Technical Report 1/95

The G#-machine: Efficient strictness analysis in Haskell

Marko Schütz

Johann Wolfgang Goethe-Universität
Frankfurt am Main

Fachbereich Informatik (20)

Contents

Chapter 1

Introduction

Strictness analysis has been an active field of research as strictness information marks an indispensable element of an efficient compilation of lazy functional languages like Haskell. Most work applies some means of abstract interpretation for strictness analysis. It has been shown that the required fixpoint iteration is generally too inefficient for practical implementation.
Another approach are type based algorithms. These seem to be efficient enough for practical use, but as they again are restricted to finite domains they fail in some obvious cases to derive strictness information.
We present an efficient implementation of abstract reduction as proposed by Nöcker [Nöc93] for a core language of Haskell. This implementation is based on an abstract machine, the $G^{\#}$-machine, which realises abstract reduction.
The implementation is done completely in Haskell and compares favourably with known implementations.
The implementation is based on the $G^{\#}$-machine, which is an extension of the G-machine that has been adapted to the needs of abstract reduction. Changing lazy evaluation (call-by-need) to eager evaluation (call-by-value) for specific functions has been proven to be necessary for good program performance. This has been *inter alia* pointed out as one result of doing work on benchmarking by Hartel [HFA+94]. In order to preserve lazy semantics, such changes in the order of evaluation may only be made for functions which have been proven to be strict. Strictness information enables a safe change of the reduction ordering, i.e. the termination behaviour of the program is unaffected.
Examples of how strictness information can be used are described in [HB93, PJP].
Although a large amount of research to lay theoretical foundations for strictness analysis, see e.g. [Bur91], has been carried out since the initiating work by Mycroft [Myc80], only a few results have become known with regard to practical implementations. Today, most compilers, as e.g. by the Glasgow team [PJHH+92], employ some very basic strictness analyser. The most efficient strictness analyser for Haskell [HPW+92] both in terms of speed and precision is claimed to be a recent implementation by Jensen et al [JHR94]. Their implementation of abstract interpretation using "chaotic" fixpoint iteration is implemented in C and has not yet been incorporated in any compiler. Another implementation has been given for Clean [NSvP91, PvE93]. This implementation is based on abstract reduction

[Nöc93]. It is also written in C. Unfortunately, it is hidden in the Clean compiler and, therefore, comparisons to other implementations are hardly possible.
We have developed an adaptation of abstract reduction to Haskell and have made an implementation of a strictness analyser for the core language as presented in [PJL91]. Our implementation is entirely written in Haskell.
It is realised via a newly developed abstract machine, the $G^{\#}$-machine. The source gets compiled into $G^{\#}$-code. The subsequent interpretation of $G^{\#}$-code then yields strictness information. This interpretation is done in the same way as in the G-machine.
Although our implementation is not fine tuned in terms of speed, it turned out to be very good both in terms of precision and efficiency. In a script of almost 290 functions all strictness information was found in a little over 30 seconds on a PC. We are quite positive that it can be tailored to be incorporated in any Haskell compiler. The implementation is freely available by anonymous ftp from ftp.uni-frankfurt.de in the directory /pub/dist/kist/functional/abs-g.

Fortunately, there is a growing body of literature on functional languages which thoroughly treats many of the fundamentals we will need. We will not quote the relevant parts, but refer the reader to this literature. The concepts of weak head normal form, $\bot$, supercombinators, supercombinator redexes (or redexes), the functional strategy, safety, reduction, normal order, lazy evaluation, patterns and pattern matching can be found in [Nöc93, Nöc92, Bur91, PJL91, BW88, PJ87].

We will call constructor patterns that only have variables as their arguments *simple patterns.*

As in [Nöc93], we use the notation $t@e\ldots$ as a shorthand for `let` $t = e$ `in` $\ldots$.

The language we analyse is the core language from [PJL91] enriched with `Bot` and `Top` for the abstract values $\bot^{\#}$ and $\top^{\#}$ respectively and the union notation `<...>`. Also, to be able to use more of our program as test file, we introduce characters, strings and the primitive functions `chr` and `ord` to the core language.

The G-machine has been a milestone in the development of models for the evaluation of functional languages. Its idea is to compile the expressions in the program into G-code, which when executed creates an instance of the expressions. The G-machine has been treated very well in many works, we will not try to add to these, instead we refer the reader to [PJL91, Bur91, Les88, PJ87, Aug84, Joh84].

It is often better to deviate from normal order, because early evaluation of subexpressions can speed up evaluation. What we must avoid, however, is:

- to increase the running time by evaluating unnecessary expressions and
- to change the semantics of the program. Such a change may result from a try to evaluate an unnecessary expression, which fails to terminate.

The definition of strictness gives a criterion for a function evaluating its argument.

Definition 1.1 *A function is* strict *in its argument iff its application to an argument without head normal form has no head normal form, that is iff* $f\ \bot = \bot$.

By reducing the halting problem to the problem of detecting strictness, one can easily see that strictness is undecidable.

The following chapter will present the method we implemented. Although it was conceived by Nöcker, we will speak of our method to avoid language clutter. We define abstract values by giving the concretization map. Furthermore, we define a less-equal relation on abstract values and, using it, an equivalence relation. We also introduce the notion of simple unions as well as abstract rewriting and reduction path analysis. In chapter 3 we describe our implementation of this method. We discuss the considerations and decisions characterizing our implementation to arrive at a more formal presentation in form of the operational semantics of the $G^{\#}$-machine. In chapter 4 we conclude by giving results of our implementation and an outlook on future work.

Chapter 2

Method

As we cannot expect the reader to be familiar with abstract reduction, we shall give a short review of [Nöc93].
In contrast to abstract interpretation, which abstracts the denotational semantics of a functional language, abstract reduction aims at an abstraction of the operational semantics, i.e., an abstract term is reduced until it hopefully yields the term "bottom". As area of investigation, an infinite domain is defined for a language completely determined by supercombinators and their applications:

> Let S be the set of all function symbols including data constructors defined in a program script, then the abstract domain will be the set of directed graphs over the set $S^{\#} = \{f^{\#} | f \in S\} \cup \{\bot^{\#}, \top^{\#}, \texttt{Union}^{\#}\}$ where $\bot^{\#}$ and $\top^{\#}$ are constants and $\texttt{Union}^{\#}$ is of arbitrary arity. An expression $\texttt{Union}^{\#}\ x_1 \ldots x_n$ will be written as $\langle x_1, \ldots, x_n \rangle$.

Detailed discussion of this domain can be found in [GH93]. Because of the intended meaning of the $\texttt{Union}^{\#}$ operator an ordering for this domain can only be given via a concretisation map γ. In line with the literature concerning abstract interpretation, we define γ as a map from abstract values to sets of values in the standard interpretation. For the sake of simplicity no type information is taken into account:

$$\begin{aligned}
\gamma(\bot^{\#}) &= \{\bot\} \\
\gamma(\top^{\#}) &= \{x | x \text{ is in the domain of the standard interpretation}\} \\
\gamma(\langle x_1, \ldots, x_n \rangle) &= \cup_{1 \leq i \leq n} \gamma(x_i) \\
\gamma(f^{\#}\ e_1 \ldots e_n) &= \{x | x \in \mathbf{S}\ [\![(f\ t_1 \ldots t_n)]\!], \mathbf{S}\ [\![t_i]\!] = t_i', t_i' \in \gamma(e_i)\}
\end{aligned}$$

where $\mathbf{S}\ [\![exp]\!]$ denotes the standard interpretation of exp. For anstract values which are not trees we define $\gamma(a) = \{x | \mathbf{S}\ [\![x]\!] = \mathbf{S}\ [\![y]\!], y \in \gamma(U(a))\}$, where $U(\cdot)$ is the unrolling of a graph. As ordering on the abstract domain we take set inclusions on the concretisations:

$$a_1 \leq_{\alpha} a_2 := \gamma(a_1) \subseteq \gamma(a_2)$$

This ordering provides information as to how specific a value is. The least specific one is $\top^\#$, which expresses all concrete values, the most specific one is $\bot^\#$, which represents just $\bot$ expressions. Generally, it cannot be decided whether $a \leq_\alpha b$. This would mean that $e \leq_\alpha \bot^\#$ can be decided for every abstract value e, which would give a decision precedure for strictness analysis.
An approximation $\leq'$ of $\leq_\alpha$ with $a \leq' b \Rightarrow a \leq_\alpha b$ can be given as follows:

$$\begin{array}{rcl} \bot^\# & \leq' & t \\ t & \leq' & \top^\# \\ t & \leq' & t \\ \texttt{f}\ x_1, \ldots, x_n & \leq' & \texttt{f}\ y_1, \ldots, y_n, \text{if } \forall 1 \leq i \leq n : x_i \leq' y_i \\ t & \leq' & \langle x_1, \ldots, x_n \rangle, \text{if } \exists 1 \leq i \leq n : t \leq' x_i \end{array}$$

In order to handle cyclic structures a refinement of this approximation is introduced. This refinement is a third argument, P, the *assumptions*:

$$\begin{array}{lll} \leq^\# (\bot^\#, t, P) & & \\ \leq^\# (t, \top^\#, P) & & \\ \leq^\# (t, t, P) & & \\ \leq^\# (t, t', P), & & \text{if } (t, t') \in P \\ \leq^\# (t@(fx_1 \ldots x_n), t'@(fy_1 \ldots y_n), P), & & \text{if } \forall 1 \leq i \leq n :\leq^\# (x_i, y_i, (t, t') : P) \\ \leq^\# (t, t'@\langle x_1, \ldots, x_n \rangle, P), & & \text{if } \exists 1 \leq i \leq n :\leq^\# (t, x_i, (t, t') : P) \\ \leq^\# (t@\langle x_1, \ldots, x_n \rangle, t', P), & & \text{if } \forall 1 \leq i \leq n :\leq^\# (x_i, t, (t, t') : P) \end{array}$$

Naturally, an equivalence relation holds between abstract values. Some important equivalences used to simplify unions are:

$$\begin{array}{rcll} \langle x \rangle & \equiv_\alpha & x & (2.1) \\ \langle x_1, \ldots, x_n \rangle & \equiv_\alpha & \langle \pi(x_1, \ldots, x_n) \rangle & (2.2) \\ & & \text{for a permutation } \pi & \\ \langle x, x_1, \ldots, x_n \rangle & \equiv_\alpha & \langle x_1, \ldots, x_n \rangle & (2.3) \\ & & \text{if } (\exists 1 \leq i \leq n) : x \leq_\alpha x_i & \\ C(\langle x_1, \ldots, x_n \rangle) & \equiv_\alpha & \langle C(x_1), \ldots, C(x_n) \rangle & (2.4) \\ x@\langle x, x_1, \ldots, x_n \rangle & \equiv_\alpha & \langle x_1, \ldots, x_n \rangle & (2.5) \\ \langle \langle x_1, \ldots, x_n \rangle, x'_1, \ldots, x'_m \rangle & \equiv_\alpha & \langle x_1, \ldots, x_n, x'_1, \ldots, x'_m \rangle & (2.6) \end{array}$$

Such equivalences on unions will allow us to simplify every abstract value to an equivalent abstract value which has at most one union construct and to shift this union operator up to the root of the value.[1] To define simplified unions, we write $\langle x_1, \ldots, x_n \rangle \setminus x$ for the union consisting of the x_is different from x:

Definition 2.1 (maxima)

$$M(X) := \{x \in X | y \in X \wedge x \leq_\alpha y \Rightarrow x = y\}$$

[1] Here we made some simplifications: abstract values are graphs. A `letrec` construct on the root of the abstract value can generally not be drawn into a union.

Using equivalence 2.1 we replace unions with one element only by their element.

Definition 2.2 (simple union) $M(x@\langle x_1, \ldots, x_n\rangle) \setminus x)$ *is the* simplification *of* $\langle x_1, \ldots, x_n\rangle$. Simple unions *are those which are already simplified.* Simple (abstract) values *are abstract values in which all unions are simple ones.*

Next, the concept of abstract reduction is introduced. An abstract value a_1 reduces to a_2, written $a_1 \rightarrow_\alpha a_2$, if for every expression which abstracts to a_1, a needed concrete reduction can be given, resulting in an expression that abstracts to a_2. A reduction is *needed* if for all reduction sequences which lead to *head normal form*, this reduction step has to be performed. For a formal introduction of this notion the reader is referred to the original work by Nöcker [Nöc93]. The general form of abstract reduction cannot be implemented, because it is not algorithmically defined. Therefore, algorithmic reduction-steps on abstract values are defined which approximate abstract reduction. These are two, namely abstract rewriting and reduction path analysis with bottom or cycle introduction.

- Abstract rewriting is just a sort of symbolic computation on abstract values. An expression $(f^\# \ a_1 \ldots a_n)$ is rewritten according to the rules defined for the supercombinator f. If more than one alternative of a function matches, the different results will be collected in a `Union`$^\#$ construct. The only question that arises is how to determine which function alternatives match. Since the language Nöcker uses (Clean) includes full pattern-matching, his matching rules are more complex than ours. In the Core language only simple patterns of one constructor with pattern variables are used. This makes the rules for matching much simpler. A decision algorithm for matching of such simple patterns is given by:

$$\begin{aligned}
\text{Match}^\#(\top^\#, p) &= \text{true} \\
\text{Match}^\#(\bot^\#, p) &= \text{false} \\
\text{Match}^\#(C\ v_1 \ldots v_n, C'\ p_1 \ldots p_m) &= \text{false, if } C \neq C' \\
\text{Match}^\#(C\ v_1 \ldots v_n, C\ p_1 \ldots p_m) &= \text{true} \\
\text{Match}^\#(\langle e_1, \ldots, e_n\rangle, p) &= \text{Match}^\#(e_i, p), \text{ if } \exists 1 \leq i \leq n
\end{aligned}$$

- Reduction path analysis is a kind of loop detection. If in a sequence of abstract rewritings an expression exp can be found, of which a generalisation exp' had been reduced before, we will be able to introduce a reduction cycle. Reducing exp can be done in the same way as reducing exp' *ad infinitum*. This cycle in the abstract reduction sequence will correspond to an infinite path in the concrete reduction, if it is needed, i.e., if a concretisation of exp has to be reduced in the concrete reduction. This will be the case, if exp is in every alternative of the reduced expression, i.e., if it is in every component of the resulting union of the case expression. A reduction cycle of a needed expression yields an infinite reduction in the concrete counterpart and therefore can be reduced to $\bot^\#$.

There are two rules for reduction path analysis. They differ in the requirement of needed terms: $\bot$-introduction requires a needed term, while cycle-introduction does not. $C(\cdot)$ denoting a context, the rules are:

cycle-introduction:

$$t \to_\alpha^* C(t') \wedge t' \leq_\alpha t \Rightarrow t \to_\alpha a@C(a).$$

$\bot$-introduction:

$$t \to_\alpha^* C(t') \wedge t' \leq_\alpha t \wedge t \text{ is needed } \Rightarrow t \to_\alpha \bot$$

Cycle-introduction is typically applied if the term from which we reduced is part of a union and not present in every component.

As an example for reduction path analysis we give the iterative length function:

```
length xs n  =  case xs of
                    <1> -> n;
                    <2> y ys -> length ys (n+1);
```

Strictness of the second argument can be found by the following abstract reduction:

$$\begin{aligned}
&\texttt{length}\ \top^\# \ \bot^\# \\
&\quad\to\ \langle \bot^\#, \texttt{length}\ \top^\#\ (\bot^\# \texttt{+1})\rangle \\
&\quad\equiv\ \texttt{length}\ \top^\#\ (\bot^\# \texttt{+1}) \\
&\quad\to\ \texttt{length}\ \top^\#\ \bot^\# \\
&\quad\to\ \bot^\#
\end{aligned} \tag{2.7}$$

A correctness proof for strictness analysis with abstract reduction for orthogonal term rewriting systems can be found in [Nöc92].

2.1 List domain and context information

The method is not restricted to WHNF strictness on $\bot$, i.e., more information for a function `f` than `f` $\bot = \bot$ can be generated. On recursive types like lists one might be interested in more information than the information that a list does not have a WHNF. So analysis can be made for functions, which have lists as arguments or lists as results.

2.1.1 4 point list domain

For lists there has been proposed a domain of four points, each representing a certain degree of where the first bottom of a list expression will be found. The four point domain on lists is defined as: $\{\bot, \infty, \bot\in, \top\in\}$ with the ordering

$\bot \sqsubseteq \infty \sqsubseteq \bot\in \sqsubseteq \top\in$ [Wad87]. $\bot$ represents expressions with no WHNF, ∞ represents expressions with no finite list structure, $\bot\in$ represents expressions with undefined list elements, and $\top\in$ all list expressions.
Fortunately, we are able to express these abstract list values:

```
topmem  =  ⊤#                          (⊤∈)
botmem  =  <Cons ⊥# topmem,            (⊥∈)
           Cons ⊤# botmem>;
   inf  =  Cons ⊤# inf;                (∞)
```

2.1.2 Context information

For a function `f` with a list result, we might be interested to know if reducing to a certain degree, e.g. to the list structure or to normal form, an application of `f` to some value, yields bottom. Such information is called context information [Bur91, Bur87].
We can generate context information with our machine by introducing evaluator functions [Bur91]. The only task of such evaluator functions is to force the reduction of its argument into a certain form in order to get a result in WHNF. An example of such an evaluator is[2]:

```
e_botmem xs  =  case xs of
                    Nil -> Nil;
                    (Cons  y ys) -> k1 (enf ys) y;


   k1 xs y  =  case xs of
                   _ ->   case y of
                              _ -> y;
```

The function `e_botmem` requires the reduction to the complete list structure of its argument and reduction of every list element to WHNF in order to produce a WHNF. Such evaluator functions allow us to reduce context analysis to strictness analysis of a certain abstract value.
Now it is possible to generate context information quite easily. We do not have to adapt the method or to enrich the machine but simply have to analyse certain functions for WHNF strictness. For example, we can generate context information for the standard `append` function by analysing:

```
(e_botmem (append botmem topmem))
```

If this expression gets abstractly reduced to $\bot^{\#}$, we know that `append` requires the reduction of its first argument to a list of WHNFs in order to produce such a list as result.
A further advantage is that different context information for a function can be generated separately.

[2]For reasons of readability we use list constructors Nil and Cons in this example and not the numbered Core language constructors.

Chapter 3

Implementation

In this chapter we will present the implementation of our method. We will be speaking of "our method" to avoid language clutter, although it was conceived by Nöcker. We will proceed by discussing the essential properties, considerations and decisions on which our implementation is based. A more formal presentation of our implementation in form of the operational semantics of the $G^{\#}$-machine will complete the chapter.

3.1 Abstract graph rewriting

Abstract graph rewriting and abstract reduction have a straightforward relation to graph rewriting and reduction respectively. Furthermore, graph reduction was successfully implemented with the G-machine model. From these two observations, it seems promising to design an abstract machine model for abstract graph rewriting based on the G-machine.

Our machine, the $G^{\#}$-machine, performs abstract graph rewriting and $\perp$- and cycle-introduction rather efficiently. One of the goals in its design was to keep much of the similarity to the G-machine from [PJL91], because among other reasons, this points up the necessary changes to implement abstract rewriting.

3.1.1 Abstract values

Obviously, the G-machine cannot handle the abstract values which are required to implement abstract graph rewriting. To be able to work with abstract values, the following parts of the G-machine need to be modified or introduced:

- instructions for primitive functions
- `Casejump` instruction
- `Split` instruction

- `MkUnion` instruction
- `Unwind` instruction
- `AbsEq` instruction

We will now look at these modifications in more detail.

Instructions for primitive functions

The primitive arithmetic, comparing and character-oriented instructions have to be modified. In the G-machine the program is assumed to be well typed, so these instructions may expect a value of suitable type. In the $G^{\#}$-machine, on the other hand, we have to distinguish between these values and abstract values. Therefore the primitive functions of the $G^{\#}$-machine are distinctly more complex than those of the G-machine.

Unions constitute the most complex case. Here we have to perform the primitive instruction for every element of the union, with this element on the top of the stack. Because these instructions change the heap and because we need to arrive at a stack in which all the addresses returned are valid heap addresses, we have to pass the resulting heap to the instruction for the next element. Finally, the results for the elements are collected in a new union.

The modification for the abstract value $\top^{\#}$ is straightforward, we simply return $\top^{\#}$.

For $\bot^{\#}$ we do not need to modify the primitive instructions if we require the values they operate on to be reduced to WHNF. The functions using our primitive instructions are then strict in their corresponding arguments. Therefore, using strictness information, $\bot^{\#}$ will be returned if one of the strict arguments is $\bot^{\#}$, without executing the primitive instruction.

`Split`

`Split` is used to obtain the arguments from a constructor and to put them onto the stack. In this way the binding of values to pattern variables is represented. `Split` can only occur as the first instruction of an alternative in a `Casejump` instruction. Unions as well as $\bot^{\#}$ are caught and handled by `Casejump`, therefore $\top^{\#}$ is the only abstract value `Split` can find on top of the stack. In this case a number of $\top^{\#}$s have to be put onto the stack according to the arity of the constructor. This corresponds to pattern matching with $\top$, binding every pattern variable to $\top$.

`Casejump`

Another respect in which the $G^{\#}$-machine and the G-machine differ is the `Casejump` instruction. We have to consider different cases according to the abstract value on the stack. The simplest case has $\bot^{\#}$ on top of the stack and $\bot^{\#}$ is returned.

In case $\top^\#$ is on the stack every alternative has to be executed with $\top^\#$ on the stack. This might result in several different results, which have to be collected in a union on the stack. Additionally, the path taken through each of these alternatives is stored in the path component of the resulting state. This serves debugging purposes only and can be omitted for everyday strictness analysis.

In case a union is on the stack we need to execute the `Casejump` for every value of the union and to collect the results in a union.

`MkUnion`

The `MkUnion` instruction is not present in the G-machine. It can be used to replace some values on the stack with their union. It was added to allow the compiler to generate code which explicitly creates unions. This in turn was done to add the `<>`-brackets to the core language, which, while not necessary for strictness analysis, helped very much in testing and debugging the implementation.

`Unwind`

Perhaps the biggest difference between G-machine and $G^\#$-machine can be observed in the implementation of the `Unwind` instruction, though only part of it is due to handling abstract values, the rest being due to the implementation of $\bot$- and cycle-introduction.

The values $\bot^\#$ and $\top^\#$ are treated by `Unwind` just as other values in (abstract) WHNF are.

Unions can be redexes or they can be in WHNF. If they are in WHNF, we treat them as we treat other values in WHNF. If they are redexes, then every element is a redex and has to be further reduced. In principle, we could follow one of two strategies: we could search the depth of the computational graph first or we could search its breadth first. Continuing to the depth first, i. e. reducing every element to WHNF before proceeding to the next element, will yield considerable disadvantages, which we will discuss in the section on $\bot$- and cycle-introduction, because that is where they show. Because of this, the $G^\#$-machine reduces every component one step further and then recollects them in a union, which replaces the original union on the stack.

`AbsEq`

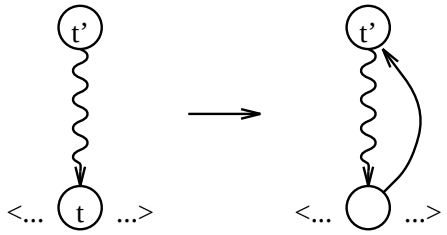
The `AbsEq` instruction is used to compare nodes instead of values in `Num` nodes. It is used to test for $\bot^\#$ on the stack.

3.1.2 Reduction path analysis

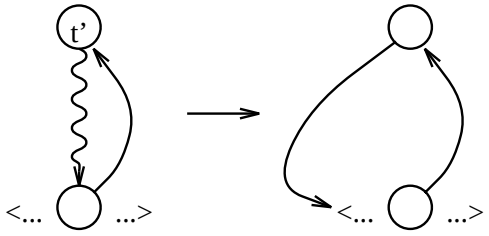
For our implementation we may view reduction path analysis as consisting of two phases. In the first phase we find candidates for $\perp$- and cycle-introduction, whereas in the second phase we decide which of the introductions to carry out.

A candidate for one of the introductions is a term t having a term t' in its computational history, such that $t \leq t'$. We do not yet know which, because we do not know if t is needed, but we can introduce an indirection to t'. Only when a root r is to be overwritten by a value a, do we test which of the introductions to carry out. If every reduction path represented by a contains the indirection to r we can introduce $\perp$, otherwise we introduce a cycle by overwriting r with an indirection to a.

We will demonstrate this with an example. Let t' be reduced to $C(t)$, where the context is a union, i. e. $C(t) = \texttt{<}\ldots, t, \ldots\texttt{>}$. Furthermore, let $t \leq^{\#} t'$, such that we replace t by an indirection to t'.



Later, if t' is to be overwritten and not all components of the union contain the indirection to t' we introduce a cycle by overwriting t' with an indirection to the union.



The modifications necessary for the implementation of reduction path analysis pertain to:

- the new state components `GmOpen` and `GmMarks`
- the type of the heap nodes
- considering marked nodes in `update`
- the contribution of `update` to reduction path analysis by calling `inEveryReductionPath`
- handling of `Ap`, `Ind` and `Union` nodes in `unwind`

State components `GmOpen` and `GmMarks`

Apparently, we need a means to access terms of the computational history. Therefore we provide a state component `GmOpen` to hold addresses of graphs which were further reduced, i. e. which have been on top of the stack for an `Unwind` instruction. It is not necessary to store arbitrary graph addresses in `GmOpen`, but it suffices to store addresses of redexes. The only kind of node that can be redexes apart from application nodes are unions, but for them the computation is eventually split into several paths, each starting with an application node on top of the stack. So there is also no need to store addresses of unions which are redexes in `GmOpen`. Likewise, it is not necessary to keep an address in `GmOpen`, once the reduction of the graph causing the address to be stored in `GmOpen` terminates by overwriting the root, because when the computational history of a graph ends so does that of all of its subgraphs. For this reason the addresses stored during said history cannot be candidates for one of the introductions anymore.

To summarize we can say that addresses of application nodes which are redexes are stored in `GmOpen` until the root from which they descend is in WHNF. These redexes will be called *open redexes*.

In our implementation only application nodes which are redexes are compared to graphs in `GmOpen` to find candidates for $\perp$- and cycle-introduction. By the discussion above it follows that this is sufficient. If a union appears in the computational history, several alternative computations have to be pursued. Here we can choose from a wide spectrum of different possibilities, ranging from trying to reduce each component to WHNF if one exists to applying the next instruction in each of the computations represented by the components.

Unions are often caused by recursive functions. If we would, for example, always choose the first component of a union to reduce it further, we might have to terminate the computation without having found strictness information. If the recursive rewrite is always put in the first component and the difference to earlier applications of this function is large enough for our $\leq^{\#}$-relation to fail in spite of the $\leq_{\alpha}$-relation holding between the terms, then we will eventually exhaust our resources and approximate with $\top^{\#}$.

But it is very well possible that the other rewrites, particularly the non-recursive ones, by overwriting the root of a subgraph used in several places help establish the $\leq^{\#}$-relation between terms and thus to find strictness information, which otherwise would not have been found.

A simple example would be the iterative `length` function:

```
length xs n  =  case xs of
                    <1> -> n;
                    <2> a b -> length b (n+1);
```

If we were to always choose the recursive alternative and try to reduce it to WHNF

we would get the following:

$$\begin{aligned}
\texttt{length}\ \top^{\#}\ \perp^{\#} &\rightarrow \langle \perp^{\#}, \texttt{length}\ \top^{\#}\ (\perp^{\#}\texttt{+1})\rangle \\
\texttt{length}\ \top^{\#}\ (\perp^{\#}\texttt{+1}) &\rightarrow \langle \perp^{\#}\texttt{+1}, \texttt{length}\ \top^{\#}\ ((\perp^{\#}\texttt{+1})\texttt{+1})\rangle \\
\texttt{length}\ \top^{\#}\ ((\perp^{\#}\texttt{+1})\texttt{+1}) &\rightarrow \langle (\perp^{\#}\texttt{+1})\texttt{+1}, \texttt{length}\ \top^{\#}\ (((\perp^{\#}\texttt{+1})\texttt{+1})\texttt{+1})\rangle \\
&\vdots
\end{aligned}$$

We could try to always choose a non-recursive alternative for further reduction. The problem with this is that it is not easily determined which alternatives are recursive and which are not.

What we can do, however, is to reduce each component a bit further and to collect the resulting graphs in a union. Now, every component of the union stands for a computational history with its open redexes, so we store the list of open redexes with it.

To reduce a component a bit further, first we mark the root of the corresponding subgraph, so that when we overwrite a root we can check if it was marked. This is done by storing the root of the subgraph in the state component `GmMarks`. If a marked root is overwritten we want to continue with a different component of our union. Therefore, we insert a `Join` instruction into the instruction sequence to the effect that evaluation is suspended and another component can be evaluated.

An additional advantage of this method is that after all components have been reduced a bit further and have been collected in a union, a simplification of the union can be performed, which shrinks the computational tree to be considered.

In summary we can say that a reduction step is performed on each component of a union using `GmMarks` and the open redexes stored with the component, followed by collecting the results in a union and simplifying this union, after which the process repeats.

We will turn towards another problem which stems from the fact that graphs in the heap having addresses in `GmOpen` can be further reduced. Such reduction could break the $\leq^{\#}$-relation even if it held before reduction. Again we will use the `length` example.

$$\begin{aligned}
\texttt{length}\ \top^{\#}\ \perp^{\#} &\rightarrow \langle \texttt{length}\ \top^{\#}\ b@(\perp^{\#}+1), \perp^{\#}\rangle \\
&\qquad \equiv_{\alpha} \texttt{length}\ \top^{\#}\ b@(\perp^{\#}+1) \\
&\rightarrow \langle \texttt{length}\ \top^{\#}\ (b@(\perp^{\#}+1)+1), b@(\perp^{\#}+1)\rangle \qquad (3.1) \\
&\rightarrow \langle \texttt{length}\ \top^{\#}\ ((b@\perp^{\#}+1)+1), b@\perp^{\#}+1\rangle
\end{aligned}$$

This looks promising, but meanwhile $b@(\perp^{\#}+1)$ has been reduced to $b@\perp^{\#}$ so in (3.1) we have $\langle \texttt{length}\ \top^{\#}\ (b@\perp^{\#}+1), b@\perp^{\#}\rangle \equiv_{\alpha} \texttt{length}\ \top^{\#}\ (b@\perp^{\#}+1)$ and for reduction path analysis we have to decide if $\langle \texttt{length}\ \top^{\#}\ ((\perp^{\#}\texttt{+1})\texttt{+1}),\ \perp^{\#}\texttt{+1}\rangle \leq^{\#} \texttt{length}\ \top^{\#}\ (\perp^{\#}+1)$. From the definition of $\leq^{\#}$ we get:

$$\langle \texttt{length}\ \top^{\#}\ ((\bot^{\#}+1)+1), \bot^{\#}+1\rangle \leq^{\#} \texttt{length}\ \top^{\#}\ (\bot^{\#}+1)$$
$$\iff \bot^{\#}+1 \leq^{\#} \bot^{\#}\ \wedge\ \bot^{\#}+1 \leq^{\#} \texttt{length}\ \top^{\#}\ (\bot^{\#}+1)$$

Both parts of the conjunction are false, but $\bot^{\#}+1 \leq_{\alpha} \bot^{\#}\ \wedge\ \bot^{\#}+1 \leq_{\alpha} \texttt{length}\ \top^{\#}\ (\bot^{\#}+1)$ holds.

Because $\bot^{\#}+1 \leq_{\alpha} \bot^{\#}$ only holds if $\bot^{\#}+1 = \bot^{\#}$, we find ourselves at the problem we started with. The second part of the conjunction is at least as hard as the first.

We see that for reduction path analysis it is important to store the open redexes before reduction takes place. Doing this in the example, line (3.1) is kept and we have to evaluate:

$$\langle \texttt{length}\ \top^{\#}\ ((\bot^{\#}+1)+1), \bot^{\#}+1\rangle \leq^{\#} \langle \texttt{length}\ \top^{\#}\ ((\bot^{\#}+1)+1), \bot^{\#}+1\rangle$$

which gives true.

In the following section we will describe how the open redexes are stored in our implementation.

Type of heap nodes

Every cell in the heap is implemented as two nodes, the first being the root node of the original subgraph and the second being the root node of a poossibly reduced graph. Thus we have access to the original redex without the need to copy. The space requirement of this approach is equivalent to copying, but copying has the disadvantage that the graph structure with possibly multiply used nodes has to be replicated.

`update` and reduction path analysis

Part of reduction path analysis is implemented in the `update` function. Here we distinguish marked from unmarked roots. `unwind` marks roots if alternative expressions have to be evaluated as with unions. `update` inserts a `Join` instruction in case a marked root is to be overwritten to suspend evaluation of the term. Furthermore, the mark on the root is removed. `update` removes marked as well as unmarked roots from the open redexes, `GmOpen`, as well as all indirections and chains of indirections leading to the root being overwritten. `update` also is the function to call `inEveryReductionPath`, which determines if a cycle or $\bot$ should be introduced. In case `inEveryReductionPath` finds an indirection to the root in every term represented by this root, $\bot$ can be introduced.

For the different kinds of nodes the following rules are applied:

- indirections are taken, as long as they do not point to the root

- for applications it suffices if the indirection to the root is in the function or the argument
- it also suffices if the indirection is found in one of the arguments to a constructor

All these cases represent *one* value or reduction path. Unions represent several reduction paths:

- in unions the indirection has to be present in every element
- no other node can contain the indirection

3.2 Neededness

In chapter 2 we saw that an essential property in the definition of $\perp$-introduction was neededness. Here, we will discuss how neededness can be elegantly implemented.

It is not hard to see that when following normal order only needed reductions will be performed. The difference between $\perp$- and cycle-introduction was merely that in one case we were operating on a needed term and in the other on a non-needed term.

Unions are the only nodes for which we possibly depart from normal order, such that we always perform needed reductions except possibly for unions. A term is needed in a union if it is needed in the evaluation of every one of its components. Now it becomes obvious why we split reduction path analysis into two phases: only after the candidates have been found can we determine which of the introductions to perform by examining if the candidate is referenced from every component.

Neededness of a term t is determined after t has been replaced by an indirection to an open redex in the function `inEveryReductionPath` by examining if the indirection is contained in every reduction path.

3.3 Storage and usage of strictness information

Another important aspect is storage and usage of strictness information, which will be discussed in this section.

To have strictness information about a function `f` means to know which of its arguments `f` needs for evaluation. How do we use this information in the strictness analysis itself? When an application of a function is to be evaluated we want to determine, using strictness information already available, if a strict position is $\perp^{\#}$. This is the only use of strictness information we make in the analysis. Here, we can also use the idea of the G-machine: we can generate $G^{\#}$-code to test strict

positions for $\perp^{\#}$, which will be executed prior to executing the $G^{\#}$-code of the original function. So, when we have found strictness information for a function `f`, we store it by:

- renaming `f` to `_f`
- creating a function `f`, which
  - tests the strict positions for $\perp^{\#}$
  - calls `_f` only if $\perp^{\#}$ is not found in any strict position
  - returns $\perp^{\#}$ if any strict position evaluates to $\perp^{\#}$

For this the $G^{\#}$-machine needs to be able to test for equality to $\perp^{\#}$. We can not use `Eq` for this purpose, since `Eq` is supposed to represent the semantics of the concrete `==`, i. e. `Eq` has to be strict in both of its arguments. So, obviously, `Eq` can not give any information on equality to the abstract value $\perp^{\#}$. We therefore create a new instruction `AbsEq` for which $\perp^{\#}$ `` `AbsEq` `` $\perp^{\#}$ holds. The $G^{\#}$-code for an n-ary function `f` with strictness $s = (s_1, \ldots, s_n)$ is `analyserCode 0 s f`, where

```
> analyserCode :: Int -> [Bool] -> Name -> GmCode
> analyserCode arg [] f =   [Pushglobal ('_':f)] ++
>           take arg (repeat Mkap) ++
>           [Update 0, Unwind]
> analyserCode arg (False:xs) f = analyserCode (arg+1) xs f
> analyserCode arg (True:xs) f  =
>   [Push arg,
>    Eval,
>    Pushglobal "Bot",
>    AbsEq,
>    Casejump [(2:= [Pushglobal "Bot",
>                    Update (arg+2+length xs),
>                    Pop (arg+2+length xs),
>                    Unwind]),
>              (1:= ((Pop 1):(analyserCode (arg+1) xs f)))]]
```

Here the `2` in the `Casejump` instruction represents "true", the case in which $\perp^{\#}$ is found. When a function containing an application of `f` is analysed, we use this strictness information by executing the $G^{\#}$-code.

3.4 Non-strict implementation

While in Nöcker's opinion [Nöc92] it is necessary to use strict evaluation, in our opinion it is not. The example given there at least does not make strict evaluation

necessary. We will also informally explain why it is not plausible that this should be so.

Nöcker argues that often a strict reduction strategy is better than a non-strict one, because otherwise certain cases of strictness will not be found. To demonstrate such a case he uses the popular `length` example and observes that strictness in the second argument will not be found using a non-strict reduction strategy. Here he argues that it is necessary to reduce $\perp^{\#}+1$ first in order to introduce $\perp^{\#}$ later. As we have shown in 3.1.2 this is not necessary in our implementation.

Why does it seem that strict evaluation is not necessary here? In the example, strict evaluation had an advantage because it made $\perp$-introduction possible. If the evaluation of an argument is needed in one of the reduction paths then it will also be performed in our implementation and we can also introduce $\perp$. If it is not needed in any of the reduction paths then it will not be performed in our implementation.

Possibly, one can find an example where:

- a strict position is examined
- there is a term t on another position, which is not needed in any resulting reduction path
- only reduction of t makes $\perp$-introduction possible

We conjecture that there is no such example.

3.5 Abandoning termination analysis

From the definition of strictness it follows that functions which never terminate are also strict, because then $f\ \perp = \perp$ holds. An example would be the function `g`:

```
g x = g (Cons{2,2} 1 x)
```

Our method can find this in many cases, still we do not look for this in the implementation, because these cases are rather contrived and therefore do not justify the time spent (in tests up to 50%). In a compiler one could, through use of directives, specify supercombinators for which non-termination should be analysed.

3.6 Operational semantics of the $G^{\#}$-machine

As in the G-machine, the $G^{\#}$-machine is a state transformation system, where the state is a 10-tuple:

$$\langle \omega, C, S, D, H, G, \sigma, \pi, O, M \rangle$$

Capitals stand for components which are indispensable for the operation of the $G^{\#}$-machine, while greek letters stand for components which are not necessary, but make the operation more evident. We have:

ω	for the output
C	for the instruction sequence
S	for the stack
D	for the dump
H	for the heap
G	for the global names
σ	for statistics
π	for paths
O	for open redexes
M	for marks

The function of these components is characterised in the state transformations for the instructions.

The instruction set of the $G^{\#}$-machine consists of:

`Eval`,
`Push` n, `Pushglobal` g, `Pushint` i,
`Mkap`, `MkUnion` n,`Pack` t n,
`Pop` n, `Slide` n,
`Add`, `Sub`, `Mul`, `Div`, `Neg`,
`Le`, `Ge` `Lt`, `Gt`, `Eq`, `Ne`,
`Ord`, `Chr`,
`Casejump` as,
`Print`,
`Alloc` n,
`AbsEq`,
`Join`,
`Update` n,
`Unwind`

The graph is represented by the heap, in which the following kinds of nodes are available:

`Num` n the number n.

`Ap` a_1 a_2 the application of the function pointed to by a_1 to the argument pointed to by a_2.

`Global` s c the instructions c of a supercombinator of arity s.

`Ind` a an indirection to another node.

`Constr` r as the rth constructor of a type together with the addresses of its arguments.

`Top` the abstract value $\top^{\#}$.

`Bot` the abstract value $\perp^{\#}$.

`Union` us the union of the abstract graphs in us. us is a list of pairs whose first component stands for the graph to unite and whose second component stands for the open redexes on the corresponding reduction path.

`Visited` an already visited node. (for dependency analysis)

In unions we also store the list of the open redexes for each component, so we extend the definition of $\setminus$ for unions and for maxima of unions in a canonical way.

Definition 3.1 *Let*

$$\langle (x_1, o_1), \ldots, (x_n, o_n) \rangle \setminus x := \langle (x_{k_1}, o_{k_1}), \ldots, (x_{k_s}, o_{k_s}) \rangle,$$

such that $\{x_{k_1}, \ldots, x_{k_s}\} = \{x_1, \ldots, x_n\} \setminus x$

Definition 3.2 *Let*

$$M(\langle (x_1, o_1), \ldots, (x_n, o_n) \rangle) := \begin{cases} \langle (x_{m_1}, o_{m_1}), \ldots, (x_{m_r}, o_{m_r}) \rangle & r \geq 2 \\ (x_{m_r}, o_{m_r}) & r = 1 \end{cases}$$

where $(\forall 1 \leq i \leq r) : x_{m_i} \leq_\alpha x_{m_j} \Rightarrow i = j$.

We obtain the extended definition of simple unions by using the new definitions for $\setminus$ and maxima in definition 2.2.

There is no instruction, which affects all state components. Where a transition does not depend on a particular component we write $*$ in the original state. A $*$ in the resulting state shows that the component has not changed. For readability we write the next instruction in front of the state, that is, instead of

$$\langle *, \texttt{Push 0} : C, s : S, *, \ldots, * \rangle \vdash \langle *, C, s : s : S, *, \ldots, * \rangle$$

we write

$$\texttt{Push 0} : \langle *, *, s : S, *, \ldots, * \rangle \vdash \langle *, *, s : s : S, *, \ldots, * \rangle$$

The final states of the $G^{\#}$-machine are the states in which no more instructions have to be executed and states which exhaust the resource allowance. Furthermore, states following a state whose instruction was `Join` are final.

3.6.1 Eval

The function *eval* maps a state to a final state by repeatedly applying state transitions until the final state is reached. The state transitions follow the subsequent rules.

The `Eval` instruction starts a new evaluation of the graph which has its root on top of the stack, unless it is $\top^\#$ or $\bot^\#$. The values $\top^\#$ and $\bot^\#$ should not be further reduced.

$$\texttt{Eval} : \langle *, C, a : S, D, H \left[a = \left\{ \begin{array}{l} \texttt{Top} \\ \texttt{Bot} \end{array} \right. \right], *, \ldots, * \rangle \vdash \langle *, \ldots, * \rangle$$

The other cases have to arrange for evaluation.

$$\begin{array}{rl} & \texttt{Eval} : \langle *, C, a : S, D, H \left[a \neq \left\{ \begin{array}{l} \texttt{Top} \\ \texttt{Bot} \end{array} \right. \right], *, \ldots, * \rangle \\ \vdash & \langle *, [\texttt{Unwind}], a, (C, S) : D, *, \ldots, * \rangle, \end{array}$$

`Eval` stores the context consisting of instruction sequence C and stack S in which the evaluation was started on the dump.

3.6.2 Push instructions

The `Push` instructions push a pointer onto the stack without changing the heap. An exception to this is `Pushint`[1].

$$\texttt{Push}\ n : \langle *, *, a_0 : \ldots : a_n : S, *, \ldots, * \rangle \vdash \langle *, *, a_n : a_0 : \ldots : a_n : S, *, \ldots, * \rangle$$

`Push` copies the nth stack entry to the top. `Pushglobal` performs analogously:

$$\texttt{Pushglobal}\ g : \langle *, *, S, *, *, G[g = a], *, \ldots, * \rangle \vdash \langle *, *, a : S, *, \ldots, * \rangle$$

For `Pushint` we distinguish the case where no new node is created

$$\texttt{Pushint}\ i : \langle *, *, S, *, *, G['i' = a], *, \ldots, * \rangle \vdash \langle *, *, a : S, *, *, *, \ldots, * \rangle$$

from the case in which a new node is created on the heap

$$\begin{array}{l} \texttt{Pushint}\ i : \langle *, *, S, *, H, G, *, \ldots, * \rangle \\ \quad \vdash \langle *, *, a : S, *, H', G', *, \ldots, * \rangle, \text{where } H' = H[a = \texttt{Num}\ i], G' = G['i' = a] \end{array}$$

where $'i'$ is a canonical name for i.

3.6.3 Creation of nodes

The instructions `MkAp`, `MkUnion` n and `Pack` t n serve the purpose of creating new nodes in the heap.

[1] In implementations using a separate stack for primitive values `Pushint` does not constitute an exception.

`MkAp` takes the top two stack entries and replaces them with an application node.

$$\texttt{Mkap} : \langle *, *, a_1 : a_2 : S, *, H, *, \ldots, * \rangle \vdash \langle *, *, a : S, *, H[a = \texttt{Ap}\ a_1\ a_2], *, \ldots, * \rangle$$

`MkUnion` n replaces the top n stack entries by their union.

$$\begin{aligned}\texttt{MkUnion}\ n : \langle *, *, a_1 : \ldots : a_n : S, *, H, *, \ldots, * \rangle \\ \vdash \langle *, *, a : S, *, H[a = \texttt{Union}\ [a_1, \ldots, a_n]], *, \ldots, * \rangle\end{aligned}$$

`Pack` r n packs the top n stack entries into the rth constructor.

$$\begin{aligned}\texttt{Pack}\ t\ n : \langle *, *, a_1 : \ldots : a_n : S, *, H, *, \ldots, * \rangle \\ \vdash \langle *, *, a : S, *, H[a = \texttt{Constr}\ r\ [a_1, \ldots, a_n]], *, \ldots, * \rangle\end{aligned}$$

3.6.4 `Pop` instructions

The `Pop` and `Slide` instructions remove entries from the stack. `Pop` n removes the top n entries.

$$\texttt{Pop}\ n : \langle *, *, a_1 : \ldots : a_n : S, *, \ldots, * \rangle \vdash \langle *, *, S, *, \ldots, * \rangle$$

`Slide` n removes n entries below the top of the stack.

$$\texttt{Slide}\ n : \langle *, *, a : a_1 : \ldots : a_n : S, *, \ldots, * \rangle \vdash \langle *, *, a : S, *, \ldots, * \rangle$$

3.6.5 Instructions for primitive functions

The arithmetic, the comparing and the character oriented instructions have to put $\top^\#$ on the stack, if one of their arguments is $\top^\#$.

$$\begin{aligned}\iota : \langle *, *, a_1 : a_2 : S, *, H[a_1 = \texttt{Top}], *, \ldots, * \rangle \\ \vdash \langle *, *, a_1 : S, *, \ldots, * \rangle, \\ \forall \iota \in \{\texttt{Add}, \texttt{Sub}, \texttt{Mul}, \texttt{Div}, \texttt{Lt}, \texttt{Le}, \texttt{Gt}, \texttt{Ge}, \texttt{Eq}, \texttt{Ne}\}\end{aligned}$$

analogously for a_2.

The corresponding holds for the `Neg`, `Ord` and `Chr` instructions, but these only use the top stack entry. In this case the one argument is equal to the result, so we do not need to change the stack at all.

$$\begin{aligned}&\iota : \langle *, *, a : S, *, H[a = \texttt{Top}], *, \ldots, * \rangle \quad \vdash \quad \langle *, \ldots, * \rangle \\ &\forall \iota \in \{\texttt{Neg}, \texttt{Ord}, \texttt{Chr}\}\end{aligned}$$

The primitive functions (the only place these instructions can occur) are all strict in their arguments. This guarantees that no such instructions will find $\bot^\#$ on the stack, because we use this strictness before calling a primitive function to decide whether to return $\bot^\#$ or not.

Unions will (possibly) produce new unions. We demand that unions be simplified, therefore we now define a function with which we can represent the simplification part of the transitions.

$$
\begin{array}{l}
v(\langle *, *, S, *, H, *, *, *, O, * \rangle, (u_1, O_1), \ldots, (u_n, O_n)) \\
\quad = \quad \langle *, *, u' : S, *, H', *, *, *, O', * \rangle, \\
\text{where } m = M(x@\texttt{<}(u_1, O_1), \ldots, (u_n, O_n)\texttt{>} \setminus x) \text{ in} \\
\qquad \left.\begin{array}{lcl} u' & = & u, \\ H' & = & H, \\ O' & = & O^*, \end{array}\right\} \text{ if } m = (u, O^*) \\
\qquad \left.\begin{array}{lcl} H' & = & H[u' = m], \\ O' & = & O, \end{array}\right\} \text{ otherwise}
\end{array}
$$

Let us observe the binary instructions.

$$
\begin{array}{l}
\iota : \langle *, *, a_1 : a_2 : S, *, H[a_1 = \texttt{<}(u_1, O_1), \ldots, (u_n, O_n)\texttt{>}], *, *, *, O, * \rangle \\
\quad \vdash v(\langle *, *, S, *, H^n, *, *, *, O, * \rangle, (u'_1, O'_1), \ldots, (u'_n, O'_n)) \\
\quad \forall (\iota, \odot) \in \{(\texttt{Add}, +), (\texttt{Sub}, -), (\texttt{Mul}, *), (\texttt{Div}, \setminus)\} \\
\quad \text{with} \\
\quad \iota : \overbrace{\langle *, *, u_i : a_2 : S, *, H^{i-1}, *, *, *, O_i, * \rangle}^{s^{i-1}} \\
\quad \vdash \underbrace{\langle *, *, u'_i : S, *, H^i, *, *, *, O'_i, * \rangle}_{s^i}
\end{array}
$$

And analogously for a_2. Also, for `Neg`, `Ord` and `Chr` the corresponding rules hold.

The remaining cases for all the instructions except `Ord` have `Num` nodes on the stack. `Ord` will have `Constr` t `[]` nodes on the stack in the remaining cases. We will distinguish arithmetic, comparing and character oriented instructions.

3.6.6 Arithmetic instructions

Arithmetic instructions perform the corresponding operation on the numbers in the nodes and create a `Num` node for the result on the stack.

$$
\begin{array}{l}
\iota : \langle *, *, a_1 : a_2 : S, *, H[a_1 = \texttt{Num } x; a_2 = \texttt{Num } y], *, \ldots, * \rangle \\
\quad \vdash \langle *, *, a : S, *, H[a = \texttt{Num } x \odot y], *, \ldots, * \rangle, \\
\quad \forall (\iota, \odot) \in \{(\texttt{Add}, +), (\texttt{Sub}, -), (\texttt{Mul}, *), (\texttt{Div}, \setminus)\}
\end{array}
$$

`Neg` is handled analogously, but with one entry from the stack.

$$
\begin{array}{l}
\iota : \langle *, *, a_1 : a_2 : S, *, H[a_1 = \texttt{Num } x; a_2 = \texttt{Num } y], *, \ldots, * \rangle \\
\quad \vdash \langle *, *, a : S, *, H[a = \texttt{Num } x \odot y], *, \ldots, * \rangle, \\
\quad \forall (\iota, \odot) \in \{(\texttt{Add}, +), (\texttt{Sub}, -), (\texttt{Mul}, *), (\texttt{Div}, \setminus)\}
\end{array}
$$

3.6.7 Comparing instructions

Comparing instructions have to be represented a little differently, because here we need to return the boolean value of the comparison. For boolean values we use 0-ary constructors one with the constructor number 1 for *false* and one with the number 2 for *true*.

$$\begin{aligned}
&\iota : \langle *, *, a_1 : a_2 : S, *, H[a_1 = \texttt{Num}\ x; a_2 = \texttt{Num}\ y], *, \ldots, * \rangle \\
&\quad \vdash \langle *, *, a : S, *, H[a = b(x \odot y)], *, \ldots, * \rangle, \\
&\quad \forall (\iota, \odot) \in \{(\texttt{Lt}, <), (\texttt{Le}, \leq), (\texttt{Gt}, >), (\texttt{Ge}, \geq), (\texttt{Eq}, =), (\texttt{Ne}, \neq)\}, \\
&\quad b(x) = \begin{cases} \texttt{Constr 1 []}, & x = \text{false} \\ \texttt{Constr 2 []}, & x = \text{true} \end{cases}
\end{aligned}$$

3.6.8 Character oriented instructions

The character oriented instructions are `Ord` and `Chr`.

$$\begin{aligned}
&\texttt{Ord} : \langle *, *, a : S, *, H[a = \texttt{Constr}\ t\ \texttt{[]}], *, \ldots, * \rangle \\
&\quad \vdash \quad \langle *, *, a' : S, *, H[a' = \texttt{Num}\ t], *, \ldots, * \rangle
\end{aligned}$$

Analogously, for `Chr`:

$$\begin{aligned}
&\texttt{Chr} : \langle *, *, a : S, *, H[a = \texttt{Num}\ n], *, \ldots, * \rangle \\
&\quad \vdash \quad \langle *, *, a' : S, *, H[a' = \texttt{Constr}\ n\ \texttt{[]}], *, \ldots, * \rangle
\end{aligned}$$

3.6.9 Branching instructions

The only branching instruction of the $G^{\#}$-machine is the `Casejump` instruction, which is also used to implement the `if` function in the core language.

Pattern matching is performed in the `Casejump` instruction. If $\bot^{\#}$ is on the stack $\bot^{\#}$ is the result of the matching, so no change to the state is needed.

$$\texttt{Casejump}\ cs : \langle *, *, a : S, *, H[a = \texttt{Bot}], *, \ldots, * \rangle \quad \vdash \quad \langle *, \ldots, * \rangle$$

If $\top^{\#}$ is found on the stack, all the alternatives apply, i. e. every instruction sequence has to be executed and the changes to the heap have to be passed on.

$$\begin{aligned}
&\texttt{Casejump}\ cs : \overbrace{\langle *, C, a : S, *, H[a = \texttt{Top}], *, \ldots, * \rangle}^{s^0} \\
&\quad \vdash \quad s^n, \text{ where} \\
&\qquad (\forall 1 \leq i \leq r) : s^{i-1} = \langle *, C, *, \ldots, * \rangle \\
&\qquad \Rightarrow s^i = eval\ \langle *, c_i +\!\!+ [Join] +\!\!+ C, *, \ldots, * \rangle
\end{aligned}$$

For unions this becomes a bit more complex. Here we have to execute the `Casejump` instruction for every component of the union.

$$\begin{array}{l}\overbrace{\texttt{Casejump}\ cs : \langle *, C, a : S, *, H[a = \texttt{<}(u_1, O_1), \ldots (u_r, O_r)\texttt{>}], *, \ldots, * \rangle}^{s^0} \\ \quad \vdash \quad v(\langle *, *, S, *, H^r, *, \ldots, * \rangle, (u'_1, O'_1), \ldots, (u'_r, O'_r)), \text{ where} \\ \qquad (\forall 1 \leq i \leq r) : s^{i-1} = \langle *, C, a : S, *, H^{i-1}, *, \ldots, * \rangle \\ \qquad \Rightarrow s^i = eval\ \langle *, \texttt{Casejump}\ cs : \texttt{Join} : C, u_i : S, *, H^{i-1}, *, *, *, O_i, * \rangle \\ \qquad\qquad = \langle *, C, u'_i : S, *, H^i, *, *, *, O'_i, * \rangle \end{array}$$

If an indirection is found on the stack, the $G^{\#}$-machine has to follow it.

$$\begin{array}{l}\texttt{Casejump}\ cs : \langle *, C, a : S, *, H[a = \texttt{Ind}\ a'], *, \ldots, * \rangle \\ \quad \vdash \quad \langle *, \texttt{Casejump}\ cs : C, a' : S, *, \ldots, * \rangle \end{array}$$

If a constructor is on the stack, the appropriate instruction sequence can be chosen according to its number, to be executed before the rest of the instructions.

$$\begin{array}{l}\texttt{Casejump}\ [t_1 := c_1, \ldots, t_n := c_n] : \langle *, C, a : S, *, H[a = \texttt{Constr}\ t\ cs], *, \ldots, * \rangle \\ \quad \vdash \quad \langle *, c' : C, a : S, *, \ldots, * \rangle, \text{ where} \\ \qquad c' = \begin{cases} c_i, & \exists 1 \leq i \leq n : t_i = t \\ [\texttt{Unwind}], & \text{otherwise} \end{cases} \end{array}$$

3.6.10 Print

The `Print` instruction converts the node on the stack to `Iseq`s, which then can be output as a string.

$\texttt{Print} : \langle \omega, *, a : S, *, H[a = n], *, \ldots, * \rangle \quad \vdash \quad \langle \omega +\!\!+ s(n), *, S, *, \ldots, * \rangle$, where
$+\!\!+$ is the concatenation of `Iseq`s and s the function to show nodes.

3.6.11 Alloc

`Alloc` n is used to allocate n nodes on the heap and to put their n addresses on the stack. This instruction is used to compile `letrec` constructs, so they allocate the nodes for the local definitions. The nodes are initialized with `Num 1`.

$$\begin{array}{l}\texttt{Alloc}\ n : \langle *, *, S, *, H, *, \ldots, * \rangle \\ \quad \vdash \quad \langle *, *, a_1 : \ldots : a_n : S, *, H \left[\left\{ \begin{array}{c} a_1 \\ \vdots \\ a_n \end{array} \right\} = \texttt{Num 1} \right], *, \ldots, * \rangle \end{array}$$

3.6.12 AbsEq

In contrary to `Eq` (see above), `AbsEq` can be used to compare abstract values. First of all `AbsEq` follows indirections.

$$\begin{array}{l}\texttt{AbsEq} : \langle *, C, a_1 : a_2 : S, *, H[a_1 = \texttt{Ind}\ a_1'], *, \ldots, * \rangle \\ \quad \vdash \quad \langle *, \texttt{AbsEq} : C, a_1' : a_2 : S, *, \ldots, * \rangle\end{array}$$

analogously for a_2. Otherwise `AbsEq` returns the value of comparing the nodes.

$$\begin{array}{l}\texttt{AbsEq} : \langle *, C, a_1 : a_2 : S, *, H[a_1 = \nu_1; a_2 = \nu_2], *, \ldots, * \rangle \\ \quad \vdash \quad \langle *, \texttt{AbsEq} : C, a' : S, *, H[a' = b(\nu_1 == \nu_2)], *, \ldots, * \rangle, \text{ where} \\ \qquad b(x) = \begin{cases} \texttt{Constr}\ 1, & x = \text{ false} \\ \texttt{Constr}\ 2, & x = \text{ true} \end{cases}\end{array}$$

3.6.13 Join

The `Join` instruction does not change the state, it merely provides for the next state being a final one.

$$\texttt{Join} : \langle *, \ldots, * \rangle \quad \vdash \quad \langle *, \ldots, * \rangle$$

3.6.14 Update

The `Update` n and the `Unwind` instruction are the mightiest instructions in the $G^{\#}$-machine.

`Update` removes marks from the roots it updates and if the root belongs to the open redexes it and all the redexes opened after it are removed froom the open redexes. It tests if $\perp$-introduction can take place and if the root was marked arranges for the evaluation to be suspended.

$$\texttt{Update}\ n : \langle *, C, a : a_0 : \ldots : a_n : S, *, H \left[\left\{ \begin{array}{c} h_1 \\ \vdots \\ h_s \end{array} \right\} \ \text{ind}^* a \right], *, *, *, O, M \rangle$$

$$\vdash \quad \langle *, C', a : a_0 : \ldots : a_n : S, *, H[a_n = \nu], *, *, *, O', M \setminus a_n \rangle, \text{ where}$$

$$h\ \text{ind}^i a \iff (\exists b_1, \ldots, b_i) : h = b_1 \wedge H \left[\begin{array}{lcl} b_1 & = & \texttt{Ind}\ b_2 \\ & \vdots & \\ b_i & = & \texttt{Ind}\ a \end{array} \right]$$

$$h\ \text{ind}^* a \iff (\exists i) : h\ \text{ind}^i a$$

$$\nu = \begin{cases} \texttt{Bot}, & \text{if every reduction path of } a \text{ contains an indirection to} \\ & a_n\ . \\ \texttt{Ind}\ a, & \text{otherwise} \end{cases}$$

$$C' = \begin{cases} \texttt{Join} : C, & \text{if } a_n \in M \\ C, & \text{otherwise} \end{cases}$$

$$O' = \begin{cases} [o_{i+1}, \ldots, o_m] \cap [h_1, \ldots, h_s], & \text{if } O = [o_1, \ldots, o_m] \wedge \\ & (\exists 1 \leq i \leq m) : o_j = a \\ O, & \text{otherwise} \end{cases}$$

3.6.15 Unwind

`Unwind` first of all distinguishes between an empty and a non-empty dump. For an empty dump and a term in WHNF or $\bot^{\#}$ on the stack no changes are needed.

$$\texttt{Unwind} : \langle *, *, a : S, [], H \left[a = \left\{ \begin{array}{l} \texttt{Top} \\ \texttt{Bot} \\ \texttt{Num}\ n \\ \texttt{Constr}\ t\ as \end{array} \right. \right], *, \ldots, * \rangle \quad \vdash \quad \langle *, \ldots, * \rangle$$

For a non-empty dump we move the code and stack contents from the dump to the state.

$$\texttt{Unwind} : \langle *, *, a : S, (C', S') : D, H \left[a = \left\{ \begin{array}{l} \texttt{Top} \\ \texttt{Bot} \\ \texttt{Num}\ n \\ \texttt{Constr}\ t\ as \end{array} \right. \right], *, \ldots, * \rangle$$

$$\vdash \quad \langle *, C' +\!\!+ C, a : Z, D, *, \ldots, * \rangle, \text{ where}$$

$$Z = \begin{cases} S', & \text{if } H \left[a = \left\{ \begin{array}{l} \texttt{Top} \\ \texttt{Bot} \end{array} \right. \right] \\ S +\!\!+ S', & \text{otherwise} \end{cases}$$

Application nodes are replaced by an indirection to one of the open redexes in case the application is less than the already open redex. In this case we also clear the code or replace it with a single `Unwind` if the application is marked or unmarked, respectively. If the application is not less than any open redex its first pointer is pushed onto the stack and an `Unwind` instruction is inserted in front

of the instruction sequence. If this application is a redex it is added to the open redexes.

$$
\begin{array}{l}
\texttt{Unwind} : \langle *, C, a : S, *, H[a = \texttt{Ap}\ a_1\ a_2], *, *, *, O, M\rangle \\
\quad \vdash \langle *, C', S', *, H', *, *, *, O', *\rangle, \\
\qquad S' = \left\{ \begin{array}{ll} a : S, & \text{if } (\exists o \in O) : a \leq^{\#} o \\ a_1 : a : S, & \text{otherwise} \end{array} \right. \\
\qquad H' = \left\{ \begin{array}{ll} H[a = \texttt{Ind}\ o], & \text{if } (\exists o \in O) : a \leq^{\#} o \\ H, & \text{otherwise} \end{array} \right. \\
\qquad C' = \left\{ \begin{array}{ll} [], & \text{if } (\exists o \in O) : a \leq^{\#} o \wedge a \in M \\ [\texttt{Unwind}], & \text{if } (\exists o \in O) : a \leq^{\#} o \wedge a \notin M \\ \texttt{Unwind} : C, & \text{otherwise} \end{array} \right. \\
\qquad O' = \left\{ \begin{array}{ll} a : O, & \text{if } (\forall o \in O) : a \not\leq^{\#} o \wedge \texttt{Ap}\ a_1\ a_2 \text{ ist Redex} \\ O, & \text{otherwise} \end{array} \right.
\end{array}
$$

For indirections we only have to determine if the address pointed to is that of an open redex.

$$
\begin{array}{l}
\texttt{Unwind} : \langle *, C, a : S, *, H[a = \texttt{Ind}\ a_1], *, *, *, O, *\rangle \\
\quad \vdash \langle *, \texttt{Unwind} : C, a' : S, *, H', *, \ldots, *\rangle, \text{ where} \\
\qquad \left. \begin{array}{rcl} H' & = & H[a' = \texttt{Bot}], \text{ if } a \in O \\ a' & = & a_1, \\ H' & = & H, \end{array} \right\} \text{ otherwise}
\end{array}
$$

For unions we have to test if one of the components already exhausts the resources or not and if it is a redex or not.

$$
\begin{array}{l}
\texttt{Unwind} : \langle *, C, a : S, *, H[a = \texttt{<}(u_1, O_1), \ldots, (u_n, O_n)\texttt{>}], *, *, *, O, *\rangle \\
\quad \vdash \langle *, [], *, \ldots, *\rangle, \text{ if} \\
\qquad (\exists 1 \leq i \leq n) : (u_i, O_i) \text{ exhausts the resources.}
\end{array}
$$

If this is not the case, i. e. if the resources are not yet exhausted, we have:

$$
\begin{array}{l}
\texttt{Unwind} : \langle *, C, a : S, *, H[a = x\texttt{@<}(u_1, O_1), \ldots, (u_n, O_n)\texttt{>}], *, *, *, O, *\rangle \\
\quad \vdash v(\langle *, [\texttt{Unwind}], S, *, H^n, *, \ldots, *\rangle, (u_1, O'_1), \ldots, (u_n, O'_n)), \\
\qquad \text{if } x \text{ is a redex and} \\
\qquad (1 \leq i \leq n) : \mathit{eval}\ \langle *, \texttt{Unwind} : C, u_i : S, *, H^{i-1}, *, *, O_i, u_i : M\rangle \\
\qquad\qquad\qquad = \langle *, *, *, *, H^i, *, *, O'_i, *\rangle
\end{array}
$$

Here, the same heap addresses as in the original state, u_i, can be used in the resulting state, because evaluation of the marked root continues until the root u_i is updated. If none of the preceding cases are applicable the state is transformed as for `Constr` or `Num` nodes.

If a `Global` node is encountered on the stack and the stack contains sufficient

entries for the application of the supercombinator, we get:

$$\texttt{Unwind} : \langle *, C, a : a_1 : \ldots : a_n : S, *, H \left[\begin{array}{lcl} a & = & \texttt{Global } n\ C' \\ a_n & = & \texttt{Ap } a_{n-1}\ a'_n \\ & \vdots & \\ a_1 & = & \texttt{Ap } a\ a'_1 \end{array} \right], *, \ldots, * \rangle$$
$$\vdash \quad \langle *, C' +\!\!+ C, a'_1 : \ldots : a'_n : S, *, *, \ldots, * \rangle$$

If fewer than the required number of entries are on the stack we apply the same transition as for `Constr` or `Num` nodes.

Chapter 4

Conclusion

Our implemetation efficiently approximates abstract reduction with reduction path analysis. The $G^{\#}$-machine, a new machine model based on the G-machine, systematically presents the method used. The degree of similarity with the G-machine which we were able to uphold indicates a correspondence between our method and reduction in functional languages correspond.

Although the implementation favors ease of understanding over efficiency it proves that abstract reduction with reduction path analysis is fit for every-day strictness analysis, even when implemented in a functional language and that it finds strictness information, which implementations of other methods do not find.

It is possible to optimize our implementation in several respects, there are even parts executed for every $G^{\#}$-machine instruction simulated where optimization is possible. Cautiously estimating, it should not be very difficult to halve the running time.

4.1 Results

The implementation was compiled and executed on different platforms using the Chalmers Haskell B. Compiler and the Glasgow Haskell Compiler.

The test file (see also appendix A) contained

- the `standard.prelude` of the Gofer interpreter except the floating-point functions
- the compiler from core language to $G^{\#}$-code
- the lexer for the core language
- some utility functions

The following table gives timings [Par95] for analysing this file. The output of the

command

`time Analyser -gc-gen` ⟨name⟩`.core 2>&1 >`⟨name⟩`.strict`

can be found in appendix B.

Platform	GHC	HBC[1]
Linux 486DX2/66		33s
HP9000/735	10.7s	
SPARC 10/41	19.6s	20.8s
DEC 3000/600	11.4s	

In a further test we analysed the `concat` function in its definition by `(foldr (++) [])` for complete context information in the four point list domain, i.e., we analysed `concat` $\bot$, `concat` ∞, `concat` $\bot\in$ and `concat` $\top\in$ in every one of the evaluators $\xi_\bot$, ξ_∞ and $\xi_\in$. In our test all context information was correctly found in 0.67 seconds user time on the Linux machine.

4.2 Future work

4.2.1 Modularization

In general in functional languages as in other languages one does not put the entire program text into one file, but uses modules to structure it. It is not difficult to extend the analyser to work with modules. The output of the analysis of one module can be read in and we can generate $G^\#$-code for these already analysed functions, which performs quite similar to `analyserCode`. The difference is that here we would not want to call the $G^\#$-code of the original function, in case none of the strict arguments evaluates to $\bot^\#$, but we would want to return $\top^\#$. Obviously, much information is thrown away doing so, which is available for functions in the same module and this does have a negative effect on the results. Let us consider an example from the `standard.prelude`:

```
iterate f x     =  Cons{2,2} x (iterate f (f x));
takeUntil p ys  =  case ys of
                      <1> -> Cons{1,0};
                      <2> a b -> if (p a)
                                    (Cons{2,2} a Cons{1,0})
                                    (Cons{2,2} a (takeUntil p b));
until´ p f x    =  takeUntil p (iterate f x);
```

We examine the first argument of `until´` when `takeUntil` is already analysed:

`until´` $\bot^\#$ $\top^\#$ $\top^\#$ $\rightarrow_\alpha$ `takeUntil` $\bot^\#$ `(iterate` $\top^\#$ $\top^\#$`)`

Our implementation will have detected strictness in `takeUntil`s second argument. So we approximate with $\top^\#$, because none of the strict arguments evaluate to $\bot^\#$.

`takeUntil` $\bot^\#$ `(iterate` $\top^\#$ $\top^\#$`)` $\rightarrow_\alpha$ $\top^\#$

In this case we will miss strictness in `until´s` first argument.

If, on the other hand, we use the rewrite rules for `takeUntil` and `iterate`

```
takeUntil ⊥# (iterate ⊤# ⊤#)
  →α  if (⊥# ⊤#)
         (Cons{2,2} ⊤# Cons{1,0})
         (Cons{2,2} ⊤# (takeUntil ⊥#(iterate ⊤# (⊤# ⊤#))))
  →α  ⊥#, because (⊥# ⊤#) is on a strict position
```

In this case we find strictness in `until´s` first argument.

Where does this difference come from?

`takeUntil` is not strict in its first argument, but

```
\x -> takeUntil x (iterate ⊤# ⊤#)
```

is. In the first case we lose this information by approximation, but in the second case where we continue to abstractly reduce we eventually find strictness.

What would be needed is a concept which elegantly extends our implementation to handle modules.

4.2.2 Variation of the machine model

The $G^{\#}$-machine has seen considerable development by many authors since it was first presented by Augustsson and Johnsson. This development was usually aiming at greater efficiency. Based on the above relationship between abstract reduction and reduction one might ask to what extent newer models like the ABC-machine, the $\langle \nu, G \rangle$-machine, or the STG-machine can be used for abstract reduction. From the above, one would expect abstract reduction to be performed more efficiently in these newer models.

4.2.3 Compiling

The analysis spends a lot of its time in simulating $G^{\#}$-machine instructions. It seems sensible to generate machine- or C-code instead of simulating these instructions. Particularly for large programs this could give a noticeable improvement in performance.

4.2.4 Use of several methods

Another possibility to gain noticeably in speed is to use several methods. In specific cases the compiler to $G^{\#}$-code can detect strictness very easily (see also

[PJL91]). Obviously, it is advantageous to use this strictness information and to only analyse the arguments that were not found to be strict by these simple means.

Are there cases in which such simple means can establish that a function cannot be strict in an argument? The answer is "yes", for example:

```
k a b = a
```

Here we can easily determine that `k` cannot be strict in `b`,[2] because `b` is not used on the right-hand side. Another case is functions having their right-hand side in WHNF. E. g. if the right-hand side is a constructor with its arguments or an application with less arguments than the functions arity. It remains open in which other cases there might be simple methods to find strictness information.

Acknowlegements

I would like to thank Eric Nöcker for reading and commenting on my german masters thesis. Also, this work owes much to Sven Eric Panitz and Manfred Schmidt-Schauß for their fruitful ideas, for taking the time to communicate them and for commenting on this work at all its stages. I would like to thank Will Partain and Jim Mattson for making the "registerised" GHC available for our HP workstations and for timing the execution of our implementation on SPARC and Alpha platforms.

[2] `k` could be strict in `b` according to our definition, only if `k` would not terminate at all, but we could not gain any performance from this information.

Appendix A

Source for tests

This appendix gives the file used for testing. The functions are taken from our implementation itself as well as from the `standard.prelude`. It contains:

- the compiler from core language to $G^{\#}$-code
- the lexer for the core language
- some utility functions
- the `standard.prelude` of the Gofer interpreter, except floating-point functions and functions in type classes

Together with the functions already in the analyser 287 functions will be analysed.

```
help = "press :? for a list of commands";

const k x       = k;
id    x         = x;
curry f a b     =  f (Cons{1,2} a b);
uncurry f c = case c of
                <1> a b -> f a b;

fst y   = case y of
                <1> x1 x2 -> x1;

snd y   = case y of
                <1> x1 x2 -> x2;

fst3 y  = case y of
                <1> x1 x2 x3 -> x1;

snd3 y  = case y of
                <1> x1 x2 x3 -> x2;

thd3 y  = case y of
                <1> x1 x2 x3 -> x3;
```

```
flip  f x y     = f y x;

-- Boolean functions: -------------------------------------------------------

binand x y = case x of
               <1> -> Cons{2,0};
               <2> -> x;

binor x y = case x of
               <1> -> x;
               <2> -> Cons{2,0};

not x = case x of
        <2> -> Cons{1,0};
        <1> -> Cons{2,0};

and xs = foldr binand Cons{2,0} xs;
or xs = foldr binor Cons{1,0} xs;

any p xs = or ((map p) xs);
all p xs = and ((map p) xs);

otherwise       = Cons{2,0};

-- Character functions: -----------------------------------------------------

isAscii c     =  ord c < 128;

isControl c   =  binor (c < ' ') (c == '\DEL');

isPrint c     =  binand (c >= ' ') (c <= '~');

isSpace c     =  binor (c == ' ') (binor (c == '\t') (binor (c == '\n')
                        (binor (c == '\r') (binor (c == '\f') (c == '\v')))));

isUpper c     =  binand (c >= 'A') (c <= 'Z');
isLower c     =  binand (c >= 'a') (c <= 'z');

isAlpha c     =  binor (isUpper c) (isLower c);
isDigit c     =  binand (c >= '0') (c <= '9');
isAlphanum c  =  binor (isAlpha c) (isDigit c);

toUpper c = if (isLower c) (chr ((ord c - (ord 'a')) + (ord 'A'))) c;
toLower c = if (isUpper c) (chr ((ord c - (ord 'A')) + (ord 'a'))) c;

minChar = chr 0;
maxChar = chr 255;

even x = ((x / 2) * 2) == x;
odd x  = not(even x);

rem x y  = x - (x / y) * y;
gcd' x y = if (y == 0) x (gcd' y (rem x y));
gcd x y  = gcd' (abs x) (abs y);

lcm x y  = if (y == 0) 0 (if (x == 0) 0 (abs ((x / (gcd x y)) * y)));
```

```
power x y     = if (y == 0) 1 (power´ x (y - 1) x);
power´ x n y  = if (n == 0) y (power´´ x n y);
power´´ x n y = if (even n) (power´´ (x * x) (n / 2) y)
(power´ x (n - 1) (x * y));

abs x = if (x >= 0) x (negate x);

signum x = if (x == 0) 0 (if (x > 0) 1 (negate 1));

plus x y  = x + y;
minus x y = x - y;
mul x y   = x * y;

sum xs     = foldl´ plus 0 xs;
product xs = foldl´ mul 1 xs;

sums xs     = scanl plus 0 xs;
products xs = scanl mul 1 xs;

-- Standard list processing functions: ---------------------------------------

head x = case x of
             <2> y z -> y;

last x = case x of
             <2> y z -> (case z of
                             <1> -> y;
                             <2> a b -> last z);

tail x = case x of
             <2> y z -> z;

init x = case x of
             <2> y z -> (case z of
                             <1> -> Cons{1,0};
                             <2> a b -> Cons{2,2} y (init z));

append xs ys = case xs of
                   <1> -> ys;
                   <2> z zs -> Cons{2,2} z (append zs ys);

length xs = foldl´ lam_length 0 xs;
lam_length a b = a + 1;

at x y = case x of <2> a b -> if (y == 0) a (at b (y - 1));

iterate f x = Cons{2,2} x (iterate f (f x));
repeat x    = Cons{2,2} x (repeat x);
cycle xs    = letrec xs´ = append xs xs´ in xs´;
copy n x    = take n (repeat x);
nub x = case x of
                <1> -> Cons{1,0};
                <2> y z -> Cons{2,2} y (nub (filter ((compose not eq) y) z));

reverse xs = foldl (flip cons22) Cons{1,0} xs;
```

```
eq x y = x == y;

elem x xs = any (eq x) xs;
notElem x xs = not (elem x xs);

max x y = if (x < y) y x;
min x y = if (x < y) x y;

maximum xs = foldl1 max xs;
minimum xs = foldl1 min xs;

concat xs = foldr append Cons{1,0} xs;

cons22 x xs = Cons{2,2} x xs;

transpose xs         = foldr lam_transpose Cons{1,0} xs;
lam_transpose xs xss = zipWith cons22 xs (append xss (repeat Cons{1,0}));

null x = case x of
             <1> -> Cons{2,0};
             <2> a b -> Cons{1,0};

delone xs ys = foldl del' xs ys ;
del' x y = case x of
              <1> -> Cons{1,0};
              <2> a b -> if (a == y) b (Cons{2,2} x (del' b y));

map f xs = case xs of
              <1> -> Cons{1,0};
              <2> a b -> Cons{2,2} (f a) (map f b);

filter p xs = case xs of
                 <1> -> Cons{1,0};
                 <2> a b -> let xs' = filter p b in
                        if (p a) (Cons{2,2} a xs') xs';

foldl f z xs = case xs of
                  <1> -> z;
                  <2> a b -> foldl f (f z a) b;

foldl1 f ys = case ys of
                 <2> x xs -> foldl f x xs;

foldl' f a ys = case ys of
                   <1> -> a;
                   <2> x xs -> strict (foldl' f) (f a x) xs;

scanl f q xs  = case xs of
                   <1> -> Cons{2,2} q Cons{1,0};
                   <2> x xs1 ->  Cons{2,2} q (scanl f (f q x) xs1);

scanl1 f ys   = case ys of
                   <2> x xs -> scanl f x xs;

scanl' f q xs = case xs of
```

```
                    <1> -> Cons{2,2} q Cons{1,0};
                    <2> x xs1 ->
                       Cons{2,2} q (strict (scanl´ f) (f q x) xs1);

foldr f z ys = case ys of
                   <1> -> z;
                   <2> x xs -> f x (foldr f z xs);

foldr1 f ys  = case ys of
                   <2> x xs -> (case xs of
                                    <1> -> x;
                                    <2> a b -> f x (foldr1 f xs));

scanr f q0 ys = case ys of
                    <1> -> Cons{2,2} q0 Cons{1,0};
                    <2> x xs ->
                       (case (scanr f q0 xs) of
                            <2> q qs ->
                               Cons{2,2} (f x q) (Cons{2,2} q qs));

scanr1 f ys = case ys of
                <2> x xs ->
                   (case xs of
                        <1> -> x;
                        <2> a b ->
                           (case (scanr1 f xs) of
                                <2> q qs ->
                                   Cons{2,2} (f x q) (Cons{2,2} q qs)));

take n ys = case (n == 0) of
                <2> -> Cons{1,0};
                <1> -> (case ys of
                            <1> -> Cons{1,0};
                            <2> x xs -> Cons{2,2} x (take (n - 1) xs));

drop n ys = case (n == 0) of
                <2> -> ys;
                <1> -> (case ys of
                            <1> -> Cons{1,0};
                            <2> x xs -> drop (n - 1) xs);

lam_splitAt n ys =
 case ys of
     <1> -> Cons{1,2} Cons{1,0} Cons{1,0};
     <2> x xs -> (case splitAt (n - 1) xs of
                      <1> xs´ xs´´ -> Cons{1,2} (Cons{2,2} x xs´) xs´´);

splitAt n ys = if (n == 0) (Cons{1,2} Cons{1,0} ys)
                       (lam_splitAt n ys);

takeWhile p ys =
 case ys of
     <1> -> Cons{1,0};
     <2> x xs -> if (p x) (Cons{2,2} x (takeWhile p xs)) Cons{1,0};

takeUntil p ys = case ys of
```

```
                    <1> -> Cons{1,0};
                    <2> x xs -> if (p x) (Cons{2,2} x Cons{1,0})
                                        (Cons{2,2} x (takeUntil p xs));

dropWhile p ys = case ys of
                    <1> -> Cons{1,0};
                    <2> x xs -> if (p x) (dropWhile p xs) ys;

span p ys =
 case ys of
     <1> -> Cons{1,2} Cons{1,0} Cons{1,0};
     <2> x xs ->
        if (p x) (case span p xs of
                     <1> as bs -> Cons{1,2} (Cons{2,2} x as) bs)
                 (Cons{1,2} Cons{1,0} ys);

break p xs = span (compose not p) xs;

lines s =
 case s of
     <1> -> Cons{1,0};
     <2> a b ->
        case break (eq n) s of
          <1> l s' ->
             Cons{2,2} l
               (if (null s') Cons{1,0} (lines (tail s')));

words s = case dropWhile isSpace s of
              <1> -> Cons{1,0};
              <2> t' t'' -> (case break isSpace t' of
                                  <1> w s'' -> Cons{2,2} w (words s''));

unlines xs    = concat (map lam_unlines xs);
lam_unlines l = append l "\n";

unwords ws = case ws of
                <1> -> Cons{1,0};
                <2> a b -> foldr1 lam_unwords ws;

lam_unwords w s = append w (Cons{2,2} ' ' s);

merge xs1 ys1 =
 case xs1 of
     <1> -> ys1;
     <2> x xs -> (case ys1 of
                     <1> -> xs1;
                     <2> y ys -> if (x <= y)
                                    (Cons{2,2} x (merge xs ys1))
                                    (Cons{2,2} y (merge xs1 ys)));

sort xs = foldr insert Cons{1,0} xs;

insert x zs = case zs of
                 <1> -> Cons{2,2} x Cons{1,0};
                 <2> y ys -> if (x <= y)
                                (Cons{2,2} x zs)
```

```
                                (Cons{2,2} y (insert x ys));

gt x y = x > y;
le x y = x <= y;

qsort ys = case ys of
               <1> -> Cons{1,0};
               <2> x xs -> append (qsort (filter (gt x) xs))
                                 (append (Cons{2,2} x Cons{1,0})
                                       (qsort (filter (le x) xs)));

zip as bs = zipWith lam_zip as bs;
lam_zip a b = Cons{1,2} a b;

zip3 as bs cs = zipWith3 lam_zip3 as bs cs;
lam_zip3 a b c = Cons{1,3} a b c;

zip4 as bs cs ds = zipWith4 lam_zip4 as bs cs ds;
lam_zip4 a b c d = Cons{1,4}  a b c d;

zip5 as bs cs ds es = zipWith5 lam_zip5 as bs cs ds es;
lam_zip5 a b c d e = Cons{1,5} a b c d e;

zip6 as bs cs ds es fs = zipWith6 lam_zip6 as bs cs ds es fs;
lam_zip6 a b c d e f = Cons{1,6} a b c d e f;

zip7 as bs cs ds es fs gs = zipWith7 lam_zip7 as bs cs ds es fs gs;
lam_zip7 a b c d e f g = Cons{1,7} a b c d e f g;

zipWith z xs ys =
 case xs of
   <1> -> Cons{1,0};
   <2> a as -> case ys of
                 <1> -> Cons{1,0};
                 <2> b bs -> Cons{2,2}
                               (z a b)
                               (zipWith z as bs);

zipWith3 z xs1 xs2 xs3 =
 case xs1 of
   <1> -> Cons{1,0};
   <2> a as ->
      case xs2 of
        <1> -> Cons{1,0};
        <2> b bs -> case xs3 of
                      <1> -> Cons{1,0};
                      <2> c cs -> Cons{2,2}
                                    (z a b c)
                                    (zipWith3 z as bs cs);

zipWith4 z xs1 xs2 xs3 xs4 =
 case xs1 of
   <1> -> Cons{1,0};
   <2> a as ->
      case xs2 of
        <1> -> Cons{1,0};
```

```
        <2> b bs ->
           case xs3 of
             <1> -> Cons{1,0};
             <2> c cs ->
                case xs4 of
                  <1> -> Cons{1,0};
                  <2> d ds -> Cons{2,2}
                               (z a b c d)
                               (zipWith4 z as bs cs ds);

zipWith5 z xs1 xs2 xs3 xs4 xs5 =
 case xs1 of
   <1> -> Cons{1,0};
   <2> a as ->
      case xs2 of
        <1> -> Cons{1,0};
        <2> b bs ->
           case xs3 of
             <1> -> Cons{1,0};
             <2> c cs ->
                case xs4 of
                  <1> -> Cons{1,0};
                  <2> d ds ->
                     case xs5 of
                       <1> -> Cons{1,0};
                       <2> e es -> Cons{2,2}
                                    (z a b c d e)
                                    (zipWith5 z as bs cs ds es);

zipWith6 z xs1 xs2 xs3 xs4 xs5 xs6 =
 case xs1 of
   <1> -> Cons{1,0};
   <2> a as ->
      case xs2 of
        <1> -> Cons{1,0};
        <2> b bs ->
           case xs3 of
             <1> -> Cons{1,0};
             <2> c cs ->
                case xs4 of
                  <1> -> Cons{1,0};
                  <2> d ds ->
                     case xs5 of
                       <1> -> Cons{1,0};
                       <2> e es ->
                          case xs6 of
                            <1> -> Cons{1,0};
                            <2> f fs -> Cons{2,2}
                                          (z a b c d e f )
                                          (zipWith6 z as bs cs ds es fs);

zipWith7 z xs1 xs2 xs3 xs4 xs5 xs6 xs7 =
 case xs1 of
   <1> -> Cons{1,0};
   <2> a as ->
      case xs2 of
```

```
        <1> -> Cons{1,0};
        <2> b bs ->
           case xs3 of
             <1> -> Cons{1,0};
             <2> c cs ->
                case xs4 of
                  <1> -> Cons{1,0};
                  <2> d ds ->
                     case xs5 of
                       <1> -> Cons{1,0};
                       <2> e es ->
                          case xs6 of
                            <1> -> Cons{1,0};
                            <2> f fs ->
                               case xs7 of
                                 <1> -> Cons{1,0};
                                 <2> g gs ->
                                    Cons{2,2}
                                      (z a b c d e f g)
                                      (zipWith7 z as bs cs ds es fs gs);

unzip ys =
 case ys of
   <1> -> Cons{1,2} Cons{1,0} Cons{1,0};
   <2> x xs ->
      case x of
        <1> x' x'' ->
           case unzip xs of
             <1> y' y'' -> Cons{1,2} (Cons{2,2} x' y')
                                     (Cons{2,2} x'' y'');

-- simulate shows strict behaviour
show x      = case x of
                <1> -> 1;

cjustify n s = letrec m = n - (length s); halfm = m / 2 in
        append (space halfm) (append s (space (m - halfm)));
ljustify n s = append s (space (n - (length s)));
rjustify n s = append (space (n - (length s))) s;

space n      = copy n ' ';

layn xs = lay 1 xs;
lay n ys =
 case ys of
   <1> -> Cons{1,0};
   <2> x xs -> append (rjustify 4 (show n))
                      (append ") "
                         (append x (append "\n" (lay (n + 1) xs))));

main = main;

enumFrom x = Cons{2,2} x (enumFrom (x + 1));

until p f x = if (p x) x (until p f (f x));
```

```
until´ p f x = takeUntil p (iterate f x);

-- simulate error
error xs = show xs;

-- end of standard.prelude.core
-- utils.core

hInitial = Cons{1,3} 0 (enumFrom 1) Cons{1,0};

hAlloc h n =
 case h of
   <1> size I2989 cts ->
      case I2989 of
        <2> next free ->
           Cons{1,2} (Cons{1,3} (size + 1)
                                free
                                (Cons{2,2} (Cons{1,2} next n) cts))
                     next;

remove as a =
 case as of
   <2> I3012 cts ->
      case I3012 of
        <1> a´ n -> if (a == a´)
                       cts
                       (Cons{2,2} (Cons{1,2} a´ n) (remove cts a));

hUpdate h a n =
 case h of
   <1> size free cts ->
      Cons{1,3} size
                free
                (Cons{2,2} (Cons{1,2} a n) (remove cts a));

hFree h a =
 case h of
   <1> size free cts ->
      Cons{1,3} (size - 1)
                (Cons{2,2} a free)
                (remove cts a);

aLookup ass k´ def =
 case ass of
   <1> -> def;
   <2> I3020 bs ->
      case I3020 of
        <1> k v ->
           if (k == k´)
              (aLookup (eq) bs k´ def)
              v;

showaddr a = append (Cons{2,2} ´#´ Cons{1,0}) (show a);

hLookup h a = case h of
  <1> x y cts -> aLookup cts a
```

```
        (error (append "can't find node " (append (showaddr a) " in heap")));

I2975_hAddresses cts =
 case cts of
   <1> -> Cons{1,0};
   <2> I3002 I2976_hAddresses ->
      case I3002 of
        <1> addr v ->
           Cons{2,2} addr (I2975_hAddresses I2976_hAddresses);

hAddresses hp =
 case hp of
   <1> size free cts -> I2975_hAddresses cts;

hSize hp =
 case hp of
   <1> size free cts -> size;

hNull = 0;

hIsnull a = a == 0;

I2978_aDomain alist =
 case alist of
   <1> -> Cons{1,0};
   <2> I3029 I2979_aDomain ->
      case I3029 of
        <1> key val -> Cons{2,2} key (I2978_aDomain I2979_aDomain);

aDomain alist = I2978_aDomain alist;

I2981_aRange alist =
 case alist of
   <1> -> Cons{1,0};
   <2> I3033 I2982_aRange ->
      case I3033 of
        <1> key val -> Cons{2,2} val (I2981_aRange I2982_aRange);

aRange alist = I2981_aRange alist;

aEmpty = Cons{1,0};

I2984_aDelete k as =
 case as of
   <1> -> Cons{1,0};
   <2> x xs ->
      case x of
        <1> k' v ->
           if (k' ~= k)
              (Cons{2,2} (Cons{1,2} k' v) (I2984_aDelete as xs))
              (I2984_aDelete as xs);

aDelete as k = I2984_aDelete k as;

aUpdate as k v = Cons{2,2} (Cons{1,2} k v) (aDelete as k);
```

```
initialNameSupply = 0;

makeName prefix ns = append prefix
        (append (Cons{2,2} '_' Cons{1,0}) (show ns));

getName name_supply prefix =
 Cons{1,2} (name_supply + 1)
   (makeName prefix name_supply);

getNames =
 Cons{1,2} (name_supply + (length prefixes))
   (zipWith makeName prefixes (enumFrom name_supply));

setEmpty = Cons{1,0};

setIsEmpty set =
 case set of
   <1> -> Cons{2,0};
   <2> x xs -> Cons{1,0};

setSingleton x = Cons{2,2} x Cons{1,0};

rmdup xs =
 case xs of
   <1> -> Cons{1,0};
   <2> x I3043 ->
      case I3043 of
        <1> -> Cons{2,2} x Cons{1,0};
        <2> y xs ->
           if (x == y)
              (rmdup (Cons{2,2} y xs))
              (Cons{2,2} x (rmdup (Cons{2,2} y xs)));

setFromList list = rmdup (sort list);

setToList set = set;

setUnion set1 set2 =
 case set1 of
   <1> ->
      (case set2 of
         <1> -> Cons{1,0};
         <2> b bs -> Cons{2,2} b bs);
   <2> a as ->
      case set2 of
        <1> -> Cons{2,2} a as;
        <2> b bs ->
           if (a < b)
              (Cons{2,2} a (setUnion as (Cons{2,2} b bs)))
              (if (a == b)
                  (Cons{2,2} a (setUnion as bs))
                  (if (a > b)
                      (Cons{2,2} b (setUnion (Cons{2,2} a as) bs))
                      Cons{1,0}));

setIntersection set1 set2 =
```

```
 case set1 of
   <1> ->
      (case set2 of
        <1> -> Cons{1,0};
        <2> s ss -> Cons{1,0});
   <2> a as ->
      case set2 of
        <1> -> Cons{1,0};
        <2> b bs ->
           if (a < b)
              (setIntersection as (Cons{2,2} b bs))
              (if (a == b)
                  (Cons{2,2} a (setIntersection as bs))
                  (if (a > b)
                      (setIntersection (Cons{2,2} a as) bs)
                      Cons{1,0}));

setSubtraction set1 set2 =
 case set1 of
   <1> ->
      (case set2 of
        <1> -> Cons{1,0};
        <2> s ss -> Cons{1,0});
   <2> a as ->
      case set2 of
        <1> -> Cons{2,2} a as;
        <2> b bs ->
           if (a < b)
              (Cons{2,2} a (setSubtraction as (Cons{2,2} b bs)))
              (if (a == b)
                  (setSubtraction as bs)
                  (if (a > b)
                      (setSubtraction (Cons{2,2} a as) bs)
                      Cons{1,0}));

setElementOf x xs =
 case xs of
   <1> -> Cons{1,0};
   <2> y ys ->
      if (x == y)
         Cons{2,0}
         if (x > y)
            (setElementOf x ys)
            Cons{1,0};

setUnionList xs = foldll setUnion Cons{1,0} xs;

first a  = case a of <1> s t -> s;
second a = case a of <1> s t -> t;

foldll f b ys =
 case ys of
   <1> -> b;
   <2> x xs -> foldll f (f b x) xs;

mapAccuml f acc ys =
```

```
 case ys of
   <1> -> Cons{1,2} acc Cons{1,0};
   <2> x xs -> case (f acc x) of
      <1> a1 a2 -> case (mapAccuml f a1 xs) of
         <1> b1 b2 -> Cons{1,2} b1 (Cons{2,2} a2 b2);

-- end of Utils
-- lex.core

nonnull p ys =
 case (span p ys) of
   <1> I3038 t ->
      case I3038 of
        <1> -> Cons{1,0};
        <2> x xs -> Cons{2,2} (Cons{1,2} I3038 t) Cons{1,0};

lexDigits xs = nonnull isDigit xs;

asciiTab =
Cons{2,2} "NUL" (Cons{2,2} "SOH" (Cons{2,2} "STX" (Cons{2,2} "ETX"
  (Cons{2,2} "EOT" (Cons{2,2} "ENQ" (Cons{2,2} "ACK" (Cons{2,2} "BEL"
  (Cons{2,2} "BS" (Cons{2,2} "HT" (Cons{2,2} "LF" (Cons{2,2} "VT"
  (Cons{2,2} "FF" (Cons{2,2} "CR" (Cons{2,2} "SO" (Cons{2,2} "SI"
  (Cons{2,2} "DLE" (Cons{2,2} "DC1" (Cons{2,2} "DC2" (Cons{2,2} "DC3"
  (Cons{2,2} "DC4" (Cons{2,2} "NAK" (Cons{2,2} "SYN" (Cons{2,2} "ETB"
  (Cons{2,2} "CAN" (Cons{2,2} "EM" (Cons{2,2} "SUB" (Cons{2,2} "ESC"
  (Cons{2,2} "FS" (Cons{2,2} "GS" (Cons{2,2} "RS" (Cons{2,2} "US"
  (Cons{2,2} "SP" Cons{1,0})))))))))))))))))))))))))))))))));

match as bs =
 case as of
   <1> -> Cons{1,2} as bs;
   <2> x xs ->
      case bs of
        <1> -> Cons{1,2} as bs;
        <2> y ys ->
           if (x == y)
              (match xs ys)
              (Cons{1,2} as bs);

isOctDigit x =
 if ((ord x) >= (ord '0'))
    ((ord x) <= (ord '7'))
    Cons{1,0};

isHexDigit x =
 if (isDigit x)
    Cons{2,0}
    if (if ((ord x) >= (ord 'A'))
           ((ord x) <= (ord 'F'))
           Cons{1,0})
       Cons{2,0}
       if ((ord x) >= (ord 'a'))
          ((ord x) <= (ord 'f'))
          Cons{1,0};
```

```
I2937_lexLitChar cs =
 case cs of
   <1> -> Cons{1,0};
   <2> x xs ->
      case x of
        <1> esc t ->
           Cons{2,2} (Cons{1,2} (Cons{2,2} '\\' esc) t)
             (I2937_lexLitChar xs);

I2934_lexEsc cs ss =
 case ss of
   <1> -> Cons{1,0};
   <2> mne I2935_lexEsc ->
      case (match mne cs) of
        <1> I3051 s' ->
           case I3051 of
             <1> -> Cons{2,2} (Cons{1,2} mne s')
                      (I2934_lexEsc cs I2935_lexEsc);
             <2> y ys -> I2934_lexEsc cs I2935_lexEsc;

I2928_lexEsc ss =
 case ss of
   <1> -> Cons{1,0};
   <2> x xs ->
    case x of
      <1> os t -> Cons{2,2} (Cons{1,2} (Cons{2,2} 'o' os) t)
                            (I2928_lexEsc xs);

I2931_lexEsc ss =
case ss of
  <1> -> Cons{1,0};
  <2> y ys ->
     case y of
      <1> xs t -> Cons{2,2} (Cons{1,2} (Cons{2,2} 'x' xs) t)
                            (I2931_lexEsc ys);

lexEsc cs =
 case cs of
   <1> -> Cons{1,0};
   <2> I3058 s ->
     if (elem I3058 "abfnrtv\\\"'")
        (Cons{2,2} (Cons{1,2} (Cons{2,2} I3058 Cons{1,0}) s) Cons{1,0})
        (case I3058 of
          '^' ->
            case s of
              <1> -> Cons{1,0};
              <2> c s ->
                 if (if ((ord c) >= (ord '@'))
                        ((ord c) <= (ord '_'))
                        Cons{1,0})
                    (Cons{2,2} (Cons{1,2}
                                  (Cons{2,2} '^'
                                     (Cons{2,2} c Cons{1,0}))
                                  s)
                               Cons{1,0})
                    (if (isDigit I3058)
```

```
                        (lexDigits cs)
                        (case I3058 of
                          'o' -> I2928_lexEsc (nonnull isOctDigit s);
                          'x' -> I2931_lexEsc (nonnull isHexDigit s))));

lexLitChar cs =
 case cs of
   <1> -> Cons{1,0};
   <2> c s ->
      case c of
        '\\' -> I2937_lexLitChar (lexEsc s);
        'c' -> Cons{2,2} (Cons{1,2} (Cons{2,2} c Cons{1,0}) s) Cons{1,0};

isSingle c = elem c ",;()[]{}_";

isSym c = elem c "!@#$%&*+./<=>?\\^|:";

isSym1 c =
if (elem c "-~")
   Cons{2,0}
   (isSym c);

I2913_lex ds c ss =
 case ss of
   <1> -> Cons{1,0};
   <2> I3016 I2914_lex ->
      case I3016 of
        <1> fe t -> Cons{2,2} (Cons{1,2} (Cons{2,2} c (append ds fe)) t)
                              (I2913_lex ds c I2914_lex);

I2922_lexExp e ss =
 case ss of
   <1> -> Cons{1,0};
   <2> I3002 I2923_lexExp ->
     case I3002 of
       <1> ds t -> Cons{2,2} (Cons{1,2} (Cons{2,2} e ds) t)
                             (I2922_lexExp e I2923_lexExp);

I2925_lexExp c e s ss =
 case ss of
   <1> -> I2922_lexExp e (lexDigits s);
   <2> I2993 I2926_lexExp ->
      case I2993 of
        <1> ds u -> Cons{2,2} (Cons{1,2} (Cons{2,2} e (Cons{2,2} c ds)) u)
                              (I2925_lexExp c e s I2926_lexExp);

lexExp cs =
 case cs of
   <1> -> Cons{2,2} (Cons{1,2} Cons{1,0} cs) Cons{1,0};
   <2> e s ->
      if (elem e "eE")
         (case s of
           <1> -> I2922_lexExp e (lexDigits s);
           <2> c t ->
              if (elem c "+-")
                 (I2925_lexExp c e s (lexDigits t))
```

```
                (I2922_lexExp e (lexDigits s)))
        (Cons{2,2} (Cons{1,2} Cons{1,0} cs) Cons{1,0});

I2918_lexFracExp I2917_lexFracExp ds I2920_lexFracExp =
 case I2920_lexFracExp of
   <1> -> I2916_lexFracExp I2917_lexFracExp;
   <2> I2979 I2919_lexFracExp ->
      case I2979 of
        <1> e u ->
           Cons{2,2} (Cons{1,2} (Cons{2,2} '.' (append ds e)) u)
                     (I2918_lexFracExp I2917_lexFracExp ds I2919_lexFracExp);

I2916_lexFracExp I2921_lexFracExp =
 case I2921_lexFracExp of
   <1> -> Cons{1,0};
   <2> I2983 I2917_lexFracExp ->
      case I2983 of
        <1> ds t -> I2918_lexFracExp I2917_lexFracExp ds (lexExp t);

lexFracExp s =
 case s of
   <1> -> Cons{2,2} (Cons{1,2} Cons{1,0} s) Cons{1,0};
   <2> I2987 s ->
      case I2987 of
        '.' -> I2916_lexFracExp (lexDigits s);
        'h' -> Cons{2,2} (Cons{1,2} Cons{1,0} s) Cons{1,0};

isIdChar c =
if (isAlphanum c)
   Cons{2,0}
   (if (elem c "_'-[]")
       Cons{2,0}
       (isSym c));

Lam3717_lex xx = (ord xx) == (ord '`');

I2910_lex I2912_lex =
 case I2912_lex of
   <1> -> Cons{1,0};
   <2> I2975 I2911_lex ->
      case I2975 of
        <1> str t -> Cons{2,2} (Cons{1,2} (Cons{2,2} '"' str) t)
                               (I2910_lex I2911_lex);

lexStrItem s1 =
 case s1 of
   <1> -> lexLitChar s1;
   <2> I2971 I2972 ->
      if ((ord I2971) == (ord '\\'))
         (case I2972 of
            <2> c s ->
               (if ((ord c) == (ord '&'))
                   (Cons{2,2} (Cons{1,2} "\\&" s) Cons{1,0})
                   (if (isSpace c)
                       (case (dropWhile isSpace s) of
                          <2> I2969 t ->
```

```
                            (if ((ord I2969) == (ord '\\'))
                                (Cons{2,2} (Cons{1,2} "\\&" t) Cons{1,0})
                                Cons{1,0}))
                        Cons{1,0})))
         (lexLitChar s1);

I2906_lexString I2905_lexString ch I2908_lexString =
 case I2908_lexString of
   <1> -> I2904_lexString I2905_lexString;
   <2> I2958 I2907_lexString ->
      case I2958 of
        <1> str u ->
           Cons{2,2} (Cons{1,2} (append ch str) u)
             (I2906_lexString I2905_lexString ch I2907_lexString);

I2904_lexString I2909_lexString =
 case I2909_lexString of
   <1> -> Cons{1,0};
   <2> I2962 I2905_lexString ->
      case I2962 of
        <1> ch t -> I2906_lexString I2905_lexString ch (lexString t);

lexString ss =
 case ss of
   <1> -> I2904_lexString (lexStrItem ss);
   <2> I2966 s ->
      if ((ord I2966) == (ord '"'))
         (Cons{2,2} (Cons{1,2} (Cons{2,2} '"' Cons{1,0}) s) Cons{1,0})
         (I2904_lexString (lexStrItem ss));

I2901_lex I2903_lex =
 case I2903_lex of
   <1> -> Cons{1,0};
   <2> I2952 I2902_lex ->
      case I2952 of
        <1> ch I2955 ->
           case I2955 of
             <1> -> I2901_lex I2902_lex;
             <2> I2956 t ->
                if ((ord I2956) == (ord '\''))
                   (let
                      V3718 = (Cons{2,2} '\'' Cons{1,0})
                    in
                      (Cons{2,2} (Cons{1,2}
                         (Cons{2,2} '\''
                            (append ch (Cons{2,2} '\'' Cons{1,0})))
                               t)
                   (I2901_lex I2902_lex)));

Lam3716_lex xx = (ord xx) ~=  (ord '\n');

lexNest f ss =
 case ss of
   <1> -> Cons{1,0};
   <2> c s ->
      case c of
```

```
        '-' ->
           (case s of
              <1> -> lexNest f s;
              <2> I2949 s ->
                 if ((ord I2949) == (ord '}'))
                    (f s)
                    (lexNest f s));
        '{' ->
           case s of
             <1> -> lexNest f s;
             <2> I2947 s ->
                if ((ord I2947) == (ord '-'))
                   (lexNest (lexNest f) s)
                   (lexNest f s);

lex ss =
 case ss of
   <1> -> Cons{2,0} ;
   <2> c s -> Cons{1,0};

-- end of lex.core
-- AbsSyntax.core
rLookup hp a = snd (hLookup hp a);
oLookup hp a = fst (hLookup hp a);

rUpdate hp a nd = hUpdate hp a (Cons{1,2} (oLookup hp a) nd);
oUpdate hp a nd = hUpdate hp a (Cons{1,2} nd (rLookup hp a));
bUpdate hp a nd = hUpdate hp a (Cons{1,2} nd nd);
bAlloc hp nd = hAlloc hp (Cons{1,2} nd nd);

allocateSc heap sc =
 case sc of
   <1> name nargs instns ->
      case (bAlloc heap (NGlobal nargs instns)) of
        <1> a b -> Cons{1,2} a (Cons{1,2} name b);

I3270_argOffset n env =
 case env of
   <1> -> Cons{1,0};
   <2> e es ->
      case e of
        <1> v m -> Cons{2,2} (Cons{1,2} v (n + m)) (I3270_argOffset n es);

argOffset n env = I3270_argOffset n env;

LmakeSpine nd = Cons{2,2} nd (Cons{1,0});

makeSpine nd =
case nd of
  <4> e1 e2 -> append (makeSpine e1) (Cons{2,2} e2 (Cons{1,0}));
  <1> a -> LmakeSpine nd;
  <2> a -> LmakeSpine nd;
  <3> t as -> LmakeSpine nd;
  <5> a b c -> LmakeSpine nd;
  <6> a -> LmakeSpine nd;
  <8> u -> LmakeSpine nd;
```

```
cons12 a b = Cons{1,2} a b;

le x y = x <= y;
ge x y = x >= y;

enumFromThen n m = takeWhile (ge m) (enumFrom n);

enumFromThenTo n n1 m = takeWhile (if (n1 >= n) (ge m) (le m))
                                  (enumFromThen n n1);

compileArgs defs env =
 let
   n = length defs
 in append (zipWith cons12 (map first defs)
                           (enumFromThenTo (n - 1) (n - 2) 0))
           (argOffset n env);

I3276_compileAlts comp env alts =
 case alts of
   <1> -> Cons{1,0};
   <2> a as ->
      case a of
        <1> tag names body ->
           Cons{2,2} (Cons{1,2} tag
                        (comp (length names)
                              body
                              (append (zipWith cons12 names (enumFrom 0))
                                      (argOffset (length names) env))))
                     (I3276_compileAlts comp env as);

compileAlts comp alts env = I3276_compileAlts comp env alts;

saturatedCons ps =
 case ps of
   <1> -> error "No match in _saturatedCons";
   <2> e es ->
      case e of
        <3> a b -> let
             V4065 = (length es)
           in b == V4065;
        <1> a -> False;
        <2> a -> False;
        <4> t as -> False;
        <5> a b c -> False;
        <6> a -> False;
        <8> u -> False;

I3273_compileE as =
 case as of
   <1> -> Cons{1,0};
   <2> s ss ->
      case s of
        <1> x env' -> Cons{2,2} (compileE x env') (I3273_compileE ss);

compileLetrec' bind env =
```

```
 case bind of
   <1> -> Cons{1,0};
   <2> first defs ->
      case first of
        <2> name expr ->
           append (compileC expr env)
                  (append (Cons{2,2} (Cons{6,1} (length defs))
                                     (Cons{1,0}))
                          (compileLetrec' defs env));

compileLetrec comp defs expr env =
 let
    env' = compileArgs defs env;
    n = length defs
 in
 append (Cons{2,2} (Cons{8,1} n) Cons{1,0})
        (append (compileLetrec' defs env')
                (append (comp expr env')
                        (Cons{2,2} (Cons{9,1} n) Cons{1,0})));

compileLet' bind env =
 case bind of
   <1> -> Cons{1,0};
   <2> first defs ->
      case first of
        <1> name expr ->
           append (compileC expr env)
                  (compileLet' defs (argOffset 1 env));

compileLet comp defs expr env =
 append (compileLet' defs env)
        (append (comp expr (compileArgs defs env))
                (Cons{2,2} (Cons{9,1} (length defs)) (Cons{1,0})));

I3279_compileC ss =
 case ss of
   <1> -> Cons{1,0};
   <2> I3338 I3280_compileC ->
      case I3338 of
        <1> x args' ->
           Cons{2,2} (compileC x args') (I3279_compileC I3280_compileC);

compileC' offset expr env =
 append (Cons{2,2} (Cons{24,1} offset) (Cons{1,0}))
        (append (compileC expr env)
                (Cons{2,2} (Cons{9,1} offset) (Cons{1,0})));

LcompileCS args e es = append (compileC e args)
                              (compileCS es (argOffset 1 args));

compileCS ps args =
 case ps of
   <2> e es ->
      case e of
        <3> t a ->
           (case es of
```

```
              <1> -> Cons{2,2} (Cons{22,2} t a) (Cons{1,0});
              <2> x xs -> LcompileCS args e es);
         <1> a -> LcompileCS args e es;
         <2> a -> LcompileCS args e es;
         <4> t as -> LcompileCS args e es;
         <5> a b c -> LcompileCS args e es;
         <6> a -> LcompileCS args e es;
         <8> u -> LcompileCS args e es;

compileC nd args =
 case nd of
   <3> t I3289 ->
      (case (I3289 == 0) of
         <2> -> Cons{2,2} (Cons{22,2} t 0) Cons{1,0});
   <1> v ->
      (if (elem v (aDomain args))
          (Cons{2,2} (Cons{4,1} (aLookup args
                                         v
                                         (error (Cons{1,0}))))
                     Cons{1,0})
          (Cons{2,2} (Cons{2,1} v) Cons{1,0}));
    <2> n -> Cons{2,2} (Cons{3,1} n) Cons{1,0};
    <5> recursive defs e ->
       (case recursive of
          <2> -> compileLetrec compileC defs e args;
          <1> -> compileLet compileC defs e args);
    <4> e1 e2 ->
       let
         spine = makeSpine (Cons{4,2} e1 e2)
       in
         (case (saturatedCons spine) of
            <1> -> append (compileC e2 args)
                          (append (compileC e1 (argOffset 1 args))
                                  (Cons{2,2} Cons{5,0} Cons{1,0}));
            <2> -> compileCS (reverse spine) args);
    <6> e as ->
       append (compileE e args)
              (Cons{2,2} (Cons{23,1} (compileAlts compileC' as args))
                         Cons{1,0});
    <8> us ->
       append (concat (I3279_compileC (zip us (iterate (argOffset 1) args))))
              (Cons{2,2} (Cons{30,1} (length us)) Cons{1,0});

LcompileE p env = append (compileC p env (Cons{2,2} Cons{10,0} (Cons{1,0})));

compileE p env =
 case p of
   <2> n -> Cons{2,2} (Cons{3,1} n) Cons{1,0};
   <5> recursive defs e ->
      if recursive (compileLetrec compileE defs e env)
                   (compileLet compileE defs e env);
   <6> e as -> append (compileE e env)
                      (Cons{2,2} (Cons{23,1} (compileAlts compileE' as env))
                                 Cons{1,0});
   <8> us -> append (concat (I3273_compileE
                            (zip us (iterate (argOffset 1) env))))
```

```
                    (Cons{2,2} (Cons{30,1} (length us))
                               (Cons{2,2} Cons{10,0} (Cons{1,0})));
    <1> a -> LcompileE p env;
    <3> a b -> LcompileE p env;
    <4> a b -> LcompileE p env;

compileE' offset expr env =
 append (Cons{2,2} (Cons{24,1} offset) (Cons{1,0}))
        (append (compileE expr env)
                (Cons{2,2} (Cons{9,1} offset) (Cons{1,0})));

compileR e env =
 append (compileC e env)
        (Cons{2,2} (Cons{6,1} (length env))
                   (Cons{2,2} (Cons{7,1} (length env))
                              (Cons{2,2} Cons{1,0} (Cons{1,0}))));

compileSc sc =
 case sc of
   <1> name env body -> Cons{1,3} name
                                  (length env)
                                  (compileR body (zipWith cons12
                                                          env
                                                          (enumFrom 0)));

getArg nd =
 case nd of
   <2> a1 a2 -> a2;

initialCode =
 Cons{2,2} (Cons{2,1} "main")
           (Cons{2,2} Cons{10,0}
                      (Cons{2,2} Cons{25,0} (Cons{1,0})));
```

Appendix B

Results of tests

The following list gives the name of the supercombinator along with the strictness information found. `strict` means the program found this argument to be strict and ? means the analysis did not find this argument to be strict.

```
I [strict]
K [strict, ?]
K1 [?, strict]
S [strict, ?, ?]
compose [strict, ?, ?]
twice [strict, ?]
if [strict, ?, ?]
help []
const [strict, ?]
id [strict]
curry [strict, ?, ?]
uncurry [strict, strict]
fst [strict]
snd [strict]
fst3 [strict]
snd3 [strict]
thd3 [strict]
flip [strict, ?, ?]
binand [strict, ?]
binor [strict, ?]
not [strict]
and [strict]
or [strict]
any [?, strict]
all [?, strict]
otherwise []
isAscii [strict]
isControl [strict]
isPrint [strict]
isSpace [strict]
isUpper [strict]
isLower [strict]
isAlpha [strict]
isDigit [strict]
```

```
isAlphanum [strict]
toUpper [strict]
toLower [strict]
minChar []
maxChar []
even [strict]
odd [strict]
rem [strict, strict]
gcd' [strict, strict]
gcd [strict, strict]
lcm [?, strict]
power [?, strict]
power' [?, strict, ?]
power'' [?, strict, ?]
abs [strict]
signum [strict]
plus [strict, strict]
minus [strict, strict]
mul [strict, strict]
sum [strict]
product [strict]
sums [strict]
products [strict]
head [strict]
last [strict]
tail [strict]
init [strict]
append [strict, ?]
length [strict]
lam_length [strict, ?]
at [strict, strict]
iterate [?, ?]
repeat [?]
cycle [strict]
copy [strict, ?]
nub [strict]
reverse [strict]
eq [strict, strict]
elem [?, strict]
notElem [?, strict]
max [strict, strict]
min [strict, strict]
maximum [strict]
minimum [strict]
concat [strict]
cons22 [?, ?]
transpose [strict]
lam_transpose [strict, ?]
null [strict]
delone [strict, strict]
del' [strict, ?]
map [?, strict]
filter [?, strict]
foldl [?, ?, strict]
foldl1 [?, strict]
foldl' [?, ?, strict]
```

```
scanl [?, ?, strict]
scanl1 [?, strict]
scanl' [?, ?, strict]
foldr [?, ?, strict]
foldr1 [?, strict]
scanr [?, ?, strict]
scanr1 [?, strict]
take [strict, ?]
drop [strict, strict]
lam_splitAt [?, strict]
splitAt [strict, ?]
takeWhile [?, strict]
takeUntil [?, strict]
dropWhile [?, strict]
span [?, strict]
break [?, strict]
lines [strict]
words [strict]
unlines [strict]
lam_unlines [strict]
unwords [strict]
lam_unwords [strict, ?]
merge [strict, strict]
sort [strict]
insert [?, strict]
gt [strict, strict]
le [strict, strict]
qsort [strict]
zip [strict, ?]
lam_zip [?, ?]
zip3 [strict, ?, ?]
lam_zip3 [?, ?, ?]
zip4 [strict, ?, ?, ?]
lam_zip4 [?, ?, ?, ?]
zip5 [strict, ?, ?, ?, ?]
lam_zip5 [?, ?, ?, ?, ?]
zip6 [strict, ?, ?, ?, ?, ?]
lam_zip6 [?, ?, ?, ?, ?, ?]
zip7 [strict, ?, ?, ?, ?, ?, ?]
lam_zip7 [?, ?, ?, ?, ?, ?, ?]
zipWith [?, strict, ?]
zipWith3 [?, strict, ?, ?]
zipWith4 [?, strict, ?, ?, ?]
zipWith5 [?, strict, ?, ?, ?, ?]
zipWith6 [?, strict, ?, ?, ?, ?, ?]
zipWith7 [?, strict, ?, ?, ?, ?, ?, ?]
unzip [strict]
show [strict]
cjustify [strict, strict]
ljustify [?, strict]
rjustify [strict, strict]
space [strict]
layn [strict]
lay [?, strict]
main []
enumFrom [?]
```

```
until [strict, ?, ?]
until´ [strict, ?, ?]
error [strict]
hInitial []
hAlloc [strict, ?]
remove [strict, strict]
hUpdate [strict, ?, ?]
hFree [strict, ?]
aLookup [strict, ?, ?]
showaddr [?]
hLookup [strict, ?]
I2975_hAddresses [strict]
hAddresses [strict]
hSize [strict]
hNull []
hIsnull [strict]
I2978_aDomain [strict]
aDomain [strict]
I2981_aRange [strict]
aRange [strict]
aEmpty []
I2984_aDelete [?, strict]
aDelete [strict, ?]
aUpdate [?, ?, ?]
initialNameSupply []
makeName [strict, ?]
getName [?, ?]
getNames []
setEmpty []
setIsEmpty [strict]
setSingleton [?]
rmdup [strict]
setFromList [strict]
setToList [strict]
setUnion [strict, strict]
setIntersection [strict, strict]
setSubtraction [strict, strict]
setElementOf [?, strict]
setUnionList [strict]
first [strict]
second [strict]
foldll [?, ?, strict]
mapAccuml [?, ?, strict]
nonnull [?, strict]
lexDigits [strict]
asciiTab []
match [strict, ?]
isOctDigit [strict]
isHexDigit [?]
I2937_lexLitChar [strict]
I2934_lexEsc [?, strict]
I2928_lexEsc [strict]
I2931_lexEsc [strict]
lexEsc [strict]
lexLitChar [strict]
isSingle [?]
```

```
isSym [?]
isSym1 [?]
I2913_lex [?, ?, strict]
I2922_lexExp [?, strict]
I2925_lexExp [?, ?, ?, strict]
lexExp [strict]
I2918_lexFracExp [?, ?, strict]
I2916_lexFracExp [strict]
lexFracExp [strict]
isIdChar [strict]
Lam3717_lex [strict]
I2910_lex [strict]
lexStrItem [strict]
I2906_lexString [?, ?, strict]
I2904_lexString [strict]
lexString [strict]
I2901_lex [strict]
Lam3716_lex [strict]
lexNest [?, strict]
lex [strict]
rLookup [strict, ?]
oLookup [strict, ?]
rUpdate [strict, ?, ?]
oUpdate [strict, ?, ?]
bUpdate [strict, ?, ?]
bAlloc [strict, ?]
allocateSc [strict, strict]
I3270_argOffset [?, strict]
argOffset [?, strict]
LmakeSpine [?]
makeSpine [strict]
cons12 [?, ?]
le [strict, strict]
ge [strict, strict]
enumFromThen [strict, strict]
enumFromThenTo [strict, strict, ?]
compileArgs [strict, ?]
I3276_compileAlts [?, ?, strict]
compileAlts [?, strict, ?]
saturatedCons [strict]
I3273_compileE [strict]
compileLetrec' [strict, ?]
compileLetrec [?, ?, ?, ?]
compileLet' [strict, ?]
compileLet [?, strict, ?, ?]
I3279_compileC [strict]
compileC' [?, ?, ?]
LcompileCS [?, strict, ?]
compileCS [strict, ?]
compileC [strict, ?]
LcompileE [?, ?]
compileE [strict, ?]
compileE' [?, ?, ?]
compileR [strict, ?]
compileSc [strict]
getArg [strict]
```

```
initialCode []
+ [strict, strict]
- [strict, strict]
* [strict, strict]
/ [strict, strict]
_+ [strict, strict]
_- [strict, strict]
_* [strict, strict]
_/ [strict, strict]
_negate [?]
negate [strict]
== [strict, strict]
~= [strict, strict]
< [strict, strict]
<= [strict, strict]
> [strict, strict]
>= [strict, strict]
_== [strict, strict]
_~= [strict, strict]
_< [strict, strict]
_<= [strict, strict]
_> [strict, strict]
_>= [strict, strict]
_ord [?]
_chr [?]
ord [strict]
chr [strict]
strict [strict, strict]

real       11.3
user       10.7
sys         0.5
```

Bibliography

[Aug84] Lennart Augustsson. A compiler for Lazy ML. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 218–227, 1984.

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

[Bur91] Geoffrey Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation.* Pitman, London, 1991.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice-HallInternational, London, 1988.

[GH93] E. Goubault and C. L. Hankin. A lattice for the abstract interpretation of term graph rewriting systems. In M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors, *Term Graph Rewriting - Theory and Practice*, chapter 10. Wiley, Chichester, 1993.

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

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

[HPW+92] Paul Hudak [ed.], Simon L. Peyton Jones [ed.], Philip Wadler [ed.], Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell. A non-strict purely functional language. version 1.2, 1992.

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

[Joh84] T. Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the ACM Conference on Compiler Construction, Montreal*, pages 58–69, 1984.

[Les88] David R. Lester. Combinator graph reduction: A congruence and its applications. Technical report, Oxford University, 1988.

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

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

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

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

[Par95] Will Partain. Private correspondence, 1995.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International, London, 1987.

[PJHH+92] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference*, 1992.

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

[PJP] Simon L. Peyton Jones and Will Partain. Measuring the effectiveness of a simple strictness analyser. Draft.

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

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