The main purpose of the system is to allow us to measure the

efficiency of the analysis. The system is able to handle full Haskell but certain parts of the language (such as cross-modular references) are given trivial interpretations.

Results of running the analyser show that it finds the best possible approximations to some complicated higher order functions in a reasonable amount of time.

2 Background

We will assume that the reader has a general knowledge of domains, abstract interpretation and strictness analysis and restrict ourselves to a short review of concepts and notations.

The aim of strictness analysis is to decide when it is safe to evaluate the i'th argument to an n-ary function f before passing it to f. This is safe when f is strict in it's i'th argument, i.e. when $fv_1 \dots v_{i-1} \perp v_{i+1} \dots v_n = \perp$ for all combinations of values of v_j.

Unfortunately strictness is an undecidable property but strictness analysis may give us a sufficient condition for strictness. In Mycroft's work [Myc80] this is done using abstract interpretation [CC77] where we compute a function $f^\sharp$ over a finite domain where strictness of $f^\sharp$ in its i'th argument implies strictness of f in its i'th argument. In its most basic form the abstract interpretation is performed on a two point domain **2** where $\perp$ denotes an undefined value and $\top$ denotes any value, using the ordering $\{\perp \sqsubset \top\}$. If

$$f^\sharp \underbrace{\top \cdots \top}_{i-1} \perp \top \cdots \top = \perp$$

then we may conclude that f is strict in its i'th argument. More sophisticated domains may be used to enhance the precision of the analysis.

Domains for lists. In 1987 Wadler introduced a four point domain for representing lists of integers. The domain has the following points:

Point	Interpretation
$\top_\in$	A fully defined list.
$\perp_\in$	Some elements are undefined.
∞	The list is either infinite or has an undefined tail. The list may be evaluated to weak-head-normal-form.
$\perp$	Totally undefined.

Since this can be considered as a double lifting of the two point domain, an obvious generalization to a list of any type is to lift the domain for this type twice. Hence a list of lists of integers has the following domain: $\mathbf{6} = \{\perp \sqsubseteq \infty \sqsubseteq \perp_\in \sqsubseteq \infty_\in \sqsubseteq \perp_{\in_\in} \sqsubseteq \top_\in\}$. In most cases it will be clear from the context in which domain a point lies, but whenever there can be doubt we will use the notation x^D meaning that the point x lies in the domain D.

We will refer to the point $\perp_\in$ as the structure point, since this is the least point in the domain where the primary structure of the list fully defined.

This analysis of lists may be generalised to recursive datatypes by double lifting the non-recursive components of the type (see [Sew91]).

Functions. Other works have extended the precision of the analysis. Among these we mention Burn, Hankin and Abramsky, who first described how to analyse higher order functions [BHA86], Abramsky, who showed that the analysis presented in [BHA86] is polymorphicly invariant [Abr85], and Baraki [Bar91] who constructed a method to approximate strictness functions for polymorphic functions from their simplest instantiation. Several other works have presented extensions for higher order functions.

Implementation. The analysis of higher order functions introduces a new problem: the domains become very large[1], and since the traditional implementation of finding the least fixpoints in a domain requires several recomputations of every point in the domain, this makes strictness analysis impractical.

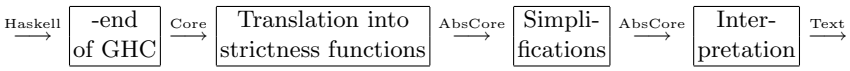
For that reason several authors have proposed better algorithms for performing the fixpoint iteration. One line of development is frontiers, first described by Clack and Peyton Jones in 1985 [CPJ85] and further developed by several others.

A different line of investigation has evolved in later years, using demand-driven or chaotic fixpoint iteration. The technique was first proposed by the Cousots in [CC78] and has recently been improved by Rosendahl to handle higher order functions [Ros93a], [Ros93a]. This technique is closely related to the semantic framework of minimal function graphs, introduced by Jones and Mycroft in [JM86]. This work has later been extended from a strict first order language, to handle higher order as well as lazy languages, see [MR92], [JR92] and [RM93].

3 An overview of the system

We have implemented a strictness analyser in Haskell, that uses three phases to perform the analysis.

The strictness analyser can be described schematicly by the following figure, where the actual analyser is described by the three last boxes.



To simplify our task, we have taken the Glasgow Haskell Compiler (version 0.16) [GHC92] as a starting point for the system, using its front-end for the parsing, type checking, simple transformations of Haskell programs etc. Input to the front-end is Haskell programs. From here we receive the program in the intermediate language, `Core`, which is essentially a typed second order lambda calculus, in which the typing has been parametriced by the use of type variables, -lambdas (Λ), and -applications. In addition to this, the front-end of GHC makes some transformations of the original program:

- All primitive applications are saturated. This means that a partial application like `(42 +)` is made into a closed lambda-expression: `(λ x. 42 + x)`. Similar transformations are made for constructor applications.

[1] [FH93] reports, that an implementation of `concat` using `foldr`, both written in continuation passing style has in the order of half a million points in the domain.

– Pattern matchings are flattened, so every pattern in every `case`-expression consists of exactly one constructor and n variables where n is the arity of the constructor.
– Trivial defaults are removed.

Besides this we perform a lambda and let-lifting of the program to simplify the evaluation of the strictness functions.

The abstract syntax of `Core` is shown in figure 1.

program	$::=$	$def_1 \ldots def_n$	$(n \geq 1)$
def	$::=$	$var\ = [\Lambda tvar.]^*$ $[\lambda var_1 \ldots var_n.]\ expr$	Definition $(n \geq 1)$
expr	$::=$	var	Variable
	$\mid$	c	Constant
	$\mid$	$C\ texpr_1 \ldots texpr_d\ expr_1 \ldots expr_n$	Constructor application
	$\mid$	$expr\ expr$	Application
	$\mid$	$expr\ @\ texpr$	Type-application
	$\mid$	`let` $var = expr$ `in` $expr$	Local definitions
	$\mid$	$primop\ expr_1 \ldots expr_n$	Primitive operations
	$\mid$	`case` $expr$ `of` $alt_1 \ldots alt_n\ [default]$	Case expression $(n \geq 1)$
alt	$::=$	$C\ var_1 \ldots var_n \rightarrow expr$	
default	$::=$	$var \rightarrow expr$	Bind-default
	$\mid$	$\_ \rightarrow expr$	Wild-default
texpr	$::=$	$tvar$	Type-variable
	$\mid$	tc	Type-constant
	$\mid$	$texpr\ @\ texpr$	Type-application

Fig. 1. Abstract syntax for `Core`

Analysis. Instead of interpreting `Core` using a strictness semantics, we have chosen to translate the program into strictness functions. The functions are represented in a abstract language called `AbsCore`. The translation is done using the schemes described by Mycroft [Myc80] and Burn et al. [BHA86] with some extra machinery needed to handle algebraic datatypes and polymorphism. We return to this in section 4.2. An advantage with using an intermediate language for strictness functions is that it allows performing optimising transformations on the strictness functions. At the moment we only perform some simple algebraic transformations.

Finally (box four) we have the interpretation, that derives strictness information from the program using chaotic fixpoint iteration described in section 5. This information is then displayed as text.

4 Translation

In this section we describe how we translate `Core`-programs into strictness functions. Strictness functions are represented in a special language called `AbsCore`.

4.1 Domains and polymorphism

The strictness analysis we have implemented is based on the techniques of BHA and Wadler with function domains for higher-order functions and double-lifting for recursive datatypes.

There are at least two techniques for analysing strictness in the presence of polymorphism.

1. We may analyse functions only for the simplest instance of the type variable and construct approximations (safe but less precise) for other instantiations using the technique of Baraki [Bar91]. We will refer to this as a *monomorphic analysis* [Sew91], [Sew93].
2. We can instantiate the polymorphism during the compilation pass to obtain (monomorphic) versions of each function for all the instantiations of type variables that may occur in the program. We will refer to this as a *polymorphic analysis*.

The second option is naturally more precise but less efficient than the first. In the context of demand-driven evaluation we may, however, use the second option without actually producing all monomorphic instances. We return to this in section 4.2.

4.2 From `Core` to `AbsCore`

For the most part the translation into strictness functions is fairly straightforward and most of the techniques have been described by Mycroft [Myc80] and Burn et al [BHA86]. They describe how to translate variables, constants, primitive operations, lambda-abstractions and function applications into abstract variables, domain points, $\sqcap$, abstract lambdas and application of abstract functions.

What remains to be done is to show how to translate constructors, pattern matching in case-expressions, and polymorphism.

Constructors. A constructor C in the original program is translated into an abstract constructor $C^\sharp$ which is an abstract function that may be derived automatically from the datatypes. This may be seen as a direct generalization of Wadlers analysis of lists [Wad87] and also discussed in [Sew91] and others.

Non-recursive constructors return the top point of its domain and other constructors return a value that depends on the recursive arguments. If any of the recursive arguments is less than the structure point $\bot_\in$ the constructor returns ∞. Otherwise we find the least upper bound of the recursive arguments and the double-lifted non-recursive arguments.

Example 1. Considering the datatype `List Int` we get the following abstract constructors:

```
Cons♯ : 2 →4 →4                                  Nil♯ : 4
Cons♯ x xs = if xs = ⊥ or xs = ∞ then ∞          Nil♯ = ⊤∈
             else x∈ ⊔ xs
```

case-expressions. The de-construction of values in case-expressions is done using inverse constructors (see also [NN92]). Each alternative in the case expression should be evaluated with bindings that, when passed as arguments to the constructor, gives a value less than or equal to the tested value. An inverted constructor $C^{-\sharp}$ must thus for all values p satisfy:

$$LC(C^{-\sharp}\ p) \supseteq LC(\{\langle v_1, \ldots, v_n\rangle \mid C^{\sharp}\ v_1 \ldots v_n = p\})$$

where LC is the lower closure

It is of course always possible to define such functions since the set $\{\top\}$ always satisfies the condition, but the precision of the analysis will be improved if the function returns values as low in the domain as possible.

Example 2. The inverse constructors for list of integers are as follows

$$\begin{array}{ll} \texttt{Cons}^{-\sharp}\ \top_{\in} = \{\langle \top, \top_{\in}\rangle\} & \texttt{Nil}^{-\sharp}\ \top_{\in} = \{\langle\rangle\} \\ \texttt{Cons}^{-\sharp}\ \bot_{\in} = \{\langle \bot, \top_{\in}\rangle, \langle \top, \bot_{\in}\rangle\} & \texttt{Nil}^{-\sharp}\ \_ = \{\} \\ \texttt{Cons}^{-\sharp}\ \infty = \{\langle \top, \infty\rangle\} & \\ \texttt{Cons}^{-\sharp}\ \bot = \{\} & \end{array}$$

Note the difference between the set of an empty tuple and the empty set.

Given this inversion of the constructors we are able to generate the alternatives of the abstract `case`-expression:

Let v be the value of the expression to be tested. For each constructor pattern $C_j\ x_1 \ldots x_{k_j}$ in the case expression we now compute $C_j^{-\sharp}\ v$ to get the following set of argument tuples:

$$J_j = \{\langle p_1^1 \ldots p_{k_j}^1\rangle, \ldots, \langle p_1^z \ldots p_{k_j}^z\rangle\}$$

For every argument tuple $\langle p_1 \ldots p_{k_j}\rangle \in J_j$ we now evaluate the abstraction of the corresponding right hand side e_j with the free variables $x_1 \ldots x_{k_j}$ bound to $p_1 \ldots p_{k_j}$ (see also figure 3)

This gives us a set of expressions for every alternative in the `case`-expression. The interpretation of the `case`-expression is then the least upper bound of evaluating all right hand sides for all these values of the pattern variables.

Polymorphism. When the GHC -end translates Haskell programs to `Core` it makes the polymorphism explicitly parametriced. Let f_α be a polymorphic function with a type variable α. The analyser may then have to analyse the function for all instantiations of α that occur in the program. We use an equivalent, but simpler, approach where we construct one strictness function for each function. This function, however, is parameterised in (a name for) the abstract domain and the function may then be called with different values of this parameter. Thus the strictness function $f_\delta^\sharp$ has a domain argument δ corresponding to the type argument α of the initial function f_α.

The two main advantages of this approach are that strictness functions normally do not depend heavily on these "domain variables", and secondly that different type-instantiations of the original function may give the same "domain variables" to the strictness function (e.g. f_{int} and f_{bool} may both be analysed using $f^{\sharp}_{\mathbf{2}}$). The parameter is only used when constructing and deconstructing values of polymorphic datatypes.

Constructors. For polymorphic datatypes we just make parameterised abstract constructors so that the "domain variables" are passed on to the abstract constructors. The domain variables are needed to determine the correct domains for the values and thus the correct computation of least upper bound.

Example 3. Extending example 1 to the datatype `List a` we need to parameterize the abstract constructors:

$$\texttt{Cons}^{\sharp}_{\delta} : \delta \to \delta_{\perp_\perp} \to \delta_{\perp_\perp} \qquad \texttt{Nil}^{\sharp}_{\delta} : \delta_{\perp_\perp}$$

```
Cons#δ x xs = if xs = ⊥ or xs = ∞ then ∞      Nil#δ = ⊤∈
              else x∈ ⊔ xs
```

In this example the constructors are parameterized using the domain variable as an index. In the evaluator these domain parameters are treated as ordinary parameters.

Normally values of domain variables may be computed from the context but since the `Core` language does allow a kind of let-polymorphism, it is not always possible. This may occur for polymorphic expressions where the result does not depend on the type parameter. In those situations we just analyse the expression with the value **2** for the uninstantiated domain variables. Since strictness is a denotational property we cannot expect strictness analysis to distinguish between instantiations that cannot be distinguished denotationally.

As an example consider the expression `length Nil` which will return the same result for empty lists of any type.

`case`-*expressions.* Since the abstract constructors are now parameterized with domain variables, $C^{-\sharp}$ will also need to know the domain variables. Furthermore we may need them to examine whether a given value is the top point in its domain, which is important for the correct handling of zero-ary constructors.

4.3 Summary

Given this description we are now able to present the abstract syntax of `AbsCore` (figure 2) as well a schema for translation from `Core` to `AbsCore` (figure 3). In both figures `AbsCore` keywords are underlined to make it easier to distinguish them from `Core` keywords.

$$
\begin{array}{lll}
\mathit{program} & ::= \mathit{def}_1 \ldots \mathit{def}_n & \\
\\
\mathit{def} & ::= \mathit{var} = [\underset{\cdot\cdot}{\Lambda}\ \mathit{domvar}\ .]^* & \\
 & \quad\ [\underset{\cdot\cdot}{\lambda}\ \mathit{var}_1 \ldots \mathit{var}_n\ .]\ \mathit{expr} & \text{Definition} \\
\\
\mathit{expr} & ::= \mathit{var} & \text{Variable} \\
 & \mid\ \mathit{domvar} & \text{Domain variable} \\
 & \mid\ p & \text{Constant point} \\
 & \mid\ C^\sharp\ \mathit{domvar}_1 \ldots \mathit{domvar}_j\ \mathit{expr}_1 \ldots \mathit{expr}_n & \text{Abstract constructor} \\
 & \mid\ \mathit{expr}\ \mathit{expr} & \text{Application} \\
 & \mid\ \mathit{expr} \sqcap \mathit{expr} & \text{Greatest lower bound} \\
 & \mid\ \mathit{expr} \sqcup \mathit{expr} & \text{Least upper bound} \\
 & \mid\ \underset{\cdot\cdot}{\texttt{let}}\ \mathit{var} = \mathit{expr}\ \underset{\cdot\cdot}{\texttt{in}}\ \mathit{expr} & \text{Local definitions} \\
 & \mid\ \underset{\cdot\cdot}{\texttt{if}}\ \mathit{expr}\ \underset{\cdot\cdot}{\texttt{then}}\ \mathit{alt}_1\ \underset{\cdot\cdot}{\texttt{else}}\ \mathit{alt}_2 & \text{conditional expressions}
\end{array}
$$

Fig. 2. Abstract syntax for `AbsCore`.

$$
\begin{array}{ll}
\mathcal{C}_{\text{program}}\text{: CoreProgram} \to \text{AbsProgram} & \\
\mathcal{C}_{\text{program}}[\![b_1 \ldots b_n]\!] & = \mathcal{C}_{\text{bind}}[\![b_1]\!] \ldots \mathcal{C}_{\text{bind}}[\![b_n]\!] \\
\\
\mathcal{C}_{\text{bind}}\text{: PlainCoreBinding} \to \text{AbsBinding} & \\
\mathcal{C}_{\text{bind}}[\![v = e]\!] & = v = \mathcal{C}[\![e]\!] \\
\\
\mathcal{C}\text{: PlainCoreExpr} \to \text{AbsExpr} & \\
\mathcal{C}[\![v]\!] & = v \\
\mathcal{C}[\![c]\!] & = \mathcal{C}_{\text{lit}}[\![c]\!] \\
\mathcal{C}[\![C\ \mathit{tvar}_1 \ldots \mathit{tvar}_d\ e_1 \ldots e_n]\!] & = C^\sharp\ \mathit{domvar}_1 \ldots \mathit{domvar}_d\ \mathcal{C}[\![e_1]\!] \ldots \mathcal{C}[\![e_n]\!] \\
\mathcal{C}[\![\mathit{primop}\ e_1 \ldots e_n]\!] & = \sqcap_{i=1}^{n}\ \mathcal{C}[\![e_i]\!] \\
\mathcal{C}[\![\lambda v_1 \ldots v_n . e]\!] & = \underset{\cdot\cdot}{\lambda} v_1 \ldots v_n\ .\ \mathcal{C}[\![e]\!] \\
\mathcal{C}[\![\Lambda \mathit{tvar}.e]\!] & = \underset{\cdot\cdot}{\Lambda} \mathit{domvar}\ .\ \mathcal{C}[\![e]\!] \\
\mathcal{C}[\![e_1\ e_2]\!] & = \mathcal{C}[\![e_1]\!]\ \mathcal{C}[\![e_2]\!] \\
\mathcal{C}[\![e\ @\ \mathit{te}]\!] & = \mathcal{C}[\![e]\!]\ \mathcal{D}[\![\mathit{te}]\!] \\
\mathcal{C}[\![\texttt{let}\ b\ \texttt{in}\ e]\!] & = \underset{\cdot\cdot}{\texttt{let}}\ \mathcal{C}_{\text{bind}}[\![b]\!]\ \underset{\cdot\cdot}{\texttt{in}}\ \mathcal{C}[\![e]\!] \\
\mathcal{C}[\![\texttt{case}\ e\ \texttt{of}\ \mathit{alt}_1\ \ldots\ \mathit{alt}_n]\!] & = \underset{\cdot\cdot}{\texttt{if}}\ \mathcal{C}[\![e]\!] = \bot\ \underset{\cdot\cdot}{\texttt{then}}\ \bot \\
 & \qquad \underset{\cdot\cdot}{\texttt{else}} \bigsqcup_{c=1}^{n} \bigsqcup_{\vec{p} \in J_c} \underset{\cdot\cdot}{\texttt{let}}\ \vec{x} = \vec{p}\ \underset{\cdot\cdot}{\texttt{in}}\ \mathcal{C}[\![e_c]\!] \\
\text{where}\ C_j\ x_{j,1} \ldots x_{j,m_j} \to e_j & = \mathit{alt}_j \\
J_j & = C_j^{-\sharp}\ \mathit{domvar}_1 \ldots \mathit{domvar}_{d_j}\ \mathcal{C}[\![e]\!] \\
\\
\mathcal{C}_{\text{lit}}\text{: CoreLiteral} \to \text{Point} & \\
\mathcal{C}_{\text{lit}}[\![c]\!] & = \top^{\mathbf{4}}\text{, if } c \text{ is an array or a string} \\
\mathcal{C}_{\text{lit}}[\![c]\!] & = \top^{\mathbf{2}}\text{, otherwise}
\end{array}
$$

Both in the description of $C^\sharp$ and $C^{-\sharp}$ $\mathit{domvar}_1 \ldots \mathit{domvar}_d$ denoted the domain variables corresponding to the parameters of the constructors datatype.
$\mathcal{D}[\![\mathit{te}]\!]$ is the domain-expression that corresponds to the type-expression te.

Fig. 3. Translation from `Core` to `AbsCore`

5 Iteration

In the simplest implementations of strictness analysis the strictness functions are tabulated and the evaluation is iterated until stability. This means that the strictness function is reevaluated at every point of the domain in every step. On the other hand this conflicts with our wish to have larger domains since larger domains give more precise information. Especially function domains tend to be *very* large.

It turns out however, that it is quite unnecessary to evaluate every point of the domain since we are only interested in evaluating the strictness functions with arguments of the form $\top \cdots \top \bot \top \cdots \top$. To compute the value at these points we may need to know the values in certain other points etc. This process of iteration is known as demand-driven or chaotic fixpoint iteration.

The generalisation to higher-order functions raises two new problems: how to detect what is needed of a function and how to compare functions for equality without evaluating for all arguments.

5.1 Higher-order needs

The general principle of recording needs of higher-order functions is to record the needs as soon as a function is called with arguments or used as an argument to a function. Simple calls, as say $f(42)$, are easy to detect. If, however, f is passed as argument to a function $g(h) = h(42)$ then we will record the call $f(42)$ already when g is called with f since we know that g will call its argument with 42. This principle of recording needs may be illustrated in the following example.

Example. Consider the program

$$\begin{aligned} f\ n\ k\ &= n \wedge ((k\ \top) \vee (f\ n\ (m\ n\ k))) \\ m\ n\ k\ x &= k(n \wedge x) \end{aligned}$$

This is the strictness function for the CPS converted factorial function.

$$\begin{aligned} f\ n\ k\ &= \textbf{if}\ n = 0\ \textbf{then}\ k\ 1\ \textbf{else}\ f\ (n-1)\ (m\ n\ k) \\ m\ n\ k\ x\ &= k(n * x) \\ id\ x\ &= x \\ fac\ x\ &= f\ x\ id \end{aligned}$$

As part of a strictness analysis we may need to evaluate the call $f\ \top\ (\lambda x.\bot)$. A demand-driven iteration may proceed as follows.

First iteration.

$$f\ n\ k\ \ [n = \top, k = \lambda x.\bot] = \bot$$

k is called with $\top$
f is called with $\langle \top, \lambda\, x.\bot \rangle$

During the evaluation we detect calls to k and f. We do not detect a call to m since we do not know whether f will call its second argument with any arguments. The value of $(m\ n\ k)$ is the bottom function since the function has not been called with any arguments yet.

Second iteration.

$f\ n\ k\ \ [n = \top, k = \lambda x.\bot] = \bot$	k is called with $\top$
	f is called with $\langle \top, \lambda x.\bot \rangle$
	m is called with $\langle \top, \lambda x.\bot, \top \rangle$
$m\ n\ k\ x\ \ [n = \top, k = \lambda x.\bot, x = \top] = \bot$	k is called with $\top$

As in the first iteration there are calls to f and k. In the call to f the arguments are $\top$ and $\lambda x.\bot$ and in the first iteration we saw that f with those arguments will call its second argument with $\top$. This means that the function $(m\ n\ k)$ will be called with the extra argument $\top$.

Third iteration.

$f\ n\ k\ \ [n = \top, k = \lambda x.\bot] = \bot$	k is called with $\top$
	f is called with $\langle \top, \lambda x.\bot \rangle$
	m is called with $\langle \top, \lambda x.\bot, \top \rangle$

When m is called with the arguments $\langle \top, \lambda x.\bot, \top \rangle$ the second argument will be called with $\top$. This means that there will be an extra call to k with $\top$ inside the call to m in function f. Since we have already detected such a call and there are no other changes of results or needs the iteration has stabilised.

5.2 Collection of needs and results

The main problem of collecting needs for higher order functions is to propagate information about needs in applications backwards as needs of the arguments. As illustrated in the example we not only need to record argument/result pairs for functions but also argument/needs-of-argument pairs. The iteration assumes initial approximations of these description (stored together as a function environment $\phi \in \Phi$) and it computes better descriptions until stability (see [Ros93b]). When testing for stability we only have to compare the needed parts of function graphs.

The body of functions are assumed to be of first-order (saturated). During the evaluation the values of parameters is stored in a parameter environment $\rho \in \mathcal{R}$. Sub-expressions may, however, be of higher-order and they will be evaluated for all the extra arguments which are needed. In the evaluation such extra arguments to unsaturated expression will be represented as a list of values $\sigma \in \Sigma$.

Summing up, we have the following arguments to the evaluation function.

$\phi \in \Phi \stackrel{\text{def}}{=} \mathcal{V} \rightarrow \mathcal{F}$	Function environment
$\rho \in \mathcal{R} \stackrel{\text{def}}{=} \mathcal{V} \rightarrow \mathcal{P}$	Parameter environment
$\sigma \in \Sigma \stackrel{\text{def}}{=} \mathcal{P}^*$	Unsaturated expressions

where $\mathcal{V}$ is the set of variable names, $\mathcal{F} = \mathcal{P}^* \rightarrow D \times \mathcal{N}$ are function graphs, and values $\mathcal{P}$ are either simple values $\mathcal{D}$ or function graphs $\mathcal{F}$. Needs $\mathcal{N}$ are explained below.

The evaluation function will evaluate an expression and collect needed arguments during the evaluation. We record three kinds of needs during the evaluation: needs of functions (defined in ϕ), needs of parameters (defined in ρ), and needs

of extra arguments (defined in σ). Needs of functions are represented as function-name/argument-tuple pairs $\mathcal{C} = 2^{(\mathcal{V},\Sigma)}$. Needs of parameters and extra arguments are represented as index/argument-tuple pairs $\mathcal{N} = 2^{(Int,\Sigma)}$ with information that the i'th parameter (in ρ) or the i'th extra argument (in σ) is called with certain arguments. Since the numbering of extra arguments and parameters will change inside lambda abstractions and applications, the recorded needs must be renumbered. This is done using an auxiliary function $\mathbf{deNum}_i : \mathcal{N} \to \mathcal{N}$ which adds i (possibly negative) to parameter positions.

The evaluation function for the fixpoint iteration is given in figure 4.

$$
\begin{array}{rl}
\mathbf{E}[\![\cdot]\!] : \Phi \to \mathcal{R} \to \Sigma \ \to (\mathcal{P} \times \mathcal{C} \times \mathcal{N} \times \mathcal{N}) & \\
\mathbf{E}[\![v]\!]\phi\rho\sigma & = (p, \{(v,\sigma)\}, \emptyset, n) \\
\quad \text{Where } (p,n) & = \phi(v)(\sigma) \\
\mathbf{E}[\![x]\!]\phi\rho\sigma & = \mathbf{if}\ isbase(\rho(x))\ \mathbf{then}\ \ (\rho(x), \emptyset, \{(\mathbf{pos}\ (x,\rho),\sigma)\}, \emptyset) \\
 & \quad\ \mathbf{else}\ \ (p', \emptyset, \{(\mathbf{pos}\ (x,\rho),\sigma)\}, n) \\
\quad \text{Where } (p',n) & = \rho(x)(\sigma) \\
\mathbf{E}[\![c]\!]\phi\rho\sigma & = \mathbf{if}\ isbase(c)\ \mathbf{then}\ \ (c, \emptyset, \emptyset, \emptyset)\ \mathbf{else}\ \ (p, \emptyset, \emptyset, n) \\
\quad \text{Where } (p,n) & = c(\sigma) \\
\mathbf{E}[\![e_1\ e_2]\!]\phi\rho\sigma & = (v_1, c_1 \cup cs, n_1 \cup ns, \mathbf{deNum}_1\ r_1) \\
\quad \text{Where } (v_2,c_2,n_2,r_2) & = \mathbf{E}[\![e_2]\!]\phi\rho\langle\rangle \\
(v_1,c_1,n_1,r_1) & = \mathbf{E}[\![e_1]\!]\phi\rho\langle v_2 : \sigma\rangle \\
apps & = \{\mathbf{E}[\![e_2]\!]\phi\rho a \mid (1,a) \in r_1\} \\
cs & = \bigcup\{c \mid (v,c,n,r) \in apps\} \\
ns & = \bigcup\{n \mid (v,c,n,r) \in apps\} \\
\mathbf{E}[\![\underline{\lambda}\ x_1,\ldots,x_k.e]\!]\phi\rho\sigma & = (v, c, \mathbf{deNum}_k\ n, r' \cup \mathbf{deNum}_{(-k)}\ r) \\
\quad \text{Where } (v,c,n,r) & = \mathbf{E}[\![e]\!]\phi\rho[x_1 \mapsto s_1, \ldots, x_k \mapsto s_k]\sigma' \\
\langle s_1 : \ldots : s_k : \sigma'\rangle & = \sigma \\
r' & = \{(i,a) \mid (i,a) \in n \wedge i \le k\} \\
\mathbf{E}[\![\underline{C^\sharp}\ d_1 \ldots d_j\ e_1 \ldots e_s]\!]\phi\rho\sigma & = (v, c_1 \cup \cdots \cup c_s, n_1 \cup \cdots \cup n_s, \emptyset) \\
\quad \text{Where } v_i & = \mathbf{E}[\![e_i]\!]\phi\rho\langle\rangle,\ i = 1 \ldots s \\
v & = C^\sharp\ d_1 \ldots d_j\ v_1 \ldots v_s \\
\mathbf{E}[\![\underline{\text{let}}\ v \underline{=} e_1\ \underline{\text{in}}\ e_2]\!]\phi\rho\sigma & = (v_2, c_1 \cup c_2, n_1 \cup \mathbf{deNum}_1\ n_2, r_2) \\
\quad \text{Where } (v_1,c_1,n_1,r_1) & = \mathbf{E}[\![e_1]\!]\phi\rho\langle\rangle \\
(v_2,c_2,n_2,r_2) & = \mathbf{E}[\![e_2]\!]\phi\rho[v \mapsto v_1]\sigma \\
\mathbf{E}[\![\underline{\text{if}}\ e_1 \underline{\text{then}}\ e_2 \underline{\text{else}}\ e_3]\!]\phi\rho\sigma & = (v_2, c_1 \cup c_2, n_1 \cup n_2, r_2) \quad \mathbf{if}\ v_1 = true \\
 & = (v_3, c_1 \cup c_3, n_1 \cup n_3, r_3) \quad \mathbf{if}\ v_1 = false \\
 & = (\bot, c_1, n_1, \emptyset) \quad \mathbf{if}\ v_1 = \bot \\
\quad \text{Where } (v_1,c_1,n_1,r_1) & = \mathbf{E}[\![e_1]\!]\phi\rho\langle\rangle \\
(v_i,c_i,n_i,r_i) & = \mathbf{E}[\![e_i]\!]\phi\rho\sigma \qquad i = 2,3 \\
\mathbf{E}[\![e_1 \sqcap e_2]\!]\phi\rho\sigma & = (v_1 \sqcap v_2, c_1 \cup c_2, n_1 \cup n_2, \emptyset) \\
\quad \text{Where } (v_i,c_i,n_i,r_i) & = \mathbf{E}[\![e_i]\!]\phi\rho\langle\rangle \qquad i = 1,2 \\
\mathbf{E}[\![e_1 \sqcup e_2]\!]\phi\rho\sigma & = (v_1 \sqcup v_2, c_1 \cup c_2, n_1 \cup n_2, \emptyset) \\
\quad \text{Where } (v_i,c_i,n_i,r_i) & = \mathbf{E}[\![e_i]\!]\phi\rho\langle\rangle \qquad i = 1,2
\end{array}
$$

Fig. 4. Evaluation function for chaotic fixpoint iteration.

6 Results

Most published works on strictness analysis have been centered around the choice of domains and the choice of the method of fixpoint iteration. Very little has been published about precision and efficiency using the various domains and techniques.

We will try to amend this by giving a short review of published results of efficiency and compare these results to the results from the described implementation.

Lastly we present a single result regarding precision which turned up during the work with this implementation.

6.1 Implementations

Considering that the field of strictness analysis is about 15 years old, surprisingly few results about the efficiency of strictness analysers have been published. Of course strictness analysers have been implemented in the past but the focus has been on the precision rather than the efficiency of the analysis.

To our knowledge only three results has been published: Seward's analysis of Haskell [Sew91], Ferguson and Hughes' implementation using sequential algorithms [FH93] and mentioned in that article are the results of Hunt and Hankin, of which we have no other account. These implementations are compared in this table:

	Year	Machine	Language	Technique	HO?
Seward	91	Sun 3	Miranda	Frontiers	No
Hunt & Hankin	91	?	?	Frontiers	Yes
Ferguson & Hughes	93	Sun 4/75	Lazy ML	Seq. algorithms	Yes
Hjæresen & Jensen	94	Sparc 2	Haskell[2]	Chaotic fixpoint	Yes

6.2 Efficiency

There seems to be some agreement that the function `concat` implemented by `foldr append []` is a suitable benchmark and — if the analysis is good enough — to rewrite both functions into continuation passing style and analyse that. We will make no attempt to enlarge this benchmark suite.

Implementors	foldr	concat	cps-foldr	cps-concat
Hunt & Hankin		15 min		> 15 hours
Ferguson & Hughes		5 sec		10 sec
Hjæresen & Jensen	0.07 sec	0.05 sec	3.5 sec	2.1 sec

Space complexity:

Implementor	foldr	concat	cps-foldr	cps-concat
Hjæresen & Jensen	4 k	2 k	37 k	28 k

One will note that Seward's implementation is not mentioned in these tables. This is because it handles higher order functions by the 2-point domain and thus gives very imprecise results for any higher order function.

[2] Time and space complexity is measured in a C implementation of the algorithm.

6.3 Precision

One notable detail that turned up during this work is that a monomorphic analyser that utilizes Baraki's result for approximating strictness functions of polymorphic functions may not give as precise an analysis as a similar polymorphic analysis. This is not trivial to see however, since polymorphism can only be of interest if combined with a higher order function:

The difference between two different instances of a type-variable α only shows, if the "contents" of α is analysed by some function f. A monomorphic function f can not make this inspection without the type checker being able to deduce a more precise instance of α. A higher order function on the other hand can make this inspection using a functional argument, and hiding this fact from the type-checker.

Example 4. Consider the following small Haskell program:

```
poly :: [[*]] -> Int
poly list = sum (map length list)

sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x+sum xs

map :: ([Int] -> Int) -> [[Int]] -> [Int]
map f [] = []
map f (xs:xss) = f xs:map f xss
```

The interesting point here is $\infty_{\in}$. In a polymorphic analysis, if $\texttt{poly}^\sharp$ is applied to $\infty_{\in}$ we find the result $\bot$. If on the other hand we made a monomorphic analysis we would have $\texttt{map}^\sharp :: (\mathbf{2} \to \mathbf{2}) \to \mathbf{4} \to \mathbf{4}$, and $\texttt{length}^\sharp$ and $\texttt{list}^\sharp$ would therefore get the domains $\mathbf{2} \to \mathbf{2}$ and $\mathbf{4}$ respectively. Furthermore the only safe approximation of $\infty_{\in} \in \mathbf{6}$ in the 4-point domain is $\top_{\in}^{\mathbf{4}}$.

Given this we get the following analysis:

expression	Is abstracted to	
	monomorphic	polymorphic
`list`	$\top_{\in}^{\mathbf{4}}$	$\infty_{\in}^{\mathbf{6}}$
`map length list`	$\top_{\in}^{\mathbf{4}}$	$\bot_{\in}^{\mathbf{4}}$
`poly list`	$\top^{\mathbf{2}}$	$\bot^{\mathbf{2}}$

That is, the monomorphic analysis does not detect that $\texttt{(poly [[1], [1..], [5,6]])}^\sharp = \bot$.

7 Extensions and Future Work

7.1 Extension already made

Besides implementing Wadler's analysis of list we have also made an implementation with a more ambitious analysis of recursive datatypes. We have there generalized the structure part of the 4-point domain by considering structural dimensions: a list has

one structural dimension, a binary tree has two, etc. This allows us to distinguish between strictness in various directions.

The domain construction can be further improved with a more precise handling of functional arguments to constructors and domain variables that are instantiated with recursive types.

For an accurate description of this construction see [HJ94].

7.2 Future work

Utilizing the obtained strictness information. The strictness analyser should be connected to the GHC back-end so the information can be used. At the same time it would be natural to improve the efficiency of the code and include some more simplifications in this module.

Compare domains for efficiency and precision. A lot of work has been done constructing various domains for abstract interpretation. Very little work has been done comparing them for usability. Since it appears that fixpoint iteration can be done efficiently even for given large domains it would be interesting to follow this branch.

Simplification of strictness functions. Our current version only implement some very simple transformations prior to the evaluation. It may in many situations be possible to transform recursively defined strictness functions into a non-recursive form and thus simplifying the evaluation. Such transformations are likely to improve the efficiency of the implementation considerably since non-recursive functions do not require iteration. In our work we have taken an extensional view of strictness functions and our approach may then be seen as orthogonal to other more intensional approaches (eg. [HLM94]).

8 Conclusion

We have described the implementation of a strictness analyser for full Haskell using abstract interpretation. The fixpoint iteration of the analyser is implemented using chaotic fixpoint iteration.

The analyser introduces a new way of handling of polymorphism: strictness functions are parameterised by domain variables in much the same way that polymorphic functions can be parametriced with type variables. The domains used for the interpretation are not described in detail, but include the domains of Burn, Hankin and Abramsky [BHA86] and Wadler [Wad87] as a special case.

Since the fixpoint iteration is demand-driven iteration, the full number of points in the domains of analysis do not directly affect the efficiency of the analysis.

A full account of the implemented analyser, including the more detailed domains used for structured datatypes, can be found in [HJ94]. The theory behind the system is described in [Ros93a], [Ros93b] and [HJ94].

Acknowledgements. We wish to thank Carsten Kehler Holst, Morten Welinder and the anonymous referees for many helpful and detailed comments.

References

[Abr85] S Abramsky, *Strictness Analysis And Polymorphic Invariance*. LNCS 217, H Ganzinger & N D Jones (ed.): "Programs as Data Objects", Springer Verlag 1986.

[AH87] S Abramsky & C Hankin, *Abstract interpretation of declarative languages*. Ellis-Horwood 1987.

[Bar91] G Baraki, *A Note on Abstract Interpretation of Polymorphic Functions*. LNCS 523, J Hughes (ed): FPCA'91, Springer Verlag 1991.

[BHA86] G L Burn, C Hankin & S Abramsky, *Strictness Analysis for Higher Order Functions*. Science of Computer Programming vol. 7, pp. 249–278.

[CPJ85] C Clack & S L Peyton Jones, *Strictness Analysis—A Practical Approach*. LNCS 201, FPCA'85, Springer Verlag 1985.

[CC77] P Cousot & R Cousot, *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximations of Fixpoints*. 4th POPL, Los Angeles, CA.

[CC78] P Cousot & R Cousot, *Static Determination of Dynamic Properties of Recursive Procedures*. In E J Neuhold: "Formal Description of Programming Concepts". North-Holland 1978.

[FH93] A Ferguson & J Hughes, *Fast Abstract Interpretation Using Sequential Algorithms*. LNCS 724, P Cousot et al.: WSA'93, Springer Verlag 1993.

[GHC92] The GRASP Team, *The Glorious Haskell Compilation System. User's Guide*. Version 0.16 1992.

[HLM94] C Hankin & D Le Metayer , *Lazy Type Inference for the Strictness Analysis of Lists*. ESOP'94, LNCS 788, Springer-Verlag.

[HJ94] P F Hjæresen & K D Jensen, *Strikthedsanalyse af Haskell*, Master's Thesis. DIKU, University of Copenhagen, Denmark, 1994.

[Hud92] P Hudak et al., *Report on the Programming Language Haskell, version 1.2*. SIGPLAN Notices 27.

[HH91] S Hunt & C Hankin, *Fixed Points and Frontiers: A New Perspective*. Journal of Functional Programming vol. 1, no. 1.

[JM86] N D Jones & A Mycroft, *Dataflow Analysis of Applicative Programs Using Minimal Function Graphs*. 13th POPL, St. Petersburg, Florida. 1986.

[JR92] N D Jones & M Rosendahl, *Higher Order Functional Graphs*. ALP'94, LNCS, Springer-Verlag 1994.

[Myc80] A Mycroft, *The theory and practise of transforming call-by-need into call-by-value*. International symposium on programming '80, Paris, France.

[MR92] A Mycroft & M Rosendahl, *Minimal function graphs are not instrumented*. WSA'92, Bordeaux, France.

[NN92] F Nielson & H R Nielson, *The Tensor Product in Wadlers Analysis of Lists*. ESOP'92

[Ros93a] M Rosendahl, *Higher Order Chaotic Iteration Sequences*. LNCS 714, PLILP'93, Springer Verlag 1993.

[Ros93b] M Rosendahl, *Higher order fixpoint iteration*. Unpublished 1993.

[RM93] M Rosendahl & A Mycroft, *Minimal Function Graph Semantics for a Lazy Language*, Unpublished 1993.

[Sew91] J Seward, *Towards a Strictness Analyser for Haskell: Putting Theory into Practice*. 1991, Master's Thesis, University of Manchester.

[Sew93] J Seward, *Polymorphic Strictness Analysis Using Frontiers*, PEPM'93, Copenhagen, Denmark.

[Wad87] P Wadler *Strictness analysis on non-flat domains (by Abstract interpretation over finite domains)*. In [AH87].