Strictness Analysis on Non-Flat Domains
(by Abstract Interpretation over Finite Domains)

Philip Wadler
Programming Research Group
Oxford University*

Recent work shows that lazy functional languages can be compiled to run on conventional architectures with very good speed [Johnsson 84, Peyton-Jones 86]. Strictness analysis is one way to make such implementations even faster. This is because when an argument to a function is known to be strict then one may evaluate the argument directly rather than build a data structure to be evaluated later.

One of the most promising techniques of strictness analysis is by abstract interpretation [Mycroft 83], which recently has been extended to include higher-order functions [Burn et al 85, Hudak and Young 85, 86] and polymorphism [Abramsky 85]. One of the remaining problems of great interest is whether this method can be extended to non-flat domains.

Accordingly, strictness analysis on non-flat domains has received a great deal of attention [Hughes 85, Karlsson 85, Kieburtz 85]. Unfortunately, the work to date on non-flat domains has used abstract domains that are infinite, whereas the use of abstract domains that are finite has been one of the most attractive features of the other work. In particular, the use of infinite domains makes it difficult to solve the fixpoint equations describing the abstract behaviour.

This Chapter presents an approach to strictness analysis on non-flat domains using abstract interpretation over finite domains. This means that the standard methods for finding least fixpoints can be used, and that higher-order functions and polymorphism can be handled by straightforward application of the recent results.

The Chapter is organized as follows. The first section presents the basic idea. For simplicity, it deals with the specific case of lists of integers. The

*In S. Abramsky and C. Hankin, *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.

second section extends this to lists of any type (including, e.g., lists of lists of integers). The third section presents discussion and conclusions.

Data structures more general than lists (such as trees) are not dealt with in this paper. However, it is clear that the method could be extended to handle them as well. There does not appear to be any serious problem in extending the method to handle general free data types, as found for instance in Miranda [Turner 85].

It is assumed that the reader is familiar with strictness analysis on flat domains. See [Clack and Peyton-Jones 85] for an introduction to this subject.

1 The basic method

This section illustrates the basic method. For simplicity, it assumes that one is interested in only two concrete domains, integers and lists of integers. This restriction is lifted in the next section.

1.1 The domain

A concrete flat domain, such as the integers, is represented by an abstract domain containing two elements, $\top$ and $\perp$. There is a function mapping each concrete element x into an abstract element $x^{\#}$. This function must satisfy the property that $x \sqsubseteq y$ implies $x^{\#} \sqsubseteq y^{\#}$. In this case, every non-$\perp$ element of the flat domain maps onto $\top$, and $\perp$ map onto $\perp$. This may be diagrammed as follows:

$$\begin{array}{ccl} \top & — & \text{any value, except } \perp \\ | & & \\ \perp & — & \perp \end{array}$$

For example, $1^{\#} = \top$ and $\perp^{\#} = \perp$. Here $\perp$ is written for an element of both the concrete domain and the abstract domain; which is meant should be clear by context.

The concrete non-flat domain of lists of integers is represented by the following four element abstract domain:

$$\begin{array}{ccl} \top\in & — & \text{any finite list, no member of which is } \bot \\ | & & \\ \bot\in & — & \text{any finite list, some member of which is } \bot \\ | & & \\ \infty & — & \text{any infinite list or approximation to one, except } \bot \\ | & & \\ \bot & — & \bot \end{array}$$

For example,

$$\begin{array}{lcl} (cons\ 1\ (cons\ 2\ nil))^{\#} & = & \top\in \\ (cons\ 1\ (cons \bot nil))^{\#} & = & \bot\in \\ (cons\ 1 \bot)^{\#} & = & \infty \\ \bot^{\#} & = & \bot \end{array}$$

Here $\bot$ is written for an element of both the domain of integers and the domain of lists of integers (both concrete and abstract). Again, which is meant should be clear from context.

The approximations to the infinite lists are all lists ending in bottom. The set of all such lists will include the infinite lists, by continuity. Similarly, the finite lists are those lists ending in nil. (More precisely: a list xs is finite if $tail^i xs = nil$ for some i.) Any approximation to an infinite list is also an approximation to some finite list, but no finite list approximates an infinite list. This explains why the ordering on the domain must be as given above, with ∞ below $\bot\in$.

1.2 Cons

It is clear from the above that $\bot^{\#} = \bot$ and $nil^{\#} = \top$. We also need to define a function $cons^{\#}$ such that

$$(cons\ x\ xs)^{\#} = cons^{\#} x^{\#} xs^{\#}$$

for any integer x and list of integers xs. This function is defined by the following table:

$cons^{\#} x^{\#} xs^{\#}$

$xs^{\#} \backslash x^{\#}$	$\top$	$\bot$
$\top\in$	$\top\in$	$\bot\in$
$\bot\in$	$\bot\in$	$\bot\in$
∞	∞	∞
$\bot$	∞	∞

It can be seen that $cons^{\#}$ is monotonic and has the desired property.

For example, from this table it follows that

$$cons^{\#} \top \bot\!\in\; = \bot\!\in.$$

That is, if one adds a non-bottom element to a finite list containing some bottom element, then the result is also a finite list containing some bottom element.

1.3 Case analysis

It is convenient to write function definitions in the following form:

$$\begin{array}{lcl} h\ nil & = & a \\ h\ (cons\ x\ xs) & = & f\ x\ xs \end{array}$$

For example, the function *sum* is defined by:

$$\begin{array}{lcl} sum\ nil & = & 0 \\ sum\ (cons\ x\ xs) & = & x + sum\ xs \end{array}$$

In this case, h is sum, a is 0, and f is the function such that $f\ x\ xs = x + sum\ xs$.

Given a definition for h in the above form, we need to derive a definition for $h^{\#}$, such that $(h\ xs)^{\#} \sqsubseteq (h^{\#} xs^{\#})$. We can do this by considering each of the four possible values for $xs^{\#}$, and using the table defining $cons^{\#}$ in a backwards manner. For example, if $us = cons\, x\, xs$ and $us^{\#} = \bot\!\in$, it follows from the definition of $cons^{\#}$ that

$$\begin{array}{lccc} & x^{\#} = \bot & \text{and} & xs^{\#} = \top\!\in \\ \text{or} & x^{\#} = \top & \text{and} & xs^{\#} = \bot\!\in \\ \text{or} & x^{\#} = \bot & \text{and} & xs^{\#} = \bot\!\in \end{array}$$

Hence,

$$\begin{array}{lcl} h^{\#} \bot\!\in & = & (f^{\#} \bot \top\!\in) \sqcup (f^{\#} \top \bot\!\in) \sqcup (f^{\#} \bot \bot\!\in) \\ & = & (f^{\#} \bot \top\!\in) \sqcup (f^{\#} \top \bot\!\in) \end{array}$$

The term $(f^{\#} \bot \bot\!\in)$ can be eliminated because $f^{\#}$ must be monotonic, so $(f^{\#} \bot \bot\!\in) \sqsubseteq (f^{\#} \top \bot\!\in)$.

Continuing this reasoning gives the complete definition of $h^{\#}$:

$$\begin{array}{lcl} h^{\#} \top\!\in & = & a^{\#} \sqcup (f^{\#} \top \top\!\in) \\ h^{\#} \bot\!\in & = & (f^{\#} \bot \top\!\in) \sqcup (f^{\#} \top \bot\!\in) \\ h^{\#} \infty & = & (f^{\#} \top \infty) \\ h^{\#} \bot & = & \bot \end{array}$$

The term $a^{\#}$ appears only in the case for $\top\!\in$, because if $us = nil$ it follows that $us^{\#} = \top\!\in$. (Note that nil cannot correspond to $\bot\!\in$, because $\bot\!\in$ corresponds to finite lists containing at least one bottom, and the empty list, of course, contains no bottom.)

1.4 Abstracting a function

We are now in a position to find the abstract function corresponding to a concrete function on lists. For example, consider the function sum, defined as follows:

$$\begin{array}{lcl} sum\ nil & = & 0 \\ sum\ (cons\ x\ xs) & = & x + sum\ xs \end{array}$$

Using the method described above, together with the rules that $0^{\#} = \top$ and $(x+y)^{\#} = x^{\#} \sqcap y^{\#}$, we have:

$$\begin{array}{lcl} sum^{\#} \top\!\in & = & \top \sqcup (\top \sqcap (sum^{\#} \top)) \\ & = & \top \\ sum^{\#} \bot\!\in & = & (\bot \sqcap (sum^{\#} \top\!\in)) \sqcup (\top \sqcap (sum^{\#} \bot\!\in)) \\ & = & sum^{\#} \bot\!\in \\ sum^{\#} \infty & = & (\top \sqcap (sum^{\#} \infty)) \\ & = & sum^{\#} \infty \\ sum^{\#} \bot & = & \bot \end{array}$$

This function has the least fix-point given in the table below. (This least fix-point can be computed using Kleene chains in the usual way; see [Clack and Peyton-Jones 85].)

$xs^{\#}$	$sum^{\#} xs^{\#}$
$\top\!\in$	$\top$
$\bot\!\in$	$\bot$
∞	$\bot$
$\bot$	$\bot$

This result is exactly what we would expect. The sum of a list is defined only if the list is finite and all of its elements are not bottom.

On the other hand, if we apply the method to the function length (defined in the usual way; see table 1 in section 1.5) then we get the following abstract function:

$xs^{\#}$	$length^{\#} xs^{\#}$
$\top\in$	$\top$
$\bot\in$	$\top$
∞	$\bot$
$\bot$	$\bot$

Again, this is the expected result. The length of a list is defined only if the list is finite, but some of its elements may be bottom.

Strictness results tell us to what extent the argument to the function can be safely evaluated. As is well known, if $f^{\#}\bot = \bot$, then it is safe to evaluate the argument to f. This is because if evaluation of x does not terminate, then evaluation of $(f\,x)$ would not terminate anyway. Similarly, if $f^{\#}\infty = \bot$, then it is safe to evaluate the argument to f, which must be a list, and to evaluate the tail of this argument, and the tail of the tail, and so on. In this case, we say that f is strict in the tail. Further, if $f^{\#}\bot\in = \bot$, then it is safe to evaluate the argument to f, and all its tails, and all its heads as well. In this case, we say that f is strict in the head and the tail. For example, the results above show that length is strict in the tail, and *sum* is strict in the head and the tail.

1.5 Further examples

Table 1 (on the next page) gives the definition of some standard list functions, and Table 2 gives the corresponding abstract functions, calculated using the method described above. (Although *append* comes later in the table, in practice $append^{\#}$ must be calculated before $rev^{\#}$, since *rev* depends on *append*.)

The reader should verify that these results are in accord with what one would intuitively expect. For example, the result for $last^{\#}$ says that *last* requires a finite list, but that some of the elements of this list may be bottom. This approximation is the best one could expect, since there are indeed lists such that $xs^{\#} = \bot\in$ but $(last\ xs)^{\#} \neq \bot$ (for example, $cons \bot (cons\ 1\ nil)$). This result is particularly noteworthy since [Hughes 85], beginning with the same definition, derives a rather worse approximation.

2 Lists of any type

This section extends the method of the previous section to deal with lists of any given type. It is assumed that every term in the language has a

Table 1: Some standard list functions.

$$
\begin{array}{lcl}
head\ nil & = & \bot \\
head\ (cons\ x\ xs) & = & x \\
\\
tail\ nil & = & \bot \\
tail\ (cons\ x\ xs) & = & xs \\
\\
sum\ nil & = & 0 \\
sum\ (cons\ x\ xs) & = & x + sum\ xs \\
\\
length\ nil & = & 0 \\
length\ (cons\ x\ xs) & = & 1 + length\ xs \\
\\
rev\ nil & = & nil \\
rev\ (cons\ x\ xs) & = & append\ (rev\ xs)\ (cons\ x\ nil) \\
\\
last\ xs & = & head\ (rev\ xs) \\
\\
append\ nil\ ys & = & ys \\
append\ (cons\ x\ xs)\ ys & = & cons\ x\ (append\ xs\ ys)
\end{array}
$$

Table 2: Abstract functions corresponding to functions in Table 1.

	$head^{\#}$	$tail^{\#}$	$sum^{\#}$	$length^{\#}$	$rev^{\#}$	$last^{\#}$
$\top\in$	$\top$	$\top\in$	$\top$	$\top$	$\top\in$	$\top$
$\bot\in$	$\top$	$\top\in$	$\bot$	$\top$	$\bot\in$	$\top$
∞	$\top$	∞	$\bot$	$\bot$	$\bot$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

$append^{\#}\, xs^{\#}\, ys^{\#}$

$xs^{\#} \backslash ys^{\#}$	$\top\in$	$\bot\in$	∞	$\bot$
$\top\in$	$\top\in$	$\bot\in$	∞	∞
$\bot\in$	$\bot\in$	$\bot\in$	∞	∞
∞	∞	∞	∞	∞
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

monomorphic type, e.g., Int or Int^* or Int^{**} or $(Int \to Int)^*$, where D^* is the type "list of D". Polymorphic types, such as α^* where α is a type variable, can be handled by using the method of [Abramsky 85].

2.1 The domain

Let D be a given concrete domain, and let D^* be the domain of lists of elements of D. Further, let $D^\#$ be the abstract domain corresponding to D. For example, D might be the domain of integers and $D^\#$ the domain $\top, \perp$.

The abstract domain $D^{*\#}$ corresponding to D^* is constructed as follows. First, we create a new domain $D\text{-}\in^\#$ by adding $\in$ to the end of the name of every element in $D^\#$. Thus, if $D^\#$ is $\{\top, \perp\}$ then $D\text{-}\in^\#$ is $\{\top\in, \perp\in\}$. Second, we add two new elements ∞ and $\perp$ to $D\text{-}\in^\#$ to get $D^{*\#}$. These elements are ordered so that

$$\perp \sqsubseteq \infty \sqsubseteq \perp\in \sqsubseteq \cdots \sqsubseteq \top\in$$

where $\perp\in$ and $\top\in$ are the bottom and top elements of $D\text{-}\in^\#$. The other elements (if any) of $D\text{-}\in^\#$ have the same order as in $D^\#$.

The interpretation of $D^{*\#}$ is as follows. As before, $\perp$ corresponds to bottom, and ∞ corresponds to any infinite list. Let xs be a finite list $[x_1, \ldots, x_n]$ in D^*. Then $xs^\#$ is $(x^\#)\text{-}\in$, where

$$x^\# = {x_1}^\# \sqcap \cdots \sqcap {x_n}^\#.$$

That is, $x^\#$ is the minimum of the abstractions of the elements of the list.

For example, if D is the domain of integers, then $D^{*\#}$ is as before, and $D^{**\#}$ is:

$$\begin{array}{c} \top\in\in \\ | \\ \perp\in\in \\ | \\ \infty\in \\ | \\ \perp\in \\ | \\ \infty \\ | \\ \perp \end{array}$$

For instance, $\bot\in\in$ corresponds to a finite list, the minimum element of which is in $\bot\in$. That is, $\bot\in\in$ corresponds to a finite list, every element of which is a finite list, but some element of which is a finite list containing bottom.

2.2 Cons

The definition of $cons^{\#}$ follows immediately from the fact that each finite list is represented by its minimum abstract element. The definition is given by the following rules:

$$\begin{array}{lcl} cons^{\#} x^{\#} (y^{\#}\text{-}\in) & = & (x^{\#} \sqcap y^{\#})\text{-}\in \\ cons^{\#} x^{\#} \infty & = & \infty \\ cons^{\#} x^{\#} \bot & = & \infty \end{array}$$

For example, if D is the domain of integers, then $cons^{\#}$ for D^* is as defined in section 1.3.

As mentioned before, we assume that each term in the language has a monomorphic type. This means that we can label each instance of *cons* to distinguish the type of elements involved. In general, $cons[D]$ will have the type $D \to D^* \to D^*$. For example, the term

$$cons\ (cons\ x\ xs)\ xss$$

where x is in Int, xs is in Int^*, and xss is in Int^{**}, could be written more fully as

$$cons[Int^*]\ (cons[Int]\ x\ xs)\ xss.$$

The definition of $cons[Int]$ is as given in section 1.3, and the definition of $cons[Int^*]$ is given by the table below.

$cons[Int^*]\ xs^{\#} xss^{\#}$

$xss^{\#} \backslash xs^{\#}$	$\top\in$	$\bot\in$	∞	$\bot$
$\top\in\in$	$\top\in\in$	$\bot\in\in$	$\infty\in$	$\bot\in$
$\bot\in\in$	$\bot\in\in$	$\bot\in\in$	$\infty\in$	$\bot\in$
$\infty\in$	$\infty\in$	$\infty\in$	$\infty\in$	$\bot\in$
$\bot\in$	$\bot\in$	$\bot\in$	$\bot\in$	$\bot\in$
∞	∞	∞	∞	∞
$\bot$	∞	∞	∞	∞

2.3 An example

The remainder of the analysis method works exactly as before. For example, let *revall* be a function that reverses a list of lists, and also reverses each list in it. It can be defined as follows:

$$\begin{array}{lcl} \mathit{revall}\ \mathit{xss} & = & \mathit{rev}\ (\mathit{map}\ \mathit{rev}\ \mathit{xss}) \\[1ex] \mathit{map}\ f\ \mathit{nil} & = & \mathit{nil} \\ \mathit{map}\ f\ (\mathit{cons}\ x\ \mathit{xs}) & = & \mathit{cons}\ (f\ x)\ (\mathit{map}\ f\ \mathit{xs}) \end{array}$$

where *rev* is as in Table 1. Then using the analysis method above yields the following table.

$\mathit{xss}^{\#}$	$\mathit{revall}^{\#}\mathit{xss}^{\#}$
$\top\in\in$	$\top\in\in$
$\bot\in\in$	$\bot\in\in$
$\infty\in$	$\bot\in$
$\bot\in$	$\bot\in$
∞	$\bot$
$\bot$	$\bot$

The function *map* will have to be analyzed differently when it is applied to a list of integers than when it is applied to a list of lists of integers. It is possible to distinguish the different calls because each term is monomorphically typed.

3 Discussion

The method of strictness analysis described here can determine the safety of three optimizations:

(1) evaluate all heads and tails of the argument list (safe if $f^{\#}\bot\in\ = \bot$).

(2) evaluate all tails of the argument list (safe if $f^{\#}\infty = \bot$).

(3) evaluate the argument list (safe if $f^{\#}\bot = \bot$).

Case (1) can be interpreted as saying that it is safe to build the argument list with a version of cons that is strict in the head and tail, and case (2) as saying that it is safe to build the argument list with a version of cons that is strict in the tail.

It is disappointing that the analysis cannot determine the case where it is safe to use a version of cons that is strict in the head, but not the tail.

However, there is an intuitive reason why this should be the case. Namely, being strict in the head corresponds to the case where it is safe to evaluate all the heads—but one cannot evaluate all the heads in a list unless one also evaluates all the tails. (I am grateful to Dick Kieburtz for pointing this out.)

One use of strictness analysis is to improve (by a constant factor) the time complexity of a program. On the other hand, Meira points out that using the results of strictness analysis to make a list strict in the tail may result in a program with worse space complexity [Meira 84]. For example, consider the term

$$\mathit{sum}\ (\mathit{upto}\ 1\ n)$$

where $(\mathit{upto}\,1\,n)$ returns the list of numbers from 1 to n. As we have seen, it is safe to build the argument to sum using a version of cons that is strict in both the head and the tail. However, doing so changes the space complexity of the program from constant to linear in n.

A general rule that avoids this problem is to only consider making a component strict if its type is such that the space to store an evaluated element is bounded by a constant. This guarantees that the time complexity increases by at most a constant. Integers occupy bounded space but lists do not, so in the case above we would choose to build the list using a version of cons that is strict only in the head. It seems likely that this simple heuristic would work well in practice.

A central feature of the method discussed here is that it treats all elements of a list in the same way. This is in contrast with other methods (e.g., [Hughes 85]) which may yield a result that says, for instance, that some list will have at least two elements, and it is safe to evaluate the tail of the first and the head of the second. The method described here achieves its simplicity by giving up the possibility of finding this sort of information. This does not seem too great a price to pay, particularly since the list data type is most appropriate when one intends to treat all elements of the list in the same way.

Some people might suspect that this method throws away too much information, since important functions such as *head* and *tail* do treat the first element of the list differently from the others. However, there is evidence that the method provides reasonably good analysis even of functions involving *head* and *tail*. For example, the function

$$\mathit{last}\ \mathit{xs} = \mathit{head}\ (\mathit{rev}\ \mathit{xs})$$

was analysed in section 1.5, with the expected result: xs can be made strict in the tail, but not in the head. Hughes' method, despite what one might expect, does not find so good a result, because it has difficulty finding a fixpoint solution to the equations in the infinite abstract domain. Hughes has also suggested a modified method which would solve this particular problem, but at the cost of further complexity to his algorithm. (I am grateful to John Hughes for pointing out this example.)

The case analysis as described in section 1.3 is also central to this method. If a function is defined without case analysis, the results of the method may not be very good. For example, if the function sum is defined by

$$sum\ xs = \mathbf{if}\ (null\ xs)\ \mathbf{then}\ 0\ \mathbf{else}\ ((head\ xs) + sum\ (tail\ xs))$$

then the result of strictness analysis is:

$xs^{\#}$	$sum^{\#}\,xs^{\#}$
$\top\in$	$\top$
$\bot\in$	$\top$
∞	$\top$
$\bot$	$\bot$

which is virtually useless. Case analysis is essential because it maintains information that would otherwise be lost, such as that if $(head \bot\in)$ is $\top$ then $(tail \bot\in)$ must be $\bot\in$; and that if $(null\ xs)$ is true then xs must be $\top$.

To summarize: A method of strictness analysis using abstract interpretation on finite domains has been presented. The method applies to non-flat list domains, and it seems straightforward to extend the method to apply to any free data type. The method can be applied to higher-order functions and programs with polymorphic types using existing results.

A great deal has been learned about strictness analysis in recent times. One can look forward to a new generation of compilers for functional languages that make use of these results.

Acknowledgements. This work owes its existence to the Workshop on Programs as Data Objects held in Copenhagen, on 16–19 October 1985, organized by Neil Jones and Harald Ganzinger. The idea germinated during a discussion with John Hughes on the flight to Copenhagen; was nurtured by the presentations of Geoffrey Burn, Samson Abramsky, and John Hughes; and blossomed into full flower at 2 am as I lay in the hotel room trying to sleep. The work was shown the next day to Samson Abramsky and Geoffrey Burn, who made many useful comments, and it was unofficially presented as the $(n+1)$'th talk, over Tuborg Green, to an audience consisting of John

Hughes, Thomas Johnsson, Neil Jones, Dick Kieburtz, and Mitch Wand. I thank all these people for their comments and companionship.

This work was performed while on a research fellowship funded by ICL.

References

[Abramsky 85] Samson Abramsky. Strictness analysis and polymorphic invariance. In [Ganzinger and Jones 85].

[Burn et al 85] Geoffrey Burn, Chris Hankin, and Samson Abramsky. The theory of strictness analysis for higher order functions. In [Ganzinger and Jones 85].

[Clack and Peyton-Jones 85] Chris Clack and Simon Peyton-Jones. Strictness analysis—a practical approach. In [Jouannaud 85].

[Ganzinger and Jones 85] Harald Ganzinger and Neil Jones, editors. *Proceedings of the Workshop on Programs as Data Objects.* Copenhagen, October 1985. Lecture Notes in Computer Science 217, Springer-Verlag, 1986.

[Hudak and Young 85] Paul Hudak and J. Young. A set-theoretic characterization of function strictness in the lambda calculus. Technical Report YALEU/DCS/RR-391, January, 1985.

[Hudak and Young 86] Paul Hudak and J. Young. Higher-order strictness analysis in untyped lambda calculus. *Proceedings of ACM Symposium on Principles of Programming Languages.* January, 1986.

[Hughes 85] John Hughes. Strictness detection in non-flat domains. In [Ganzinger and Jones 85].

[Johnsson 84] Thomas Johnsson. Efficient compilation of lazy evaluation. *Proceedings of the ACM Sigplan Symposium on Compiler Construction*, Montreal, 1984. *SIGPLAN Notices* **19**(6), June 1984.

[Jouannaud 85] Jean-Pierre Jouannaud, editor. *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985. Lecture Notes in Computer Science 201, Springer-Verlag, 1985.

[Kieburtz 85] Dick Kieburtz. Strictness detection in non-flat domains. Talk delivered at Oxford University, October 1985.

[Karlsson 85] Kent Karlsson. Strictness analysis. Talk deliverd at Workshop on Abstract Interpretation, University of Kent, August 1985.

[Meira 84] Silvio Meira. Optimised combinatoric code for applicative language implementation. *Proceedings of the 6'th International Symposium on Programming.* Springer-Verlag, 1984.

[Mycroft 83] Alan Mycroft. Abstract interpretation and optimising transformations for applicative programs. Doctoral dissertation. Department of Computer Science, University of Edinburgh, 1983.

[Peyton-Jones 86] Simon Peyton-Jones. *Implementing Functional Languages using Graph Reduction.* Prentice-Hall, 1987.

[Turner 85] David Turner. Miranda: A non-strict functional language with polymorphic types. In [Jouannaud 85].