Strictness analysis - a practical approach

Chris Clack and Simon L Peyton Jones
Department of Computer Science, University College London
Gower Street, London WC1E 6BT, United Kingdom

Abstract

Significant improvements in performance arise if we can arrange for parallel execution of programs. The absence of side effects in functional languages allows concurrent evaluation of the program, but in lazy implementations this risks wasting work by evaluating expressions which are subsequently discarded. We discuss the use of strictness analysis to determine at compile time which parts of program evaluation can safely be carried out concurrently. We give a practical explanation of this technique, concentrating particularly on the problem of finding fixed points.

1. Introduction

Functional programming languages are naturally extensible to parallel evaluation, since they are free from side effects. For example, the expression

$$(6 + 7) * (4 + 5)$$

can be evaluated by first evaluating the two subexpressions (6 + 7) and (4 + 5) **simultaneously** and then evaluating the resulting expression (13 * 9).

This suggests that parallel evaluation should start when we apply a function to an argument. However, a major feature of some implementations of functional languages is **lazy evaluation,** which delays evaluation of arguments until they are needed. Thus, a function may not evaluate all of its arguments - an argument may never be needed - so we cannot assume that they may be evaluated in parallel. The parallelism available to lazy implementations of functional languages therefore seems to be limited to simultaneous evaluation of the arguments to primitive functions such as "+" and "*".

More parallelism could be achieved if the programmer were to annotate non-primitive functions to indicate which arguments may safely be evaluated in parallel. A better method however is to analyse the program at compile-time, in the manner of an optimising compiler. This static analysis of the program text is called **strictness analysis.**

Strictness analysis may also allow more efficient code to be generated for a sequential machine, since an argument may be evaluated immediately rather than building and retaining a closure.

In this paper we explain informally the theoretical background for strictness analysis, and describe some practical algorithms. We concentrate particularly on the problems of finding fixed points, pointing out some of the pitfalls for the unwary.

2. Background

Strictness analysis is one of the many compile-time optimisations that can be achieved through **abstract interpretation** of the program text.

The pioneers in the field of abstract interpretation were Cousot and CousotCouso77a. Since then the theory has been extended by MycroftMycro83a, whose doctoral thesis explained how the Cousots' theory could be applied to functional languages. In particular, he presented a formal explanation of strictness analysis, albeit limited to first order functions in flat domains (a flat domain does not include list structures). He described an implementation of his theoretical work, which produced promising results.

JohnssonJohns81a gives a method based on recursive set equations, which was developed independently of, and is less powerful than, Mycroft's work. An example of the use of strictness analysis in an optimising compiler is to be found inHudak84a. More recent work by Hudak and Young on higher order strictness analysis appears inHudak85a.

Mishra et alMishr84a were not concerned with strictness analysis, but extended Mycroft's work to cover other compile time optimisations such as the identification of relevant clauses. This work includes an extension to non-flat domains.

MeiraMeira84a has given an extension of abstract interpretation based on combinatoric code. His algorithms cope naturally with higher order functions and he claims that they also cover non-flat domains, although he does not show the algorithmic details. WrayWray85a describes a strictness analysis algorithm which, unusually, seems not be be based on abstract interpretation.

An important step forward is taken in a recent paper by Burn, Hankin and Abramsky, which extends the theory of abstract interpretation to include higher order functionsBurn86a.

3. Strictness analysis through abstract interpretation

The application of abstract interpretation to strictness analysis was first suggested by Mycroft, but his presentation is primarily theoretical, so we give a practical exposition of the approach here. In doing so, we try to give an intuitive grasp of the technique, and inevitably we gloss over several important theoretical issues. Fortunately, the intuitive approach leads us to a correct implementation. We assume an understanding of basic domain theory, including fixed points.

3.1 What does it mean for a function to be strict?

An intuitive definition of strictness is that **a function is strict if and only if it always needs the value of its argument.** We can formalise this by saying that a function f is strict iff

(†) $$f \perp = \perp$$

(function application is denoted by juxtaposition). That is, given a non-terminating argument f will not terminate. Of course, f could be failing to terminate for reasons other than trying to evaluate its argument, but the net result is the same. Certainly if (†) holds then it is safe to evaluate the argument in parallel with the call of f.

This notion extends naturally to functions of several arguments. For instance, if f is a function of 3 arguments (x, y and z) we say that it is strict in y iff

$$f\ x \perp z = \perp \quad \textbf{for any} \quad x \text{ and } z$$

3.2 An archetypal example - the rule of signs

To extract information about strictness from the program text we use a technique called **abstract interpretation.** A classic example of abstract interpretation is given by the "rule of signs":

$$\begin{array}{llll} (+) \text{ times\# } (+) & = (+) \qquad & (-) \text{ times\# } (+) & = (-) \\ (+) \text{ times\# } (-) & = (-) \qquad & (-) \text{ times\# } (-) & = (+) \end{array}$$

(For the moment, ignore the #'s.) To formalise what is going on, we are abstracting the domain of Integers onto a two-point **abstract domain,** {(+), (-)}, thus

$$\text{Integers} \xrightarrow[\text{ABS}]{} \{(+), (-)\}$$

The "times" function is a function from pairs of Integers to Integers, thus:

$$\begin{array}{c} \text{Integers} \\ \Big\downarrow{\scriptstyle \text{times}} \\ \text{Integers} \end{array}$$

The "times#" function models the "times" function, **but in the abstract domain,** thus:

$$\begin{tikzcd} \text{Integers} \arrow[r, "\text{ABS}"'] \arrow[d, "\text{times}"] & \{(+), (-)\} \arrow[d, "\text{times\#}"] \\ \text{Integers} \arrow[r, "\text{ABS}"'] & \{(+), (-)\} \end{tikzcd}$$

From this diagram we can see that what we require from "times#" is that

(□) ABS (times a b) = times# (ABS a) (ABS b)

(Note: in general the "=" becomes "≤". This is one of the main theoretical issues we gloss over.) So by applying the abstract functions in the abstract domain, we may be able to extract useful information about the behaviour of the original functions in the original domain. This process is called abstract interpretation.

Since the domains in which we work include the function space, we see that the following should hold:

times# = ABS times

Of course, this means that the target domain is not just {(+), (-)}, but also functions over this space, and so on. Burn, Hankin and AbramskyBurn86a have recently shown that we can indeed extend ABS to the function space in a perfectly straightforward manner. In what follows, however, we will use f# to denote the abstracted version of f.

3.3 Abstracting termination information

To use this approach for strictness analysis we want to know the answer to "Does this function application terminate?". Hence a natural abstract domain is the two-point domain

T = {**0**, **1**} ordered by **0** ≤ **1**

We intend that

ABS x = **1** iff x MAY PERHAPS terminate

or, equivalently,

ABS x = **0** iff x DEFINITELY fails to terminate

and also, of course, that ABS satisfies the basic commutativity equation for any function f, depicted by the diagram:

$$
\begin{tikzcd}
\mathbf{D} \arrow[r, "\mathrm{ABS}"'] \arrow[d, "f"] & \{\mathbf{0}, \mathbf{1}\} \arrow[d, "f\#"] \\
\mathbf{D} \arrow[r, "\mathrm{ABS}"'] & \{\mathbf{0}, \mathbf{1}\}
\end{tikzcd}
$$

Supposing that we could find such an ABS, and we calculate that

$$f\#\ \mathbf{0} = \mathbf{0}$$

Then we are sure that

$$f\ \perp = \perp$$

that is, we have discovered at compile time that f is strict.

3.4 Developing the abstraction function ABS

The whole system depends on the effectiveness of ABS. Clearly one ABS that satisfies the constraints is

BadABS x = **1**   (for any x)

This ABS gives us no information at all. We can easily do better than this. For instance, we know that

ABS (f x) = f# (ABS x)

and repeated application of this rule will push applications of ABS and # down to the leaves of the expression. For example, consider the function definition

g p q r   =   if (= p 0) (+ q r) (+ q p)

By abstracting both sides of the definition, we can see that

```
g# (ABS p)(ABS q)(ABS r)   =   if#   (=# (ABS p) (ABS 0))
                                     (+# (ABS q) (ABS r))
                                     (+# (ABS q) (ABS p))
```

Now, clearly

ABS constant = **1**   (constants always terminate)

and we can rename the bound variables, to get

$$g\#\ \mathbf{p}\ \mathbf{q}\ \mathbf{r} \quad = \quad if\#\ (=\#\ \mathbf{p}\ \mathbf{1})\ (+\#\ \mathbf{q}\ \mathbf{r})\ (+\#\ \mathbf{q}\ \mathbf{p})$$

Furthermore, we know that

(= x y) MAY terminate <=> (x MAY terminate) AND (y MAY terminate)

so

$$(=\#\ \mathbf{x}\ \mathbf{y}) = \mathbf{x}\ \&\ \mathbf{y}$$

where we define "&" as the boolean AND operator (in **T**). Similarly we define "|" as OR. The definition of "+#" is identical to that of "=#". However, "if#" is more interesting. We know that

(if x y z) MAY terminate <=> (x MAY terminate) AND
((y MAY terminate) OR (z MAY terminate))

Thus

$$if\#\ \mathbf{x}\ \mathbf{y}\ \mathbf{z} = \mathbf{x}\ \&\ (\mathbf{y}\ |\ \mathbf{z})$$

So all we need to do is to construct the # versions of all the basic functions, and we can compute the # versions of any user defined functions. In fact we can regard abstract interpretation as a non-standard semantics, in which the only difference from the standard semantics is the interpretation of primitive functions.

Having obtained # versions of all user functions, we can now compute their strictness, thus

$$\begin{aligned} g\#\ \mathbf{p}\ \mathbf{q}\ \mathbf{r} \quad &= \quad (\mathbf{p}\ \&\ \mathbf{1})\ \&\ ((\mathbf{q}\ \&\ \mathbf{r})\ |\ (\mathbf{q}\ \&\ \mathbf{p})) \\ &= \quad \mathbf{p}\ \&\ \mathbf{q}\ \&\ (\mathbf{p}\ |\ \mathbf{r}) \end{aligned}$$

For example, to find whether g is strict in p, we compute

$$\begin{aligned} g\#\ \mathbf{0}\ \mathbf{1}\ \mathbf{1} \quad &= \quad \mathbf{0}\ \&\ \mathbf{1}\ \&\ (\mathbf{0}\ |\ \mathbf{1}) \\ &= \quad \mathbf{0} \end{aligned}$$

This tells us that g fails to terminate if p fails to terminate, even if all the other arguments terminate; so g is strict in p. To discover strictness in q and r, we compute

$$\begin{aligned} g\#\ \mathbf{1}\ \mathbf{0}\ \mathbf{1} \quad &= \quad \mathbf{0} \quad \text{(so g is strict in q)} \\ g\#\ \mathbf{1}\ \mathbf{1}\ \mathbf{0} \quad &= \quad \mathbf{1} \quad \text{(so g is not strict in r)} \end{aligned}$$

3.5 Coping with recursion

There is one fly in the ointment, which is that the user defined functions may be recursive. To see that we can't simply execute the #'d function normally, consider

f x y = if (= x 0) y (f (- x 1) y)

We get

f# **x y** = **x** & (**y** | f# **x y**)

and the evaluation of (f# **1 0**) would not terminate. This would be a disaster, because this evaluation occurs at compile time and the compiler would loop. However, it is intuitively clear that f is strict in y, and we would like the compiler to be able to deduce this.

We will now examine algorithms for dealing with recursion, beginning with two attempts that turn out to be inadequate.

3.5.1 The first wrong way At first it looks as if we could just assume that recursive calls to f# were strict in everything. Thus

f# **1 0**	=	**1** & (**0** | f# **1 0**)
	=	**1** & (**0** | (**1** & **0**))
	=	**0**

which is the correct answer. This simple method is, however, easily defeated. Consider the function

f x y z = if (= y 0) (f 0 1 x) x

The simple method says this function is strict in x and y, whereas it is, of course, only strict in y. In retrospect this seems obvious, but this mistake was actually made in two published implementations of Mycroft's work.

3.5.2 The second wrong way The reason the first method fails is that it uses a bad approximation to f#. To see this, observe that the definition of f# is a perfectly good recursive function definition. Domain theory tells us that the function thus defined is given by the least upper bound of an ascending sequence of approximations to f# - the **ascending Kleene chain** (AKC). For example,

if	f# **x y z**	=	...f#...	(a recursive definition)
then	f#0 **x y z**	=	**0**	(zeroth approximation)
	f#1 **x y z**	=	...f#0...	(first approximation)
	f#2 **x y z**	=	...f#1...	(second approximation)
and so on				

Since we are in the abstract 2-element domain, **there are only a finite number of functions of 3 arguments.** This sequence must therefore reach a limit in a finite number of steps. The first method failed because we used the first approximation only, which may not be the limit. So we must examine successive approximations until we reach a fixed point, and our problem boils down to deciding when this fixed point has been reached. Notice that the application of any f#i to any arguments will always terminate.

The second bad method says "we have reached a fixed point when the set of variables in which the approximations are strict remains unchanged from one approximation to the next". This copes with the previous counterexample, because

```
f#0 is strict in {x, y, z}
f#1 is strict in {x, y}
f#2 is strict in {y}
f#3 is strict in {y}
```

and we conclude that f# itself is strict in y alone. This method is attractive because it is quite easy to compute the set of strict variables for a function from its boolean expression. Unfortunately this is not a genuine check for a fixed point, as the following counterexample shows:

```
f x y z p    =    if    (= p 0)
                        (+ x z)
                        (+  (f y 0 0 (- p 1))  (f z z 0 (- p 1)))
```

The test is better, so the counterexample is more contorted! Working out the details of this example is left as an exercise. The results are

```
f#0 is strict in {x, y, z, p}
f#1 is strict in {x, z, p}
f#2 is strict in {z, p}
f#3 is strict in {z, p}
f#4 is strict in {p}
```

The second and third approximations are the same, so we might conclude that the AKC has converged. However, the fourth approximation shows that this is false. We call such false convergence a **plateau,** and it is these plateaus that defeat the second bad method.

The only correct way to find a fixed point is to assure ourselves that

f#n **x y z** = f#(n+1) **x y z** **for any** **x**, **y**, **z**

This looks expensive, requiring 2^3 evaluations. It can, however, be done more cheaply than this, and methods for doing so are described in Section 4.

3.6 Higher order functions

Exactly the same technique applies to higher order functions. For example, consider

```
hof g x y   =   (g  (hof (K 0) x (- y 1)))  +
                (if (= y 0) x (hof I 3 (- y 1)))
```

where K x y = x

and I x = x

Performing abstraction in a straightforward way, we get

hof# g x y = g (hof# (K# **1**) x y) & y & (x **|** hof# I# **1** y)

We need to take some care when looking for a fixed point to ensure that successive approximations deliver the same result **for all values of g.** Since g is a function, it can take a whole lattice of values (3 values in this case: (K **0**), I, and (K **1**)). This example is a particularly interesting one, since it turns out that we have to go to the fourth approximation to find a fixed point.

The validity of this natural extension to higher order functions is shown by Burn, Hankin and AbramskyBurn86a.

3.7 Order of analysis and mutual recursion

Everything we have said so far assumes that the functions being analysed have no free variables, and indeed it seems rather hard to analyse functions which do have free variables. Rather than address this problem directly, it is simpler to "lift" all function definitions to the top level using **lambda lifting**Johns85a, or Hughes' **supercombinator** transformationHughe82a, so that there are no free variables to worry about. Most state of the art compilers do this anyhow for other reasons.

Suppose the definition of a function f involves a function g but not vice versa, thus

f = ...g..f...
g = ...g......

Then we can safely first analyse g, find the fixed point of g#, and use this information in the subsequent analysis of f. This can prove very important when analysing large systems of equations since finding the fixed point of f and g simultaneously is much more costly than analysing g first, and using this information to analyse f. Unfortunately, functional programmers often write large collections of equations in a single "letrec", so all the equations may potentially be mutually recursive. This suggests that any practical compiler will perform some sort of **dependency analysis** to sort the definitions into minimal mutually recursive groups.

Notice that it is not sufficient simply to remember the arguments in which g is strict, and use only this information for the analysis of f. To see this, consider the function

g p q r = if p q r

This g is just the "if" function dressed up, but it is strict only in p. Therefore, during subsequent analysis we must use the full g#

g# **p q r** = **p** & (**q** | **r**)

not the degraded version

g# **p q r** = **p**

Similar care is needed when dealing with mutually recursive sets of equations, such as

f x = ...g...f...
g y = ...f...g...

Here we cannot fully analyse either function before the other; instead we must perform the fixed point iterations simultaneously, thus

f#0 **x**	=	**0**	g#0 **y**	=	**0**
f#1 **x**	=	...g#0...f#0...	g#1 **y**	=	...f#0...g#0...
f#2 **x**	=	...g#1...f#1...	g#2 **y**	=	...f#1...g#1...

It is slightly more efficient (and gives the same result) to use f#1 in g#1, since f#1 is now available (assuming we perform each step of the f iteration before the corresponding g step). This is the meaning of the function Z inMycro83a.

4. Improved algorithms for monitoring the AKC

In section 3.5.2, we pointed out that the second bad method produces plateaus. These plateaus occur because we choose a bad way to monitor the convergence of the AKC.

We want to discover when f#n = f#(n+1), and the efficiency of this equality test is crucial to the efficiency of the whole algorithm. In particular, we will require some **explicit** representation of the f#i functions, since we cannot perform equality tests on the functions themselves.

We know that in the worst case the cost of the test must be exponential in the number of argumentsHudak85a, but it requires considerable contortion to invent examples with plateaus, so in practice we expect rapid convergence. This suggests that we should try to develop representations and heuristics which will perform well in the common cases, and will still give correct answers (albeit more slowly) in the difficult cases. The rest of the paper is devoted to a discussion of such methods.

We need to characterise each f#i by some explicit representation which we can test for equality. We will examine three such representations: **truth tables, boolean expressions,** and **frontiers.**

4.1 Truth tables

A function may be fully characterised by a set of argument-result pairs. In the case of first order functions in the abstract domain the argument is a tuple of members of **T** (ie **0** or **1**), and the result is a member of **T**. If we write down the argument-result pairs systematically, we will produce a table looking very much like the truth table for a boolean expression. For example

a	b	f# a b
0	0	0
0	1	0
1	0	0
1	1	1

If f is higher order, and an argument to f is a function, then that argument can take not just two but a whole lattice of possible values. These can still be tabulated in the same way, but the table gets more bulky. For example, for the "hof" function of Section 3.6, we get the following sequence of truth tables (omitting y for simplicity):

g	x	hof#0	hof#1	hof#2	hof#3	hof#4	hof#5
(K 0)	0	0	0	0	0	0	0
(K 0)	1	0	0	0	0	0	0
I	0	0	0	0	0	1	1
I	1	0	0	1	1	1	1
(K 1)	0	0	0	0	1	1	1
(K 1)	1	0	1	1	1	1	1

We have reached a fixed point when the truth table for f#n is the same as that for f#(n+1). This clarifies why the second bad method fails. It monitors only a subset of the 2n combinations; the subset of all combinations where one parameter takes the value **0** and all others take the value **1**.

The obvious disadvantage of this representation is that it is **always** exponential in the number of arguments to f, regardless of how "well-behaved" f is.

4.2 Boolean expressions

In the discussion above we have denoted our abstract functions as boolean expressions over **T**, with connectives "&" and "|" ("not" is not required). Taking this hint, we can choose to represent our abstract functions as a symbolic boolean expression, in the manner of an algebra system.

This has the advantage that "well-behaved" functions can be very compactly represented. For example

f# a b c d = a & d

is a far more economical representation than a truth table with 16 entries. This argument becomes even more compelling when higher order functions are involved, where the truth tables become large.

To decide if we have reached a fixed point we must now compare two boolean expressions for equality, and here they become rather less convenient. Two boolean expressions may look different, and yet denote the same function, so we must manipulate them into some canonical form. The most common normal forms are disjunctive or conjunctive normal form (DNF and CNF). However, the former corresponds to identifying the rows of the truth table whose value is **1**, and the latter those whose value is **0**, so we are not much better off. Furthermore for some expressions DNF is a much more compact representation than CNF and vice versa, so there is a problem knowing which to use.

We can often simplify an expression by eliminating redundant terms. For example

x & (x | y) = x

This process must be performed consistently and completely if we are still to be able to compare expressions for equality.

Using boolean expressions as the representation of our abstract functions becomes completely unmanageable when we move to higher order functions. The simplest example is

f g = g (f g)

So

f# g = g (f# g)

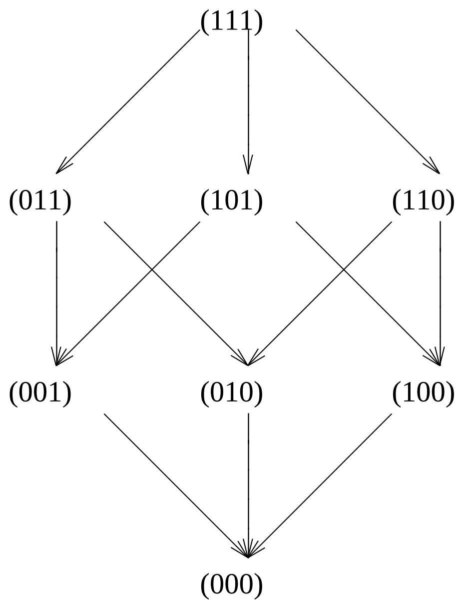
f#0 g	=	0
f#1 g	=	g 0
f#2 g	=	g (g 0)

and so on. The expression gets larger and larger at each iteration, but at no stage does it "look like" the previous iteration. This is because we can't simplify the call to g. However, from f#1 onwards, f#n = f#(n+1) **for any g,** but this is impossible to detect by examining the boolean expressions.

We conclude that representing abstract functions directly in terms of boolean expressions is probably inappropriate.

4.3 Frontiers

The possible arguments to an abstract function, f#, form a finite lattice, which we may represent as a directed graph. Each node in the graph stands for a particular combination of arguments. For example, suppose f# is a first order function of three arguments. Then the lattice of argument values ($\mathbf{T}^3$) looks like this:



If the result of applying f# to the arguments specified by one of the nodes is **0**, then that node is called an **0-node,** and similarly for **1**-nodes.

Now, f# is monotonic, so for example if

f#k **1 1 0** = **0**

then

f#k **i j 0** = **0** for all **i**, **j**.

In general, if a node **X** is a **0**-node, then every node that approximates **X** must also be a **0**-node (more intuitively we say that any node **below X** must be a **0**-node). Similarly, if a node **X** is a **1**-node, then every node that **X** approximates must also be a **1**-node (any node **above X** must be a **1**-node).

This observation gives us the clue to representing the function more efficiently. Instead of listing all the **0**-nodes or **1**-nodes, we can keep the set of nodes that constitute the boundary between **0**-nodes and **1**-nodes. We call this set a **frontier.** Specifically, the **0-frontier** is the set of **0**-nodes on the boundary, and the **1-frontier** is the corresponding set of **1**-nodes on the boundary. We want our frontier sets to be as small as possible, so we eliminate redundant points. This is easy to do, for if any two points in a frontier set are comparable then one or the other is redundant. We can now define a frontier formally.

Definition. A **0**-frontier, F, of a function, $f\#:\alpha \rightarrow \mathbf{T}$, is a set of values:$\alpha$ such that

(i) (F cuts the lattice in two, so there are **1**'s above F, **0**'s below.)
For any $v:\alpha$, $f\ v = \mathbf{0} \iff$ there is a $c \varepsilon F$ such that $v \leq c$.

(ii) (F is minimal.)
For any $c1, c2\ \varepsilon\ F$, $c1 \leq c2 \Rightarrow c1 = c2$.

There is a corresponding definition for **1**-frontiers. A frontier is a compact, canonical representation for an abstract function.

4.3.1 Monotonicity between approximations The AKC also exhibits monotonicity between one approximation and the next:

if

$$f\#n\ \mathbf{x}\ \mathbf{y}\ \mathbf{z} = \mathbf{1}$$

then

$$f\#m\ \mathbf{x}\ \mathbf{y}\ \mathbf{z} = \mathbf{1} \qquad \text{for all} \quad m \geq n$$

So as we proceed from one approximation to the next, the frontier will move away from the top element (111) and towards the bottom element (000). This gives the intuitive picture of a "low tide mark" of **1**'s starting at the top of the lattice and advancing downwards, until it comes to a halt when we reach the fixed point.

4.3.2 Generating the frontier We can generate a frontier in a similar, though less exhaustive, manner as a truth table, by applying the abstract function to an argument value to discover its result at that point. The trick lies in applying it to the minimum number of points to establish the frontier set. It is this heuristic which offers the hope of almost always getting better than worst case performance.

In moving from one iteration to the next we know that the frontier will only descend, so we can explore downwards from the current frontier. This, incidentally, suggests that the **0**-frontier will be most convenient for us, since we have reached a fixed point if all the points on the **0**-frontier remain **0**-nodes from one iteration to the next.

Unfortunately, a little practice shows that frontiers often "flip" from being rather near the top of the lattice to being rather near the bottom, and it is expensive to discover this fact by exploring downwards from the current frontier. Another way of looking at this is to see that points near the top and bottom of the lattice contain a lot of information, since they approximate (or are approximated by) many other points.

This suggests that we should explore up from the bottom in parallel with exploring down from the current frontier. In the good cases this will pay off handsomely, while in the bad cases it will not make matters much worse. Furthermore, **1**-nodes discovered while exploring upwards can be used to cut down the space being explored downwards, since any node approximated by the **1**-node will also be a **1**-node; similarly, **0**-nodes discovered while exploring downwards can reduce the space being explored upwards.

The idea underlying idea is that having several interacting activities working on a single problem may give a dramatic improvement in common case behaviour over the behaviour of any one of the individual activities working alone. This idea is not well known, but is described by KornfeldKornf79a who calls such algorithms **combinatorially implosive.**

4.3.3 Higher order functions Frontiers can handle higher order functions quite naturally; the lattice of argument values gets more complicated, though.

In the case of functions which return functions as results, we prefer to extend the definition of the function to include the "invisible" arguments, so that the result is in **T**. The purpose of doing this is primarily to discover strictness in these extra parameters, but it means incidentally that we need only concern ourselves with **0**-nodes and **1**-nodes.

When exploring the argument lattice we should use the search algorithm recursively to explore the sub-lattice generated by a functional argument.

5. Conclusions

We have given an informal account of how abstract interpretation may be used to perform strictness analysis, and discussed practical methods for finding fixed points. The worst case behaviour of any such method is exponential, but it seems to be possible to exploit the good behaviour of most cases.

6. Acknowledgements

We thank Paul Hudak and Jonathan Young for our animated transatlantic discussions on the occurrence of AKC plateaus, David Turner for directing us to Kornfeld’s paper, Geoff Burn and Chris Hankin for many stimulating discussions on higher order strictness analysis, and John Washbrook for his constructive comments.

CONTENTS