Science of Computer Programming 7 (1986) 249-278
North-Holland

STRICTNESS ANALYSIS FOR HIGHER-ORDER FUNCTIONS

Geoffrey L. BURN
GEC Hirst Research Centre, Wembley, Middlesex HA9 7PP, United Kingdom

Chris HANKIN and Samson ABRAMSKY
Department of Computing, Imperial College of Science and Technology, London SW7 2BZ, United Kingdom

Communicated by J. Darlington
Received June 1985
Revised July 1985

Abstract. Abstract interpretation is a compile-time technique which is used to gain information about a program that may then be used to optimise the execution of the program. A particular use of abstract interpretation is the strictness analysis of functional programs. This provides the key to the exploitation of parallelism in the evaluation of programs written in functional languages. In a language that has lazy semantics, the main potential for parallelism arises in the evaluation of operands of strict operators. A function is strict in an argument if its value is undefined whenever the argument is undefined. If we can use strictness analysis to detect which arguments a function is strict in, we then know that these arguments can be safely evaluated in parallel because this will not affect the lazy semantics. Experimental results suggest that this leads to significant speed-ups.

Mycroft was the first person to apply abstract interpretation to the strictness analysis of functional programs. His framework only applies to first-order functions on flat domains. Many workers have proposed practical approaches to strictness analysis of higher-order functions over flat base domains but their work has not been accompanied by extensions to Mycroft's theoretical framework. In this paper we give sound mathematical foundations for this work and discuss some of the practical issues involved. The practical approach is proved correct in relation to the theoretical framework.

1. Introduction

Abstract interpretation is a compile-time technique that may be used to infer certain properties of a program that can then be used to optimise the performance of the program. The technique has attracted a growing number of researchers since Mycroft showed how the technique could be used for the strictness analysis of functional programs [7]. Mycroft gives convincing arguments for why lazy semantics should be considered as the 'natural' semantics for functional languages but also points out the expensive overheads involved in lazy evaluation on a sequential machine. In a parallel machine the problem is compounded by the fact that lazy evaluation removes all of the potential parallelism except for the evaluation of

operands to strict operators. Mycroft shows that by annotating functions with strictness information, it is possible to optimise functional program execution by using eager evaluation techniques wherever this can be done without violating the lazy semantics.

We can illustrate the benefits of such an approach by a simple (and somewhat contrived!) example. Consider the program

$$g(x, y) = \textit{if } x = 0 \textit{ then } y \textit{ else } g(x-1, y)$$

and the call

$$g(1000, \textit{expensive})$$

If each execution of the else branch takes one time unit, a lazy evaluator would take

$$(1000 + \text{time}(\textit{expensive})) \text{ units}$$

to evaluate the call. Using Mycroft-style analysis, we are able to infer that both of the formal parameters of g are needed and may thus be passed by value. In other words, the y parameter can be evaluated in parallel with the unravelling of the recursion and the time for the whole computation will be

$$\text{maximum}(1000, \text{time}(\textit{expensive})) \text{ units}$$

potentially a significant speed-up!

Unfortunately, Mycroft's work is only valid for first-order functions over flat domains. A flat domain is just a set, S, with an extra element, $\perp$, representing non-termination. We denote the set $S \cup \{\perp\}$ by $S_\perp$. We define an ordering relation, $\leqslant$, on $S_\perp$ such that if $s_1, s_2 \in S_\perp$, then $s_1 \leqslant s_2$ if and only if $s_1 = \perp$ or $s_1 = s_2$. If the technique is to be of general use it must be extended to higher-order functions over non-flat domains. A number of researchers have developed practical techniques for the abstract interpretation of higher-order functions over flat domains. In this paper, we provide a sound theoretical basis for this work and discuss some of the practical issues involved in the abstract interpretation of higher-order functions.

In the next section we give a more detailed treatment of abstract interpretation; starting with some everyday examples we move on to a detailed account of Mycroft's work on first-order functions. Section 3 develops these ideas in a fairly obvious but informal way to examples involving higher-order functions. There is a detailed analysis of one particular example (to which we return at the end of Section 5). Having given some intuitions about the abstract interpretation of higher-order functions, we proceed in Section 4 to a development of the mathematical framework. We use the typed λ-calculus as our programming language and provide a hierarchy of abstraction maps with one map for each type. A central notion in abstract interpretation is that of *safety*; we must ensure that our abstract interpretation gives us 'true' information about a program. Such a notion is introduced and discussed in Section 4. In Section 5 we return to the practical aspects of abstract interpretation; we introduce a textual abstraction map, show how it may be viewed as a non-standard semantics for the language and prove that it is safely related to the mathematical

abstraction maps of Section 4. Finally, we suggest some directions for further work in Section 6.

Throughout this paper we assume familiarity with elementary concepts from domain theory [11, 12], denotational semantics [13] and category theory [2].

2. An introduction to abstract interpretation

We are all familiar in everyday life with the idea that often we do not require the exact answer to a question—a distance of order of magnitude of ten kilometres can be cycled, whereas a distance of order of magnitude of one hundred kilometres may require some automated form of transport. To answer the question "Do I ride my bicycle or do I go by train?", one needs only to know an approximation (order of magnitude) of the distance.

In a similar manner, we are taught at school that to tell whether a number is odd or even, all we need to do is see if the least significant digit is odd or even—a task which requires significantly less computational effort than dividing the whole number by two (unless we are dealing with a single digit number!).

What is the key concept lying behind such things? The idea is that there is some property in which we are interested, and about which we can find information without having the exact answer or doing the whole calculation.

Consider the 'rule of signs' as a slightly more complex example. Suppose the property of interest is "Is the number positive or negative or zero?". We could denote the property of being positive by $+$, the property of being negative by $-$, and zero by 0. Then we can ask what the sign of the result of a multiplication of two numbers is. The rules are well known:

$$\begin{array}{ll} + \times + = +, & 0 \times + = 0, \\ + \times - = -, & + \times 0 = 0, \\ - \times + = -, & 0 \times - = 0, \\ - \times - = +, & - \times 0 = 0, \\ 0 \times 0 = 0. & \end{array}$$

This can be put on a more formal basis by noting that what we have done is to provide an ('abstract' or 'non-standard') interpretation of the integers which captures what we are interested in—their sign. We have then raised this to give an ('abstract' or 'non-standard') interpretation of multiplication. We can define a function called *abs* which gives us this abstract interpretation of the integers:

$$abs : \mathbf{Z} \to \mathbf{sign}$$

where $\mathbf{sign} = \{+, -, 0\}$ by

$$abs(n) = \begin{cases} + & \text{if } n > 0, \\ 0 & \text{if } n = 0, \\ - & \text{if } n < 0. \end{cases}$$

This induces the interpretation on '×' given above. A serious question now arises—how can we say that a calculation in the abstract domain:

$$abs(n_1), abs(n_2) \overset{\times}{\rightarrow} abs(n_1) \times abs(n_2)$$

correctly models the calculation in the original (or concrete) domain?

$$n_1, n_2 \overset{\times}{\rightarrow} n_1 \times n_2.$$

We need a function which relates the abstract values to the concrete values. Clearly, many things abstract to give the value +, so a concretisation function, *Conc* will have type **sign** → **P(Z)**, where **P(Z)** is the powerset of integers. The canonical way to define this is

$$Conc(+) = \{n \mid n > 0\}, \qquad Conc(0) = \{0\}, \qquad Conc(-) = \{n \mid n < 0\}$$

(and so $Conc(s) = \{n \mid abs(n) = s\}$).

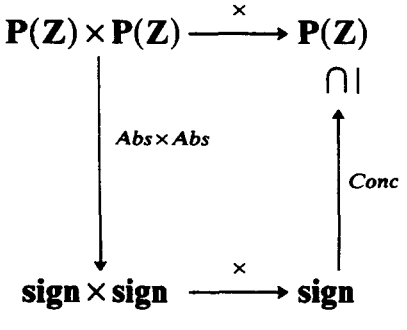
However, this is slightly unsatisfactory because the type of the result of *Conc* is not the same as the type of the argument to *abs*. So we raise *abs* to sets in a natural way, defining $Abs : \mathbf{P(Z)} \rightarrow \mathbf{sign}$ ('big Abs') and adding *?* to the set **sign** to represent a set with elements of mixed sign, as follows:

$$Abs(N) = \begin{cases} + & \text{if for all } n \in N,\ n > 0, \\ 0 & \text{if } N = \{0\}, \\ - & \text{if for all } n \in N,\ n < 0, \\ ? & \text{otherwise,} \end{cases}$$

and extend *Conc* to *?* by adding the equation

$$Conc(?) = \mathbf{Z}.$$

We can now say what we mean for a calculation in the abstract domain to correctly model a calculation in the concrete domain by the following diagram:



where × is raised to **P(Z)** × **P(Z)** pointwise. This says that when we do the calculation in the abstract domain, and then concretise the results, we always get a superset of the results we would have obtained in the concrete domain. Then if the abstract result is a particular sign, the actual set of results must all have that sign. For example, $\{1, 2\} \times \{3, 4\} = \{3, 4, 6, 8\}$ in the concrete domain, while going around the diagram in the other way we obtain $Conc(+ \times +) = Conc(+) = \{n \mid n > 0\}$, so we

have lost some information (we now have all the positive integers instead of just $\{3, 4, 6, 8\}$), but for this example we still have only the positive numbers, and we were only interested in the sign of the result.

Mycroft [7] was the first person to apply the ideas of abstract interpretation to strictness analysis. If we can evaluate arguments to a function before applying the function to them ('eagerly'), then, firstly, we save the space of creating a closure, necessary for 'lazy' evaluation, and secondly, there is the possibility of parallel evaluation—evaluate all of the arguments in parallel. However, this gives the wrong semantics—we may attempt to compute an undefined argument which is not needed by the function. Perhaps we can find out if a function definitely needs its argument, for if it does, then it makes no difference when we calculate this argument; if it is undefined in some instance, then because the function definitely needs the argument, the result of the function application will be undefined.

What is the property we are interested in? We are interested in whether a function is strict in a particular argument, that is, $f: D^n \to D$ is strict in its ith parameter if $f(a_1, \ldots, a_{i-1}, \perp, a_{i+1}, \ldots, a_n) = \perp$ for all $a_k \in D$, $k \neq i$. Thus we wish to build a framework of abstract interpretation so that we can ask whether a function definitely needs its argument or not.

(Note that Mycroft also considered a second abstract interpretation which captured the idea of definite termination. Although this information can be used to find safe opportunities for eager evaluation, it does not guarantee that arguments evaluated eagerly will actually be used by the function. Therefore we only consider strictness analysis in this paper.)

Consider again the multiplication function. Then for all $n \in \mathbf{Z}$,

$$\perp \times n = \perp, \qquad n \times \perp = \perp.$$

If we now consider multiplication to work over $\mathbf{P}(\mathbf{Z}) \times \mathbf{P}(\mathbf{Z})$, by taking a pointwise extension of multiplication, we obtain an interpretation of multiplication for which the following equations hold:

$$\perp \times \perp = \perp, \qquad \mathbf{Z}_\perp \times \perp = \perp,$$

$$\perp \times \mathbf{Z}_\perp = \perp, \qquad \mathbf{Z}_\perp \times \mathbf{Z}_\perp = \mathbf{Z}_\perp.$$

The fact that an argument is $\perp$ means that it is definitely undefined, while we can interpret an argument being $\mathbf{Z}_\perp$ as meaning that we do not know what the value of the argument is (it could be any element of $\mathbf{Z}_\perp$), and so it may (or may not) be defined. If we write 0 for $\{\perp\}$ and 1 for $\mathbf{Z}_\perp$ in the above, where we are using the domain $2 = \{0, 1\}$ with $0 \leqslant 1$ which appears in [7], we obtain the following:

$$0 \times 0 = 0, \qquad 1 \times 0 = 0,$$

$$0 \times 1 = 0, \qquad 1 \times 1 = 1.$$

We would obtain the same result for '+', as for any other strict function of two arguments. If we look at this more carefully, we see that it is just the table for x

and y! Similarly, since the conditional $if(x, y, z)$ may terminate if and only if x may terminate and either y or z may terminate, we see that the abstract interpretation of $if(x, y, z)$ is x **and** (y **or** z). From now on, we will represent the abstract interpretation of e as $e^{\#}$.

We can extend this to user-defined functions by 'propagating the # inwards' (this is formalised in Section 5). For example,

$$f(x, y, z) = x + (y \times z),$$

$$f^{\#}(x, y, z) = x \textbf{ and } (y \times z)^{\#} = x \textbf{ and } y \textbf{ and } z.$$

To see if f needs its first argument, we put in 0 for x and 1 for y and z (representing the idea that we are putting in $\perp$ for x and testing to see what the result will be if y and z can take on any other values), and see if $f(0, 1, 1) = 0$, which it clearly does. Hence f needs its first argument and indeed it also needs its second and third arguments.

Recursive functions are handled by propagating the # inwards and taking the least fixed point. For example,

$$f(x, y) = \text{if } x = 0 \textit{ then } y \textit{ else } f(x - 1, y),$$

$$f^{\#}(x, y) = (x = 0)^{\#} \textbf{ and } (y \textbf{ or } f^{\#}(x, y)) = x \textbf{ and } (y \textbf{ or } f^{\#}(x, y)).$$

What should the first approximation to $f^{\#}$ be? Clearly it should be $\lambda x.\lambda y.0$. Substituting this into the formula we obtain $\lambda x.\lambda y.x$ **and** y as the second approximation. All other approximations to $f^{\#}$ will be the same, so $f^{\#}(x, y) = x$ **and** y, and so f needs both its arguments.

Mutual recursion is handled just as easily, the details being left to the reader.

3. Abstract interpretation and higher-order functions

In the last section, abstraction was defined for first-order functions over flat base domains; we have not yet considered higher-order functions. The rest of the paper extends the notion of abstract interpretation for strictness analysis to higher-order functions over flat base domains.

Consider the *apply* function

$$apply = \lambda f^{A \to A}.\lambda x^{A}.f(x),\ ^{1}$$

where A is a flat domain. Taking the lead from the first-order case, we may wish to 'propagate the # inwards, changing all the A's to 2's'. So, we would make our

[1] Because we are dealing with the typed λ-calculus, we superscript variables with their types. Thus in the above case we have that f is a parameter of type $A \to A$. For clarity we only put type super-scripts on the variables where they are bound by the λ, understanding that occurrences of the variables which appear in the body of the λ-abstraction also have this type.

abstract interpretation of the apply function to be

$$apply^{\#} = \lambda f^{2\to 2}.\lambda x^2.f(x).$$

We can now ask whether *apply* needs its arguments. We need some concept of '0' for the function space $2 \to 2$ to see if *apply* needs its functional argument. How about using $\lambda x^2.0$? Then

$$apply^{\#}(\lambda x^2.0, 1) = (\lambda x^2.0)1 = 0$$

and so *apply* needs its functional argument. We must now test to see if it needs its second argument. But what does this mean? It will only need it if $apply^{\#}(g, 0) = 0$ for all g in $2 \to 2$. So, we need to try it for each g in $\{\lambda x^2.0, \lambda x^2.x, \lambda x^2.1\}$. However, since $apply^{\#}$ is monotonic (in fact, it is also continuous) we need only test it for $\lambda x^2.1$, for if it is zero for this, then it will be zero for all the other possibilities. So let us try this:

$$apply^{\#}(\lambda x^2.1, 0) = (\lambda x^2.1)0 = 1.$$

We see that *apply* does not definitely need its second argument, which is correct, for we may sometimes apply it to a strict function and sometimes apply it to a non-strict function. But here are the seeds of another idea. Suppose we did not try to find out whether *apply* needed its second argument until it was actually applied to a function. Then in the cases where it was applied to a strict function, we could pass the second parameter by value, whereas in the cases of a non-strict function, we would not be able to do this.

What if we only ever found out if a function needed its first argument? Then, if it did, when it was applied to something, we could label that apply node with "need" to indicate that the function definitely needed its argument. When a function is applied to an argument, we can then ask whether the result needs its argument (and so if the original function needs its second argument) and so on. In this way we are able to label every apply node in the entire program, thus finding sources of parallelism. We see that we are able to evaluate both arguments to strict functions in parallel. For instance, '+' is strict in its first argument and '$(+e)$' is strict in its argument and so both apply nodes will be labelled with "need". Thus there are two distinct passes to the process, the first works out the abstract interpretation of function definitions (allowing us to find out whether they need their first argument) and the second annotates apply nodes.

Further, we can see that we are naturally dealing with curried functions which we supply with their arguments one at a time.

This requires another imaginative step, for certainly the function values will no longer be in the base domain, 2, something that is also true of more general higher-order functions. Returning to the *apply* example again, we see that

$$apply^{\#}(\lambda y^2.0) = \lambda x^2.0$$

which is the undefined function in $2 \to 2$, in other words, the bottom element. Clearly

this is undefined if it is applied to anything, and so *apply* will be undefined for all values of its 'second' argument if it is passed an undefined 'first' argument. So it seems to be a rule that if f is in the domain $A \to B$ (for A and B being domains of any height), then if

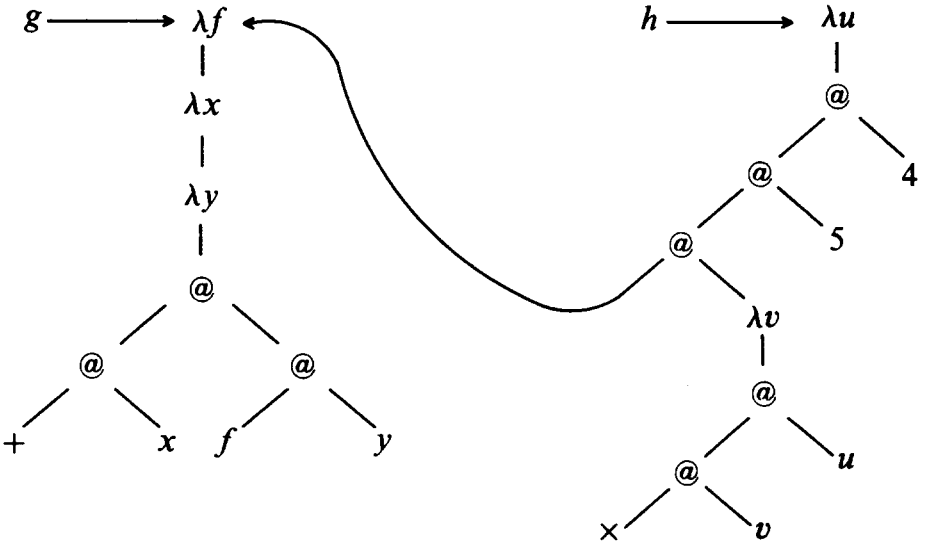
$$f^{\#}(\perp_A^{\#}) = \perp_B^{\#},$$

then f is undefined if its argument is undefined. This is a result of our notion of safety, which is discussed in the next section.

Let us look at a slightly more complex example, the two equations:

$$g = \lambda f^{A \to A}.\lambda x^A.\lambda y^A.x + f(y), \qquad h = \lambda u^A.g(\lambda v^A.v \times u)5\ 4.$$

These two functions have the following graph [14]:

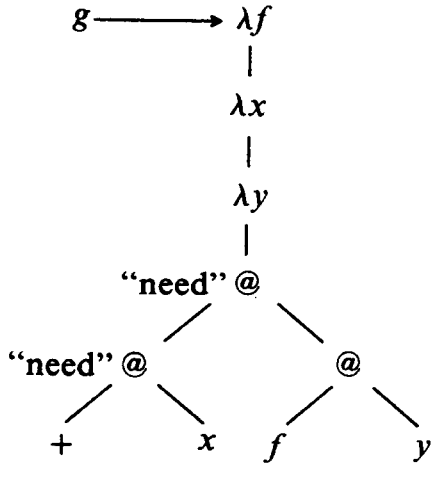


Then

$$g^{\#} = \lambda f^{2 \to 2}.\lambda x^2.\lambda y^2.x \textbf{ and } f(y),$$

$$h^{\#} = \lambda u^2.g^{\#}(\lambda v^2.v \textbf{ and } u)1\ 1 = \lambda u^2.u.$$

Having obtained the abstract interpretation of the two functions, we can annotate all the apply nodes in the graphs of the functions. Reducing innermost application nodes first, we notice that in the definition of g we have two apply nodes for the strict operator '+' and both of these can be annotated with "need":

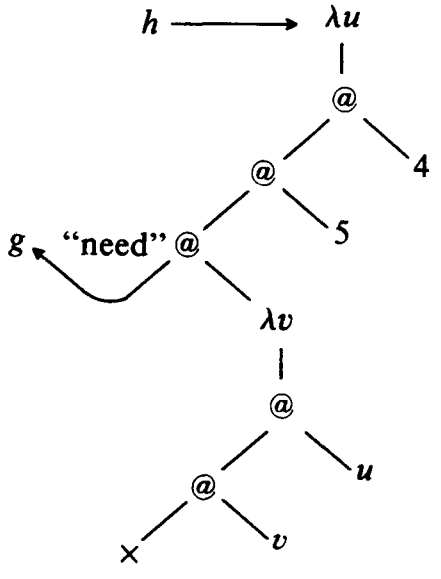


Note that we are unable to say anything about the third apply node because the annotation will depend on the function f that is supplied as a parameter and may be different for different parameters.

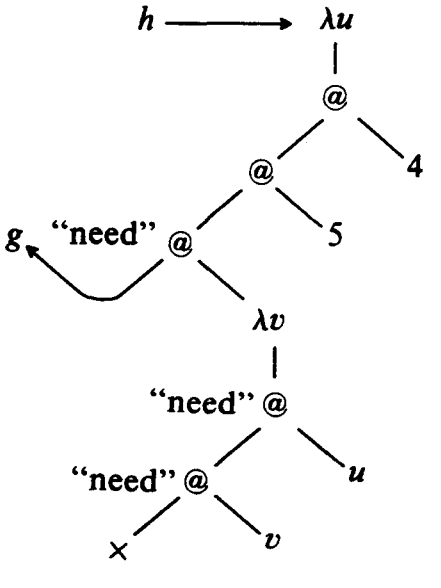
The situation for h is slightly more complicated. The innermost application node has g as the function. So we must see whether g needs its (first) parameter.

$$\begin{aligned} g^{\#}(\lambda z^2.0) &= \lambda x^2.\lambda y^2.x \ \textbf{and}\ (\lambda z^2.0)(y) \\ &= \lambda x^2.\lambda y^2.x \ \textbf{and}\ 0 \\ &= \lambda x^2.\lambda y^2.0 \end{aligned}$$

which is clearly the bottom element of $2 \to 2 \to 2$, so we can say that g needs its first argument and annotate the graph for h accordingly:



Proceeding as we did for g, we can annotate the two apply nodes for the function '×':



Next we ask if $g(\lambda v^A.v \times u)$ needs its parameter. We notice that $\lambda v^A.v \times u$ has a free variable that is bound in an outer context; for the purposes of this analysis we set this to be the top element of the appropriate abstract domain in the abstract interpretation, indicating that it may be defined (that is, we know no information about it):

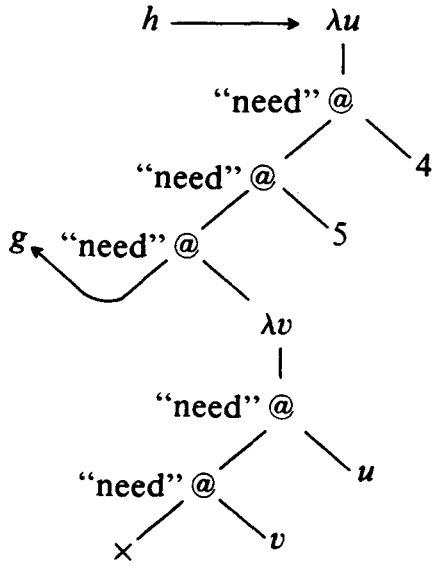
$$\begin{aligned} g^{\#}(\lambda v^A.v \times u)^{\#}(0) &= g^{\#}(\lambda v^2.v \textbf{ and } 1)(0) \\ &= \lambda y^2.0 \textbf{ and } 1 \\ &= \lambda y^2.0 \end{aligned}$$

which is the bottom of the domain $2 \to 2$ and so the parameter is needed.

There is one more apply node we have to check, and so we ask if $g(\lambda v^A.v \times u)$ 5 needs its argument.

$$\begin{aligned} g^{\#}(\lambda v^A.v \times u)^{\#}5^{\#} &= (\lambda f^{2\to 2}.\lambda x^2.\lambda y^2.x \textbf{ and } f(y))(\lambda v^2.v \textbf{ and } 1)1 \\ &= (\lambda x^2.\lambda y^2.x \textbf{ and } y)1 \\ &= \lambda y^2.y \end{aligned}$$

and since $(\lambda y^2.y)(0) = 0$, we see that it does need its parameter and so we can produce the final annotated graph for h:



Note that in an implementation we do not need to find out whether a function needs its parameter every time it is applied. We could store this information along with the abstract interpretation of the function after finding it out the first time. This is especially useful for functions like addition where we know we can always annotate both apply nodes for this function with "need".

4. Abstraction of semantic domains

In this section we develop the mathematical framework for the abstract interpretation of a language with higher-order functions over flat base domains. We do this by first defining the domains in which our concrete functions and our abstracted

functions exist, and then defining a hierarchy of abstraction and concretisation maps between them. A notion of *safety* is introduced, where we say that calculation in the abstract domain, in some sense, correctly models computation in the concrete domain, and our abstraction and concretisation maps are shown to satisfy this criterion.

It must be noted that we are working with *semantic* objects here. For example, the textual expressions "$+5$", "$\lambda x^A.x+5$" and "$\lambda x^A.2\times 2+x+1$" all denote the same semantic object (under a normal interpretation)—the function of one argument which adds 5 to any argument. To obtain these semantic objects, we can just take a denotational semantics of the program text.

We are working with the typed λ-calculus with a single ground type, A. The syntax of type expressions, ranged over by α, β is then given by

$$\alpha ::= A \mid \alpha \to \beta.$$

Now let D_A be a flat domain containing, amongst other things, the integers and the booleans.[2] Then we can recursively define the domains at higher types by $D_{\alpha\to\beta} = D_\alpha \to D_\beta$. Then the domain of the typed λ-calculus is

$$D = +\{D_\alpha \mid \alpha \text{ is a type expression}\}$$

where '$+$' is the separated sum functor. Instead of this we could have defined the domain $D = A + D \to D$ and then interpreted correctly typed expressions in suitable retracts of D. However, we have chosen the first method of forming D because it more closely mirrors the way we wish to proceed with the theoretical development.

In the same manner, we can define $B_A = 2$ and $B_{\alpha\to\beta} = B_\alpha \to B_\beta$. We can then say

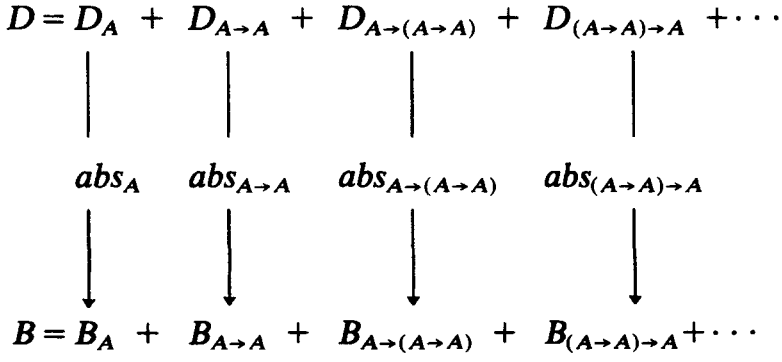
$$B = +\{B_\alpha \mid \alpha \text{ is a type expression}\}$$

where again '$+$' is the separated sum functor.

The idea is that B is to be our domain of abstract interpretations of elements of D. We will not define an abstraction map

$$D \xrightarrow{abs} B$$

directly, but will define maps which mirror the structure of the domains, as shown in the following diagram:



[2] At the expense of decreased clarity, we could construct D_A as a sum of its various components, with injection and projection functions so that arguments to operators like '$+$' and 'if' would be correctly typed. We prefer not to worry about such details here.

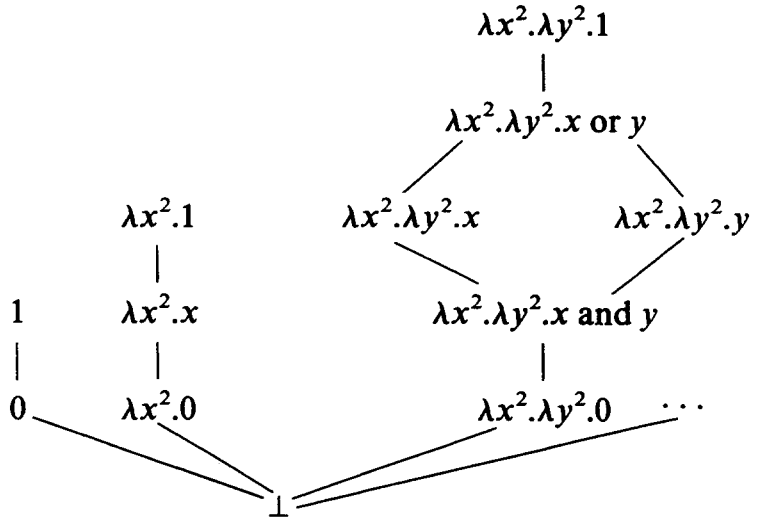
To make notation easier, we will write typed λ-expressions in the following form:

$\lambda x^{\alpha}.e$ for $\lambda x^{D_\alpha}.e$

$\lambda x^{\bar{\alpha}}.e$ for $\lambda x^{B_\alpha}.e$

and we similarly use $\lambda x^{\mathbf{P}\alpha}.e$ and $\lambda x^{\mathbf{P}\bar{\alpha}}.e$. Also we write $\lambda x^{2}.e$ rather than $\lambda x^{\bar{A}}.e$, with obvious extensions to other 'barred types', and we subscript $\perp$ by the type rather than the domain, thus using $\perp_\alpha$ for $\perp_{D_\alpha}$ and $\perp_{\bar{\alpha}}$ for $\perp_{B_\alpha}$.

Before we go on and define the abstraction maps, it is useful to have a look at the structure of B. It has some extremely important properties. In particular, each summand of B is a complete lattice, as indicated below:



This means that the test for strictness of a function

$$f: \alpha_1 \to \alpha_2 \to \cdots \to \alpha_n \to A$$

reduces to testing whether

$$(abs_{\alpha_1 \to \alpha_2 \to \cdots \to \alpha_n \to A}(f))\perp_{\overline{\alpha_1}} \top_{\overline{\alpha_2}} \cdots \top_{\overline{\alpha_n}} = 0$$

(where $\top_{\overline{\alpha_i}}$ is the top element of B_{α_i}), for if this holds then, by monotonicity, we have for any arguments $\overline{a_i} \in B_{\alpha_i}$

$$(abs_{\alpha_1 \to \alpha_2 \to \cdots \to \alpha_n \to A}(f))\perp_{\overline{\alpha_1}} \overline{a_2} \cdots \overline{a_n} = 0$$

and so indeed $(abs_{\alpha_1 \to \alpha_2 \to \cdots \to \alpha_n \to A}(f))(\perp_{\overline{\alpha_1}})$ is the bottom element of $B_{\alpha_2 \to \cdots \to \alpha_n \to A}$.

Furthermore, all the domains B_α are finite, which means we are guaranteed to be able to find fixed points and do any other calculations we may be required to do in a finite amount of time.

We shall now develop the basic definitions and facts from domain theory which we will be using throughout the rest of the paper. We shall employ some terminology from elementary category theory (see [2]), but this is not essential for understanding what we do. The proofs of the basic facts cited below are either directly in the literature, or obtainable by minor modifications therefrom; see [5, 9].

We shall be working over the category of domains described in [11, 12]. The objects of this category are the bounded-complete ω-algebraic cpo's, and the morphisms are the continuous functions between domains. The composition of morphisms $f: D \to E$, $g: E \to F$ is written thus:

$$g \circ f: D \to F.$$

The identity morphism on D is written id_D. Given domains D and E, the domain $D \to E$ is formed by taking all continuous functions from D to E, with the pointwise ordering:

$$f \leqslant g \quad \text{iff} \quad \text{for all } x \in D, f(x) \leqslant g(x).$$

Given a domain D, then $\mathbf{P}D$, the *Hoare* (*lower* or *partial correctness*) *power domain* is formed by taking as elements all non-empty Scott-closed[3] subsets of D, ordered by subset inclusion. A subset $X \subseteq D$ is *Scott-closed* if

(i) If $Y \subseteq X$ and Y is directed, then $\bigsqcup Y \in X$.

(ii) If $y \leqslant x \in X$, then $y \in X$.

The least Scott-closed set containing X is written X^*.

Another useful concept is that of *left-closure*; a set $X \subseteq D$ is left-closed if it satisfies (ii) above. The left closure of a set X is written $\mathbf{LC}(X) = \{y \mid \text{there exists } x \in X, y \leqslant x\}$.

Note that for elements of the Hoare power domain, the subset inclusion ordering is equivalent to the well-known *Egli–Milner* ordering:

$$X \subseteq Y \text{ iff for all } x \in X, \text{there exists } y \in Y, x \leqslant y$$

$$\text{and for all } y \in Y, \text{there exists } x \in X, x \leqslant y.$$

We shall also apply $\mathbf{P}$ to morphisms. If $f: D \to E$, then $\mathbf{P}f: \mathbf{P}D \to \mathbf{P}E$ is defined thus:

$$(\mathbf{P}f)(X) = \{f(x) \mid x \in X\}^*.$$

The main properties of $\mathbf{P}$ are:

(P1) If D is a domain, $\mathbf{P}D$ is a domain.

(P2) If $f: D \to E$, $\mathbf{P}f: \mathbf{P}D \to \mathbf{P}E$ is a continuous function.

(P3) $\mathbf{P}(f \circ g) = (\mathbf{P}f) \circ (\mathbf{P}g)$.

(P4) $\mathbf{P}id_D = id_{\mathbf{P}D}$.

This says that $\mathbf{P}$ is a *functor* from the category of domains to itself. A further property of $\mathbf{P}$ is that is is *locally monotonic* and *continuous*. This means that if $\{f_i\}$ is a chain of functions in $A \to B$, then for all i, $\mathbf{P}f_i \leqslant \mathbf{P}f_{i+1}$, and $\mathbf{P}(\bigsqcup f_i) = \bigsqcup \mathbf{P}f_i$.

Why are we using the Hoare power domain construction? Previous work on the theory of strictness analysis, e.g. [7, 8], has either used the Plotkin power domain, or modifications thereof. However, that work was concerned with *two* kinds of

[3] This terminology is due to the fact that these are the closed sets with respect to the Scott topology (cf. for example [4]).

abstract interpretation; strictness analysis—definitely will *not* terminate—and also information that a computation definitely *will* terminate. The latter kind of analysis requires the Plotkin power domain. However, as already mentioned, we are only interested in strictness analysis. The Hoare power domain is well suited to this purpose, since for example at a flat domain F, $\mathbf{P}F$ contains the sets $\{\perp\}$ and F, the first of which corresponds to definite non-termination, the second to possible termination. The Hoare power domain is also pleasant to work with from a technical point of view.

We shall need to use some additional constructions associated with the power domain functor. Firstly, for each domain D we have a map

$$\{\cdot\}_D : D \to \mathbf{P}D$$

defined by

$$\{d\}_D = \mathbf{LC}(\{d\}).$$

This satisfies the following properties:

(P5) $\{\cdot\}_D$ is continuous.
(P6) For $f: D \to E$, $\mathbf{P}f \circ \{\cdot\}_D = \{\cdot\}_E \circ f$.

This says that $\{\cdot\}$ is a *natural transformation* from $\mathbf{I}$, the identity functor on the category of domains, to $\mathbf{P}$.

Secondly, for each domain D we define

$$\uplus_D : \mathbf{PP}D \to \mathbf{P}D$$

by

$$\uplus_D(\Theta) = \{x \mid \text{for some } X \in \Theta,\ x \in X\} = \bigcup \Theta$$

This satisfies

(P7) $\uplus_D$ is continuous.
(P8) For $f: D \to E$, $\uplus_E \circ \mathbf{PP}f = \mathbf{P}f \circ \uplus_D$.

This says that $\uplus$ is a natural transformation from $\mathbf{P}^2$ to $\mathbf{P}$.

Now $(\mathbf{P}, \{\cdot\}, \uplus)$ forms a *monad* or *triple*. We shall not use this fact, but we will use the following, additional observation. Suppose D is a domain which is a complete lattice. Then the least upper bound operation, viewed as a function

$$\bigsqcup : \mathbf{P}D \to D$$

satisfies

(P9) $\bigsqcup$ is continuous.
(P10) $\bigsqcup \circ \{\cdot\}_D = id_D$.
(P11) $\bigsqcup \circ \mathbf{P}(\bigsqcup) = \bigsqcup \circ \uplus_D$.

This says that $\bigsqcup : \mathbf{P}D \to D$ is an *algebra* of the monad $(\mathbf{P}, \{\cdot\}, \uplus)$. We will use (P9) and (P10) in the sequel.

Henceforth, we shall omit the subscripts from instances of $\{\cdot\}$ and $\uplus$ where they are clear from the context. The facts we shall be assuming about the constructions introduced above are summarised in (P1)–(P11). By using 'function-level reasoning', we are able to give simple, algebraic proofs of most results.

We now turn to the crux of this paper, namely the definitions of the abstraction maps. This provides the basis for the entire theory subsequently developed. Recall that our task is to define, for each type α, a map

$$abs_\alpha : D_\alpha \to B_\alpha.$$

We proceed by induction over types. At each type α, we will in fact define *three* maps:

$$\begin{aligned} &abs_\alpha : D_\alpha \to B_\alpha && \text{(pronounced 'little abs')},\\ &Abs_\alpha : \mathbf{P}D_\alpha \to \mathbf{P}B_\alpha && \text{(pronounced 'big abs')},\\ &Conc_\alpha : \mathbf{P}B_\alpha \to \mathbf{P}D_\alpha. \end{aligned}$$

We have already motivated the need for the concretisation maps, and the introduction of power domains, in Section 2.

The crucial part of the definition is abs_α. Once we have defined abs_α, the other definitions follow directly:

$$Abs_\alpha = \mathbf{P}abs_\alpha, \qquad Conc_\alpha(S) = \uplus\{T \mid Abs_\alpha(T) \leqslant S\}.$$

Moreover, the base case for the definition of *abs*, that is abs_A, is straightforward—it is exactly the same as Mycroft's definition of HALT in [7]:

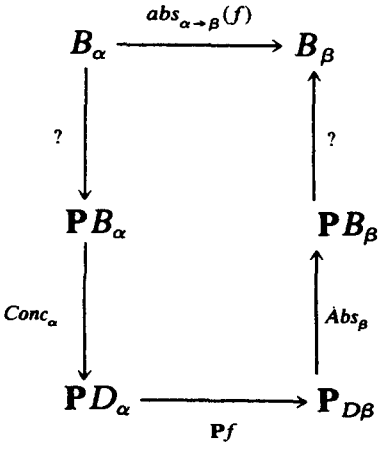
$$abs_A(a) = 1 \quad (a \neq \bot_A), \qquad abs_A(\bot_A) = 0.$$

(Note that we then have

$$Conc_A(\{0\}) = \{\bot_A\}, \qquad Conc_A(\{0, 1\}) = D_A,$$

as expected.)

Thus what remains to be done is the inductive step in the definition of *abs*, i.e. to define $abs_{\alpha\to\beta}$, assuming we have already defined abs_α, Abs_α and $Conc_\alpha$, and abs_β, Abs_β and $Conc_\beta$. A diagram is helpful here:



Given $f: D_\alpha \to D_\beta$, we want to define $abs_{\alpha\to\beta}(f)$. Starting with an argument b in B_α, a diagram chase using the functions already defined will give us a result in B_β, *provided* we can

(i) embed b in $\mathbf{P}B_\alpha$,

(ii) collapse the set of abstractions of possible results in $\mathbf{P}B_\beta$ into a *single* abstract value in B_β.

Clearly, we have a ready-made solution to (i) in the form of $\{\cdot\}$, whose purpose is precisely to embed a domain in its power domain. For (ii), we use the function $\bigsqcup \cdot \mathbf{P}B_\beta \to B_\beta$. The intuitive motivation for this is that in performing (ii) we must ensure that any possibility of termination in the actual computation is recorded, in order to preserve the safety of our abstract interpretation. This is done by taking the least upper bound, as the minimal safe representative.

We can also justify the choice of $\bigsqcup$ formally, by noting that it gives an algebra for the power domain monad, and indeed is the unique function from $\mathbf{P}B_\beta$ to B_β which does so. In particular, the identity (P10) will be important in the sequel.

Thus our definition of $abs_{\alpha\to\beta}$ is

$$abs_{\alpha\to\beta}(f) = \bigsqcup \circ Abs_\beta \circ \mathbf{P}f \circ Conc_\alpha \circ \{\cdot\}.$$

If we define

$$\Psi : (\mathbf{P}B_\alpha \to \mathbf{P}B_\beta) \to (B_\alpha \to B_\beta)$$

by

$$\Psi(g) = \bigsqcup \circ g \circ \{\cdot\},$$

then obviously we have

$$abs_{\alpha\to\beta}(f) = \Psi(Abs_\beta \circ \mathbf{P}f \circ Conc_\alpha).$$

A useful property of Ψ, which follows from (P6) and (P10), is:

(P12) $\Psi(\mathbf{P}f) = f$.

There are many things which we must show about these definitions. We must show that $Conc_\alpha$ is well defined, that is, that the set $\{T \mid Abs_\alpha(T) \leqslant S\}$ is an element of $\mathbf{PP}D_\alpha$, so that we can apply $\biguplus$ to it. We must also prove that all the functions defined are continuous. All these things are proved in Lemma 4.2 in an induction over the type α, but we first prove an auxiliary result, namely that $\mathbf{P}$ preserves onto-ness.

Lemma 4.1. *If $f: D \to E$ is onto, then $\mathbf{P}f: \mathbf{P}D \to \mathbf{P}E$ is onto, where $\mathbf{P}$ is the Hoare power domain functor.*

Proof. Let $S \in \mathbf{P}E$. Then S is Scott-closed and non-empty. Hence, since f is continuous (and thus continuous with respect to the Scott topology), $T = f^{-1}(S)$ is

Scott-closed, and non-empty since f is onto, and so is in $\mathbf{P}D$. Now

$$\begin{aligned}\mathbf{P}f(T) &= \{f(t) \mid t \in T\}^* \\ &= S^* \qquad \text{since } f \text{ is onto} \\ &= S. \qquad\qquad \square\end{aligned}$$

Lemma 4.2. *For all types α we have the following:*

(i) *abs_α and Abs_α are continuous.*

(ii) *There is a continuous function $abs_\alpha^{-1}: B_\alpha \to D_\alpha$ which is a right inverse of abs_α, i.e. $abs_\alpha \circ abs_\alpha^{-1} = id_{B_\alpha}$.*

(iii) *abs_α and Abs_α are onto.*

(iv) *$Abs_\alpha \circ Conc_\alpha = id_{\mathbf{P}B_\alpha}$.*

(v) *$Conc_\alpha$ is well defined and continuous.*

Proof. These are trivially true for the base case where $\alpha = A$ except for (ii) where we define

$$abs_A^{-1}(0) = \perp_A, \qquad abs_A^{-1}(1) = a \quad \text{for some } a \in A,\ a \neq \perp_A.$$

This is clearly continuous and has the required inverse property. Assume (i) to (v) are true for types α and β and we will prove them true for $\alpha \to \beta$.

(i) $abs_{\alpha\to\beta} = \lambda f^{\alpha\to\beta}.\Psi(Abs_\beta \circ \mathbf{P}f \circ Conc_\alpha)$ and is continuous because it is the composition of continuous functions by induction (i) and (v). Hence $Abs_{\alpha\to\beta} = \mathbf{P}abs_{\alpha\to\beta}$ is continuous by (P2).

(ii) Define $abs_{\alpha\to\beta}^{-1} = \lambda f^{\bar\alpha\to\bar\beta}.abs_\beta^{-1} \circ f \circ abs_\alpha$. This is certainly continuous because it is the composition of continuous functions. Let $f \in B_{\alpha\to\beta}$. Then

$$\begin{aligned}abs_{\alpha\to\beta}(abs_{\alpha\to\beta}^{-1}(f)) &= abs_{\alpha\to\beta}(abs_\beta^{-1} \circ f \circ abs_\alpha) \\ &= \Psi(Abs_\beta \circ \mathbf{P}(abs_\beta^{-1} \circ f \circ abs_\alpha) \circ Conc_\alpha) \\ &= \Psi(\mathbf{P}(abs_\beta \circ abs_\beta^{-1} \circ f \circ abs_\alpha) \circ Conc_\alpha) \quad \text{(P3)} \\ &= \Psi(\mathbf{P}(f \circ abs_\alpha) \circ Conc_\alpha) \qquad \text{by induction (ii)} \\ &= \Psi(\mathbf{P}f \circ Abs_\alpha \circ Conc_\alpha) \\ &= \Psi(\mathbf{P}f) \qquad \text{by induction (iv)} \\ &= f \qquad\qquad \text{(P12)}\end{aligned}$$

and so, by the principle of extensionality, $abs_{\alpha\to\beta} \circ abs_{\alpha\to\beta}^{-1} = id_{\bar\alpha\to\bar\beta}$.

(iii) For any $\bar f \in B_{\alpha\to\beta}$, choose $f \in D_{\alpha\to\beta}$ such that $f = abs_{\alpha\to\beta}^{-1}(\bar f)$. Then $abs_{\alpha\to\beta}(f) = \bar f$. $Abs_{\alpha\to\beta}$ is then onto by Lemma 4.1.

(iv) Let $S \in \mathbf{P}B_{\alpha\to\beta}$.

$$\begin{aligned} Abs_{\alpha\to\beta}(Conc_{\alpha\to\beta}(S)) &= Abs_{\alpha\to\beta}(\uplus\{T \mid Abs_{\alpha\to\beta}(T) \leqslant S\}) \\ &= \uplus \mathbf{P}Abs_{\alpha\to\beta}(\{T \mid Abs_{\alpha\to\beta}(T) \leqslant S\}) \quad \text{(P8)} \\ &= \uplus(\{Abs_{\alpha\to\beta}(T) \mid Abs_{\alpha\to\beta}(T) \leqslant S\})^* \\ &= (\bigcup\{Abs_{\alpha\to\beta}(T) \mid Abs_{\alpha\to\beta}(T) \leqslant S\})^* \\ &\quad \text{by a simple adaptation of a result in [9, p. 477]} \\ &= (S)^* \qquad \text{since } Abs_{\alpha\to\beta} \text{ is onto} \\ &= S. \end{aligned}$$

(v) To prove well-definedness, we must show that $\{T \mid Abs_{\alpha\to\beta}(T) \leqslant S\}$ is a non-empty Scott-closed subset of $\mathbf{P}D_{\alpha\to\beta}$. It is non-empty because $Abs_{\alpha\to\beta}$ is onto (induction (iv)). Denoting $\{T \mid Abs_{\alpha\to\beta}(T) \leqslant S\}$ by Θ, to show that Θ is Scott-closed we need to show that

(a) Θ is left-closed and

(b) Θ is closed under least upper bounds of directed sets.

The first is true since if $Y \leqslant X \in \Theta$, then $Abs_{\alpha\to\beta}(Y) \leqslant Abs_{\alpha\to\beta}(X) \leqslant S$ and so $Y \in \Theta$. The second is true for if $\Delta \subseteq \Theta$ is a directed set, then $Abs_{\alpha\to\beta}(\bigsqcup\Delta) = \bigsqcup\{Abs_{\alpha\to\beta}(X) \mid X \in \Delta\}$ and since $Abs_{\alpha\to\beta}(X) \leqslant S$ for all $X \in \Delta$, $\bigsqcup\{Abs_{\alpha\to\beta}(X) \mid X \in \Delta\} \leqslant S$.

To prove continuity, we only have to prove monotonicity because $Conc_{\alpha\to\beta}$ works on a finite domain and so is continuous if it is monotonic. Let $S_1, S_2 \in \mathbf{P}B_{\alpha\to\beta}$, $S_1 \leqslant S_2$. Then

$$Conc_{\alpha\to\beta}(S_1) = \uplus\{T \mid Abs_{\alpha\to\beta}(T) \leqslant S_1\}$$

and

$$Conc_{\alpha\to\beta}(S_2) = \uplus\{T \mid Abs_{\alpha\to\beta}(T) \leqslant S_2\}.$$

Clearly, $\{T \mid Abs_{\alpha\to\beta}(T) \leqslant S_1\} \subseteq \{T \mid Abs_{\alpha\to\beta}(T) \leqslant S_2\}$ since $S_1 \leqslant S_2$ and so $Conc_{\alpha\to\beta}(S_1) \leqslant Conc_{\alpha\to\beta}(S_2)$. Thus $Conc_{\alpha\to\beta}$ is monotonic and hence continuous. □

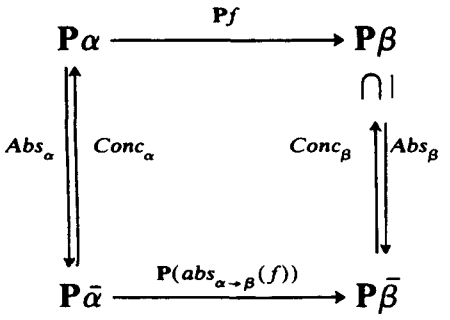
Having defined our abstraction and concretisation maps, we must define what is an appropriate notion of safety. A criterion must be established which says when calculations in the abstract domain:

$$B_\alpha \xrightarrow{abs_{\alpha\to\beta}(f)} B_\beta$$

correctly mirror calculations in the concrete domain:

$$D_\alpha \xrightarrow{f} D_\beta.$$

The following diagram expresses what we will mean by *safety*. It is essentially a generalisation of diagrams appearing elsewhere (for example [7, 8]):



In words this diagram says that if we abstract the argument (set) to the function, apply the abstracted function (pointwise) and concretise the result, then we will obtain more results than if we had applied the function (pointwise) to the argument (set). For instance, when doing strictness analysis, this means that if the calculation along the first route gives the set with only $\perp_\beta$ in it when applied to $\{\perp_\alpha\}$, then the route along the top must also have given $\{\perp_\beta\}$, and so the function must have been strict in its argument.

That the abstraction and concretisation maps we have defined satisfy the safety criterion is Proposition 4.5, and we will proceed by proving a preliminary lemma and a proposition. Proposition 4.4 is well worth taking special note of, as it is used repeatedly throughout the rest of the paper. It says that abstraction is a semi-homomorphism of function application.

Lemma 4.3. $Conc_\alpha \circ Abs_\alpha \supseteq id_{\mathbf{P}D_\alpha}$.

Proof. Let $S \in \mathbf{P}D_\alpha$. $Conc_\alpha(Abs_\alpha(S)) = \biguplus\{T \mid Abs_\alpha(T) \leqslant Abs_\alpha(S)\}$ and $Abs_\alpha(S) \leqslant Abs_\alpha(S)$, so $S \in \{T \mid Abs_\alpha(T) \leqslant Abs_\alpha(S)\}$, hence the result. □

Proposition 4.4. *If $f \in D_{\alpha\to\beta}$, then $abs_{\alpha\to\beta}(f) \circ abs_\alpha \geqslant abs_\beta \circ f$ (or in terms of elements, if $s \in D_\alpha$, then $abs_{\alpha\to\beta}(f)(abs_\alpha(s)) \geqslant abs_\beta(f(s))$).*

Proof.

$$
\begin{aligned}
abs_{\alpha\to\beta}(f) \circ abs_\alpha &= \biguplus \circ Abs_\beta \circ \mathbf{P}f \circ Conc_\alpha \circ \{\cdot\} \circ abs_\alpha \\
&= \biguplus \circ Abs_\beta \circ \mathbf{P}f \circ Conc_\alpha \circ Abs_\alpha \circ \{\cdot\} \quad \text{(P6)} \\
&\geqslant \biguplus \circ Abs_\beta \circ \mathbf{P}f \circ \{\cdot\} \quad \text{Lemma 4.3} \\
&= \biguplus \circ \mathbf{P}abs_\beta \circ \mathbf{P}f \circ \{\cdot\} \\
&= \biguplus \circ \mathbf{P}(abs_\beta \circ f) \circ \{\cdot\} \quad \text{(P3)} \\
&= \biguplus \circ \{\cdot\} \circ abs_\beta \circ f \quad \text{(P6)} \\
&= abs_\beta \circ f. \qquad \square
\end{aligned}
$$

We are now in a position to be able to prove the safety of our abstraction and concretisation maps.

Proposition 4.5. *The abstraction and concretisation maps satisfy the safety criterion.*

That is,

$$\mathbf{P}f \subseteq Conc_\beta \circ \mathbf{P}(abs_{\alpha\to\beta}(f)) \circ Abs_\alpha.$$

Proof.

$$\begin{aligned}
\mathbf{P}(abs_{\alpha\to\beta}(f)) \circ Abs_\alpha &= \mathbf{P}(abs_{\alpha\to\beta}(f)) \circ \mathbf{P}abs_\alpha \\
&= \mathbf{P}(abs_{\alpha\to\beta}(f) \circ abs_\alpha) \qquad \text{(P3)} \\
&\geqslant \mathbf{P}(abs_\beta \circ f) \\
&\quad \text{Proposition 4.4 and } \mathbf{P} \text{ is locally monotone} \\
&= \mathbf{P}abs_\beta \circ \mathbf{P}f \qquad \text{(P3)} \\
&= Abs_\beta \circ \mathbf{P}f.
\end{aligned}$$

Now we can take $Conc_\beta$ of both sides and write $\geqslant$ as $\supseteq$ to obtain

$$\begin{aligned}
Conc_\beta \circ \mathbf{P}(abs_{\alpha\to\beta}(f)) \circ Abs_\alpha &\supseteq Conc_\beta \circ Abs_\beta \circ \mathbf{P}f \\
&\supseteq \mathbf{P}f \qquad \text{Lemma 4.3.} \quad \square
\end{aligned}$$

The next two lemmata allow us to see what happens when we abstract and concretise the least element of various types.

Lemma 4.6. (i) $abs_\alpha(\bot_\alpha) = \bot_{\tilde{\alpha}}$ *and* $Abs_\alpha(\{\bot_\alpha\}) = \{\bot_{\tilde{\alpha}}\}$.
(ii) *If* $f \in D_\alpha$, $f \neq \bot_\alpha$, *then* $abs_\alpha(f) \neq \bot_{\tilde{\alpha}}$ *and* $Abs_\alpha(\{f\}) \neq \{\bot_{\tilde{\alpha}}\}$.

Proof. (i) Since abs_α is onto, something maps to $\bot_{\tilde{\alpha}}$, so by monotonicity, $\bot_\alpha$ does. Similarly for Abs_α.

(ii) If $f \in D_A$, $f \neq \bot_A$, then $abs_A(f) \neq 0 = \bot_2$, and similarly $Abs_A(\{f\}) \neq \{\bot_2\}$. Supposing it is true for β, we prove it for $\alpha \to \beta$. $Abs_{\alpha\to\beta}(f) = \lambda x^{\tilde{\alpha}} \bigsqcup Abs_\beta(\mathbf{P}f(Conc_\alpha(\{x\})))$. Since $f \neq \bot_{\alpha\to\beta}$, there is some $a \in D_\alpha$ such that $f(a) \neq \bot_\beta$, and so

$$\begin{aligned}
\bigsqcup Abs_\beta(\mathbf{P}f(Conc_\alpha(\{abs_\alpha(a)\}))) &= \bigsqcup Abs_\beta(\mathbf{P}f(Conc_\alpha(Abs_\alpha(\{a\})))) \qquad \text{(P6)} \\
&\geqslant \bigsqcup Abs_\beta(\mathbf{P}f(\{a\})) \qquad \text{Lemma 3} \\
&= \bigsqcup Abs_\beta(\{f(a)\}) \qquad \text{(P6)} \\
&\neq \bot_{\tilde{\beta}} \qquad \text{by inductive hypothesis.}
\end{aligned}$$

Hence $abs_{\alpha\to\beta}(f) \neq \bot_{\tilde{\alpha}\to\tilde{\beta}}$. Again $Abs_{\alpha\to\beta}(\{f\}) \neq \{\bot_{\tilde{\alpha}\to\tilde{\beta}}\}$ since $Abs_{\alpha\to\beta} = \mathbf{P}abs_{\alpha\to\beta}$. $\square$

Lemma 4.7. $Conc_\alpha(\{\bot_{\tilde{\alpha}}\}) = \{\bot_\alpha\}$.

Proof.

$$\begin{aligned} Conc_\alpha(\{\perp_{\bar\alpha}\}) &= \biguplus\{T \mid Abs_\alpha(T) \leqslant \{\perp_{\bar\alpha}\}\} \\ &= \biguplus\{T \mid Abs_\alpha(T) = \{\perp_{\bar\alpha}\}\} \\ &= \biguplus\{\{\perp_\alpha\}\} \quad \text{Lemma 4.6} \\ &= \{\perp_\alpha\}. \end{aligned}$$

□

The following proposition shows that $abs_{\alpha\to\beta}$ is completely accurate as regards strictness information. Use of this proposition simplifies the ensuing development. It was suggested to us by Simon Peyton Jones.

Proposition 4.8. $(abs_{\alpha\to\beta}(f))\perp_{\bar\alpha} = \perp_{\bar\beta}$ *if and only if* $f(\perp_\alpha) = \perp_\beta$.

Proof. (only if)

$$\begin{aligned} \perp_{\bar\beta} &= abs_{\alpha\to\beta}(f)(\perp_{\bar\alpha}) \\ &= abs_{\alpha\to\beta}(f)(abs_\alpha(\perp_\alpha)) \quad \text{Lemma 4.6} \\ &\geqslant abs_\beta(f(\perp_\alpha)) \quad \text{Proposition 4.4.} \end{aligned}$$

So $abs_\beta(f(\perp_\alpha)) = \perp_{\bar\beta}$ and thus, by Lemma 4.6, $f(\perp_\alpha) = \perp_\beta$.

(if)

$$\begin{aligned} (abs_{\alpha\to\beta}(f))(\perp_{\bar\alpha}) &= \bigsqcup Abs_\beta(\mathbf{P}f(Conc_\alpha(\{\perp_{\bar\alpha}\}))) \\ &= \bigsqcup Abs_\beta(\mathbf{P}f(\{\perp_\alpha\})) \quad \text{Lemma 4.7} \\ &= \bigsqcup \mathrm{Abs}_\beta(\{f(\perp_\alpha)\}) \quad \text{(P6)} \\ &= \bigsqcup Abs_\beta(\{\perp_\beta\}) \quad \text{since } f(\perp_\alpha) = \perp_\beta \\ &= \bigsqcup\{\perp_{\bar\beta}\} \quad \text{Lemma 4.6} \\ &= \perp_{\bar\beta} \quad \text{(P10).} \end{aligned}$$

□

We are now in the position where we can work out what the abstract interpretations of the predefined functions are. We use a slightly different form of $abs_{\alpha\to\beta}$, namely $abs_{\alpha\to\beta} = \lambda f^{\alpha\to\beta}.\lambda x^{\bar\alpha}.\bigsqcup \mathbf{P}(abs_\beta \circ f)(Conc_\alpha(\{x\}))$ which can be seen to be equivalent to the definition by the following derivation:

$$\begin{aligned} abs_{\alpha\to\beta} &= \lambda f^{\alpha\to\beta}.\Psi(Abs_\beta \circ \mathbf{P}f \circ Conc_\alpha) \\ &= \lambda f^{\alpha\to\beta}.\Psi(\mathbf{P}(abs_\beta \circ f) \circ Conc_\alpha) \\ &= \lambda f^{\alpha\to\beta}.\lambda x^{\bar\alpha}.\bigsqcup \mathbf{P}(abs_\beta \circ f)(Conc_\alpha(\{x\})). \end{aligned}$$

Lemma 4.9. *Denoting* A *by* A^1 *and* $A \to A^n$ *by* A^{n+1}, *if* $f \in A^{n+1}$, $f \neq \perp_{A^{n+1}}$, f *strict in all its arguments, then* $abs_{A^{n+1}}(f) = \lambda x_1^2 \cdots \lambda x_n^2.x_1$ **and** $\cdots$ **and** x_n.

Proof. We prove this by induction on n, with the base case being $n = 2$.

$$abs_{A\to A}(f) = \lambda x^2.\biguplus \mathbf{P}(abs_A \circ f)(Conc_A(\{x\})),$$

and so

$$\begin{aligned}
abs_{A\to A}(f)(0) &= \perp_2 \quad \text{Proposition 4.8,}\\
abs_{A\to A}(f)(1) &= \biguplus \mathbf{P}(abs_A \circ f)(Conc_A(\{1\}))\\
&= \biguplus \mathbf{P}(abs_A \circ f)(D_A)\\
&= \biguplus \{abs_A(f(a)) \mid a \in D_A\}^*\\
&= \biguplus \{0, 1\} \qquad \text{since } f \text{ is strict and not } \perp_{A\to A}\\
&= 1.
\end{aligned}$$

Therefore, by extensionality, $abs_{A\to A}(f) = \lambda x^2.x$. Now suppose it is true for all $n \leqslant k$, and prove it true for $n = k+1$.

$$abs_{A^{k+1}}(f) = \lambda x^2.\biguplus \mathbf{P}(abs_{A^k} \circ f)(Conc_A(\{x\})),$$

and so

$$\begin{aligned}
abs_{A^{k+1}}(f)(0) &= \lambda x_1^2 \cdots \lambda x_{k-1}^2.0 \quad \text{Proposition 4.8,}\\
abs_{A^{k+1}}(f)(1) &= \biguplus \mathbf{P}(abs_{A^k} \circ f)(Conc_A(\{1\}))\\
&= \biguplus \mathbf{P}(abs_{A^k} \circ f)(D_A)\\
&= \biguplus \{abs_{A^k}(f(a)) \mid a \in D_A\}^*\\
&= \biguplus \{\lambda x_1^2 \cdots \lambda x_{k-1}^2.0, \lambda x_1^2 \cdots \lambda x_{k-1}^2.x_1 \textbf{ and } \cdots \textbf{ and } x_{k-1}\}^* \quad (1)\\
&= \lambda x_1^2 \cdots \lambda x_{k-1}^2.x_1 \textbf{ and } \cdots \textbf{ and } x_{k-1}
\end{aligned}$$

and hence $abs_{A^{k+1}}(f) = \lambda x_1^2 \cdots \lambda x_k^2.x_1$ **and** $\cdots$ **and** x_k. The element $\lambda x_1^2 \cdots \lambda x_{k-1}^2.0$ is in the set marked by (1) because f is strict, $\perp_A \in D_A$, and so $abs_{A^k}(f(\perp_A)) = abs_{A^k}(\lambda x_1^2 \cdots \lambda x_{k-1}^2.\perp_A)$ (since f is strict) $\in \{abs_{A^k}(f(a)) \mid a \in D_A\}$, and by Lemma 4.6, $abs_{A^k}(\lambda x_1^2 \cdots \lambda x_{k-1}^2.\perp_A) = \lambda x_1^2 \cdots \lambda x_{k-1}^2.0$. The element $\lambda x_1^2 \cdots \lambda x_{k-1}^2.x_1$ **and** $\cdots$ **and** x_{k-1} is in the set by the induction hypothesis since $f(a)$ is strict. □

We see this means that functions such as '+' and '×' have abstract interpretation $\lambda x^2.\lambda y^2.x$ **and** y which is the same as in [7], except that we regard the functions as being curried.

For each type α we have the conditional if_α of type $if_\alpha : A \to \alpha \to \alpha \to \alpha$. In finding the abstract interpretation for the conditional, we will make use of the following lemma:

Lemma 4.10. *For any continuous* $f \in D_\alpha \to B_\beta$, $y \in B_\alpha$, $(\mathbf{P}f \circ Conc_\alpha)(\{y\}) = \{f(t) \mid abs_\alpha(t) \leqslant y\}^*$.

Proof.

$$(\mathbf{P}f \circ Conc_\alpha)(\{y\}) = \{f(t) \mid t \in \biguplus \{T \mid Abs_\alpha(T) \leqslant \{y\}\}\}^*$$
$$= \{f(t) \mid t \in \biguplus \{T \mid \text{for all } t' \in T,\ abs_\alpha(t') \leqslant y\}\}^*$$
since $Abs_\alpha(T) \leqslant \{y\}$ if and only if for all $t \in T$, $abs_\alpha(t) \leqslant y$
$$= \{f(t) \mid t \in \bigcup \{T \mid \text{for all } t' \in T,\ abs_\alpha(t') \leqslant y\}\}^*$$
$$= \{f(t) \mid abs_\alpha(t) \leqslant y\}^*. \quad \square$$

Lemma 4.11.

$$abs_{A\to\alpha\to\alpha\to\alpha}(if_\alpha) = \lambda x^2.\lambda y^{\bar\alpha}.\lambda z^{\bar\alpha}.x\ \mathbf{and}_{\bar\alpha} \bigsqcup \{y, z\}$$

where $\mathbf{and}_{\bar\alpha} : 2 \to B_\alpha \to B_\alpha$ *is defined by*

$$0\ \mathbf{and}_{\bar\alpha} e = \bot_{\bar\alpha}, \qquad 1\ \mathbf{and}_{\bar\alpha} e = e.$$

Proof.

$$abs_{A\to\alpha\to\alpha\to\alpha}(if_\alpha) = \lambda x^2.\bigsqcup \mathbf{P}(abs_{\alpha\to\alpha\to\alpha} \circ if_\alpha)(Conc_A(\{x\})),$$
$$abs_{A\to\alpha\to\alpha\to\alpha}(if_\alpha)(0) = \bot_{\bar\alpha\to\bar\alpha\to\bar\alpha} \quad \text{Proposition 4.8,}$$
$$abs_{A\to\alpha\to\alpha\to\alpha}(if_\alpha)(1) = \bigsqcup \mathbf{P}(abs_{\alpha\to\alpha\to\alpha} \circ if_\alpha)(Conc_A(\{1\}))$$
$$= \bigsqcup \{abs_{\alpha\to\alpha\to\alpha}(if_\alpha a) \mid a \in D_A\}^*.$$

Now,

$$abs_{\alpha\to\alpha\to\alpha}(if_\alpha a) = \lambda y^{\bar\alpha}.\bigsqcup \mathbf{P}(abs_{\alpha\to\alpha} \circ (if_\alpha a))(Conc_\alpha(\{y\}))$$
$$= \lambda y^{\bar\alpha}.\bigsqcup \{abs_{\alpha\to\alpha}(if_\alpha\ a\ a') \mid abs_\alpha(a') \leqslant y\}^* \qquad \text{Lemma 4.10.}$$

Similarly,

$$abs_{\alpha\to\alpha}(if\ a\ a') = \lambda z^{\bar\alpha}.\bigsqcup \{abs_\alpha(if_\alpha\ a\ a'\ a'') \mid abs_\alpha(a'') \leqslant z\}^*.$$

Thus

$$abs_{A\to\alpha\to\alpha\to\alpha}(if_\alpha)(1)$$
$$= \bigsqcup \{\lambda y^{\bar\alpha}.\bigsqcup \{\lambda z^{\bar\alpha}.\bigsqcup \{abs_\alpha(if_\alpha\ a\ a'\ a'') \mid abs_\alpha(a'') \leqslant z\}^* \mid$$
$$abs_\alpha(a') \leqslant y\}^* \mid a \in D_A\}^*$$
$$= \bigsqcup \{\lambda y^{\bar\alpha}.\bigsqcup \{\lambda z^{\bar\alpha}.\bigsqcup \{abs_\alpha(\bot_\alpha), abs_\alpha(a'), abs_\alpha(a'') \mid abs_\alpha(a'') \leqslant z\}^* \mid$$
$$abs_\alpha(a') \leqslant y\}^* \mid a \in D_A\}^*$$
$$= \bigsqcup \{\lambda y^{\bar\alpha}.\bigsqcup \{\lambda z^{\bar\alpha}.\bigsqcup \{\bot_{\bar\alpha}, y, z\}^*\}^*\}^*$$
$$= \bigsqcup \{\lambda y^{\bar\alpha}.\bigsqcup \{\lambda z^{\bar\alpha}.\bigsqcup \{y, z\}\}^*\}^*$$
$$= \bigsqcup \{\lambda y^{\bar\alpha}.\bigsqcup \{\lambda z^{\bar\alpha}.\bigsqcup \{y, z\}\}\}$$
$$= \lambda y^{\bar\alpha}.\lambda z^{\bar\alpha}.\bigsqcup \{y, z\} \quad \text{by two applications of (P10)}$$

and so $abs_{A\to\alpha\to\alpha\to\alpha}(if_\alpha) = \lambda x^2.\lambda y^{\bar\alpha}.\lambda z^{\bar\alpha}.x\ \mathbf{and}_{\bar\alpha} \bigsqcup \{y, z\}$. $\square$

This is the natural extension of the interpretation of the conditional over $A \to A \to A \to A$, for in this case $\mathbf{and}_{\bar{\alpha}}$ is just the ordinary **and** function, and $\sqcup$ is just the **or** function. For objects, y and z, of higher type, $\sqcup\{y, z\}$ is computed pointwise.

5. Relationship between textual abstraction and semantic abstraction

In Section 3, we showed how we would like to be able to find the abstract interpretation of expressions by just examining the program script, having some sort or 'textual' abstraction map. We now put this on a formal footing, by defining a textual abstraction map, *tabs*, and showing how it relates to our semantic abstraction maps, and so how we can claim that *tabs* is a safe abstraction mechanism.

A problem which was not really addressed in Section 3 was the occurrence of 'free' variables when doing an abstract interpretation of an expression. For example, in the function $\lambda x^A.4+((\lambda y^A.x+y)5)$, in the subexpression $(\lambda y^A.x+y)$, x is 'free'. However, by the time that we actually evaluate $\lambda y^A.x+y$, x will be bound in some environment. So it seems sensible that *tabs* should take a syntactic expression and an environment and produce an expression in the appropriate abstract domain.

As we have said, the abstraction maps work on semantic values, so we must give a semantics of program texts in the typed λ-calculus. First we present an abstract syntax for the typed λ-calculus:

$\mathbf{Exp} = c^\alpha$ -constants (such as 4, +, etc.)

$|x^\alpha$ -variables of type α

$|\lambda x^\alpha.\mathbf{Exp}$ -function in $\alpha \to \beta$ if $\mathbf{Exp}:\beta$

$|(\mathbf{Exp}_1\ \mathbf{Exp}_2)$-function application, result of type β if $\mathbf{Exp}_1:\alpha \to \beta$ and $\mathbf{Exp}_2:\alpha$

$|$*fix* **Exp** -fixed point definition, result of type α if **Exp** of type $\alpha \to \alpha$.

If we let Env denote the set of type-respecting mappings from variables to elements of D (i.e. environments), then define

$$sem : \mathbf{Exp} \to \mathrm{Env} \to D$$

by

$$sem[[c]]\sigma = \mathbf{K}([[c]]) \text{ where } \mathbf{K}: constants \to D,$$

$$sem[[x]]\sigma = \sigma([[x]]),$$

$$sem[[\lambda x^\alpha.e]]\sigma = \lambda y^\alpha.sem[[e]]\sigma[y^\alpha/x^\alpha],$$

$$sem[[(e_1 e_2)]]\sigma = (sem[[e_1]]\sigma)(sem[[e_2]]\sigma),$$

$$sem([[fix\ e]])\sigma = fix(sem\ [[e]]\sigma),$$

and we define our textual abstraction map as a non-standard semantics, similarly letting Env′ denote the set of type-respecting mappings from variables to elements of B,

$$tabs : \mathbf{Exp} \to \mathrm{Env}' \to B$$

by

$tabs[[c]]\rho = abs_\alpha(\mathbf{K}([[c]]))$ if c is a constant of type α,

$tabs[[x^\alpha]]\rho = \rho([[x^\alpha]])$,

$tabs[[\lambda x^\alpha.e]]\rho = \lambda x^{\bar{\alpha}}.tabs[[e]]\rho[x^{\bar{\alpha}}/x^\alpha]$,

$tabs[[(e_1 e_2)]]\rho = (tabs[[e_1]]\rho)(tabs[[e_2]]\rho)$,

$tabs[[fix\ e]]\rho = fix(tabs[[e]]\rho)$.

Clearly *tabs* defines a computable function (recall that the abstract domains are all finite). How can we state the correctness of this function? What we require is, given any expression $e : \alpha \to \beta$, if $(tabs[[e]](abs \circ \sigma))(\bot_{\bar{\alpha}}) = \bot_{\bar{\beta}}$ then e 'really does need its argument' (i.e. $(sem[[e]]\sigma)(\bot_\alpha) = \bot_\beta$). Before proving the result, the Soundness Theorem for Strictness Analysis, we need two technical lemmata. The first states a semi-homomorphic relationship between fixed points and abstraction. This is used in the second, which states that if the environments ρ and σ stand in the relationship $\rho \geqslant abs \circ \sigma$, then the *tabs* of an expression in the environment ρ is at least as defined as the abstraction of the semantics of the expression in the environment σ.

Lemma 5.1. $fix(abs_{\alpha\to\alpha}(f)) \geqslant abs_\alpha(fix(f))$.

Proof. Let h_i be the approximations to $fix(abs_{\alpha\to\alpha}(f))$ and f_i be the approximations to $fix(f)$. Then $h_0 = \bot_{\bar{\alpha}} = abs_\alpha(\bot_\alpha) = abs_\alpha(f_0)$. Assume that $h_i \geqslant abs_\alpha(f_i)$. Then

$$\begin{aligned} h_{i+1} &= (abs_{\alpha\to\alpha}(f))(h_i) \\ &\geqslant (abs_{\alpha\to\alpha}(f))(abs_\alpha(f_i)) && \text{induction hypothesis and monotonicity of } abs_{\alpha\to\alpha}(f) \\ &\geqslant abs_\alpha(f(f_i)) && \text{Proposition 4.4} \\ &= abs_\alpha(f_{i+1}). \end{aligned}$$

So $h_i \geqslant abs_\alpha(f_i)$ for all i. Taking the least upper bounds of both sides we obtain

$$\begin{aligned} fix(abs_{\alpha\to\alpha}(f)) &= \bigsqcup h_i \\ &\geqslant \bigsqcup \{abs_\alpha(f_i)\} \\ &= abs_\alpha(\bigsqcup \{f_i\}) && \text{since } abs_\alpha \text{ continuous, } \{f_i\} \text{ directed} \\ &= abs_\alpha(fix(f)). \quad \square \end{aligned}$$

Lemma 5.2. *If* $\rho \geqslant abs_\alpha \circ \sigma$ *and* $e:\alpha$, *then tabs* $[[e]]\rho \geqslant abs_\alpha(sem[[e]]\sigma)$.

Proof. We prove this by structural induction:

– constants:

$$tabs[[e]]\rho = abs_\alpha(sem[[e]]\sigma) \quad \text{definition}$$
$$\geqslant abs_\alpha(sem[[e]]\sigma),$$

– variables:

$$tabs[[x^\alpha]]\rho = \rho(x^\alpha)$$
$$\geqslant abs_\alpha(sem[[x^\alpha]]\sigma) \quad \text{by condition of lemma,}$$

– application:

$$tabs[[(e_1 e_2)]]\rho = (tabs[[e_1]]\rho)(tabs[[e_2]]\rho)$$
$$\geqslant (abs_{\alpha\to\beta}(sem[[e_1]]\sigma))(abs_\alpha(sem[[e_2]]\sigma))$$
induction hypothesis
$$\geqslant abs_\beta((sem[[e_1]]\sigma)(sem[[e_2]]\sigma)) \quad \text{Proposition 4.4}$$
$$= abs_\beta(sem[[(e_1 e_2)]]\sigma);$$

– λ-abstraction: $\lambda x^{\bar\alpha}.e$ where $(sem[[e]]\sigma) \in D_\beta$

$$abs_{\alpha\to\beta}(sem[[\lambda x^\alpha.e]]\sigma) = abs_{\alpha\to\beta}(\lambda y^\alpha.sem[[e]]\sigma[y^\alpha/x^\alpha])$$
$$= \lambda y^{\bar\alpha}.\bigsqcup \mathbf{P}(abs_\beta \circ (\lambda y^\alpha.sem[[e]]\sigma[y^\alpha/x^\alpha])(Conc_\alpha(\{y\}))$$
$$= \lambda y^{\bar\alpha}.\bigsqcup \{abs_\beta((\lambda y^\alpha.sem[[e]]\sigma[y^\alpha/x^\alpha])t) | abs_\alpha(t) \leqslant y\}^*$$
Lemma 4.10
$$= \lambda y^{\bar\alpha}.\bigsqcup \{abs_\beta(sem[[e]]\sigma[t/x^\alpha]) | abs_\alpha(t) \leqslant y\}^*,$$
$$tabs[[\lambda x^\alpha.e]]\rho = \lambda x^{\bar\alpha}.tabs[[e]]\rho[x^{\bar\alpha}/x^\alpha].$$

Now, $tabs[[\lambda x^\alpha.e]]\rho \geqslant abs_{\alpha\to\beta}(sem[[\lambda x^\alpha.e]]\sigma)$ if and only if $(tabs[[\lambda x^\alpha.e]]\rho)(s) \geqslant (abs_{\alpha\to\beta}(sem[[\lambda x^\alpha.e]]\sigma))(s)$ for all $s \in B_\alpha$. So let $s \in B_\alpha$ and apply the two expressions to it:

$$(abs_{\alpha\to\beta}(sem[[\lambda x^\alpha.e]]\sigma))(s) = \bigsqcup \{abs_\beta(sem[[e]]\sigma[t/x^\alpha]) | abs_\alpha(t) \leqslant s\}^*,$$
$$(tabs[[\lambda x^\alpha.e]]\rho)(s) = tabs[[e]]\rho[s/x^\alpha].$$

Now suppose that u is such that $abs_\alpha(u) \leqslant s$. Then $\rho[s/x^\alpha] \geqslant abs_\alpha \circ \sigma[u/x^\alpha]$, and so we have

$$abs_\beta(sem[[e]]\sigma[u/x^\alpha]) \leqslant tabs[[e]]\rho[s/x^\alpha]$$

by our induction hypothesis. So each element in the set

$\{abs_\beta(sem[[e]]\sigma[t/x^\alpha]) \mid abs_\alpha(t) \leqslant s\}$ approximates $(\text{tabs}[[\lambda x^\alpha.e]]\rho)(s)$, Thus

$$(tabs[[\lambda x^\alpha.e]]\rho)(s) \geqslant (abs_{\alpha\to\beta}(sem[[\lambda x^\alpha.e]]\sigma))(s)$$

and hence

$$tabs[[\lambda x^\alpha.e]]\rho \geqslant abs_{\alpha\to\beta}(sem[[\lambda x^\alpha.e]]\sigma);$$

– fixed points:

$$\begin{aligned} tabs[[fix\, e]]\rho &= fix(tabs[[e]]\rho) \\ &\geqslant fix(abs_{\alpha\to\alpha}(sem[[e]]\sigma)) \\ &\quad \text{induction hypothesis and monotonicity of } fix \\ &\geqslant abs_\alpha(fix(sem[[e]]\sigma)) \qquad \text{Lemma 5.1} \\ &= abs_\alpha(sem[[fix\, e]]\sigma). \quad \square \end{aligned}$$

We have the following corollary which shows that the relationship between ρ and σ is preserved by the non-standard and the standard semantics.

Corollary 5.3. *If* $\rho \geqslant abs_\alpha \circ \sigma$, *then* $\rho[(tabs[[e]]\rho)/x^\alpha] \geqslant abs_\alpha \circ \sigma[(sem[[e]]\sigma)/x^\alpha]$.

Proof. As ρ and $\rho[(tabs[[e]]\rho)/x^\alpha]$, and σ and $\sigma[(sem[[e]]\sigma)/x^\alpha]$ differ only in the values they give to x^α, this is true if and only if $tabs[[e]]\rho \geqslant abs_\alpha(sem[[e]]\sigma)$. This is just Lemma 5.2. $\square$

We now come to the main result of the paper.

Theorem 5.4. (Soundness Theorem for Strictness Analysis). *If* $f: \alpha \to \beta$ *and* $\rho \geqslant abs \circ \sigma$, *then* $(tabs[[f]]\rho)(\perp_{\bar\alpha}) = \perp_{\bar\beta}$ *implies* $(sem[[f]]\sigma)(\perp_\alpha) = \perp_\beta$.

Proof. Suppose $f: \alpha \to \beta$.

$$(tabs[[f]]\rho)(\perp_{\bar\alpha}) = \perp_{\bar\beta}$$

implies

$$abs_{\alpha\to\beta}(sem[[f]]\sigma)(\perp_{\bar\alpha}) = \perp_{\bar\beta} \quad \text{Lemma 5.2.} \quad \square$$

implies

$$(sem[[f]]\sigma)(\perp_\alpha) = \perp_\beta \quad \text{Proposition 4.8.} \quad \square$$

Thus if $(tabs[[f]]\rho)(\perp_{\bar\alpha}) = \perp_{\bar\beta}$ then we can safely deduce that f needs its argument. Finally, we return to the example given in Section 3:

$$g = \lambda f^{A\to A}.\lambda x^A.\lambda y^A.+\, x f(y),$$

$$h = \lambda u^A.g(\lambda v^A.\times\, v\, u)5\,4$$

(where × and + are now written as curried functions in prefix form).

Letting ρ be some initial environment, we work out

$$
\begin{aligned}
&tabs[[g]]\rho \\
&= \lambda f^{2\to 2}.tabs[[\lambda x^A.\lambda y^A.+xf(y)]]\rho[f^{2\to 2}/f^{A\to A}] \\
&= \lambda f^{2\to 2}.\lambda x^2.tabs[[\lambda y^A.+xf(y)]]\rho[f^{2\to 2}/f^{A\to A}, x^2/x^A] \\
&= \lambda f^{2\to 2}.\lambda x^2.\lambda y^2.tabs[[+xf(y)]]\rho', \text{ where } \rho' = \rho[f^{2\to 2}/f^{A\to A}, x^2/x^A, y^2/y^A] \\
&= \lambda f^{2\to 2}.\lambda x^2.\lambda y^2.(tabs[[+x]]\rho')(tabs[[f(y)]]\rho') \\
&= \lambda f^{2\to 2}.\lambda x^2.\lambda y^2.((tabs[[+]]\rho')(tabs[[x^A]]\rho')) \\
&\quad ((tabs[[f^{A\to A}]]\rho')(tabs[[y^A]]\rho')) \\
&= \lambda f^{2\to 2}.\lambda x^2.\lambda y^2.((\lambda u^2.\lambda v^2.u \textbf{ and } v)x^2)(f^{2\to 2}y^2) \\
&= \lambda f^{2\to 2}.\lambda x^2.\lambda y^2.x \textbf{ and } f(y).
\end{aligned}
$$

Similarly, we find that $tabs[[h]]\rho[\lambda f^{2\to 2}.\lambda x^2.\lambda y^2.x \textbf{ and } f(y)/g] = \lambda u^2.u$. Thus after the first abstraction pass, the environment, which we will call ρ for simplicity, will contain abstract versions of the functions g and h. We can now start to annotate the apply nodes. Considering the definition of g, we have

$$tabs[[+]]\rho = \lambda u^2.\lambda u^2.u \textbf{ and } v$$

and

$$
\begin{aligned}
tabs[[+]]\rho\ 0 &= (\lambda u^2.\lambda v^2.u \textbf{ and } v)\ 0 \\
&= \lambda v^2.0 \textbf{ and } v \\
&= \lambda \dot{v}^2.0,
\end{aligned}
$$

thus '+' needs its first argument. Then

$$
\begin{aligned}
tabs[[+x]]\rho &= (tabs[[+]]\rho)(tabs[[x]]\rho) \\
&= (\lambda u^2.\lambda v^2.u \textbf{ and } v)\rho(x) \\
&= \lambda v^2.\rho(x) \textbf{ and } v
\end{aligned}
$$

and $(\lambda v^2.\rho(x) \textbf{ and } v)0 = \rho(x) \textbf{ and } 0 = 0$, thus $(+x)$ needs its argument. The final apply node in the graph of g is (fy), and for this we have

$$
\begin{aligned}
tabs[[f(y)]]\rho &= (tabs[[f]]\rho)(tabs[[y]]\rho) \\
&= \rho(f)(\rho(y)).
\end{aligned}
$$

We do not know the value of f because it may change from application to application of g. The only safe value to use is the top element of the appropriate domain. Thus

$$
\begin{aligned}
tabs[[f(y)]]\rho &= (\lambda x^2.1)\rho(y) \\
&= 1.
\end{aligned}
$$

This tells us that we cannot infer anything about the application $f(y)$.

Considering now the apply nodes in h, we proceed in a similar fashion and find that g needs its first parameter.

Turning to the expression '$\times v\,u$' in the λ-expression; this is annotated in the same way as the two '+' application nodes of g. We also find that $g(\lambda v^A.\times\, v\,u)$ needs its first parameter. The analysis for $g(\lambda v^A.\times\, v\,u)\ 5$ proceeds as follows:

$$\begin{aligned} tabs[[g(\lambda v^A.\times v\,u)5]]\rho &= (tabs[[g]]\rho(tabs[[\lambda v^A.\times v\,u]]\rho)\ 1 \\ &= (\lambda f^{2\to 2}.\lambda x^2.\lambda y^2.x \textbf{ and} \\ &\quad f(y))(tabs[[\lambda v^A.\times v\,u]]\rho)\ 1 \\ &= (\lambda x^2.\lambda y^2.x \textbf{ and } (tabs[[\lambda v^A.\times v\,u]]\rho)(y))\ 1 \\ &= \lambda y^2.(\lambda v^2.v \textbf{ and } \rho(u))(y) \\ &= \lambda y^2.y \textbf{ and } \rho(u). \end{aligned}$$

Then as we can see that

$$\begin{aligned} (\lambda y^2.y \textbf{ and } \rho(u))0 &= 0 \textbf{ and } \rho(u) \\ &= 0, \end{aligned}$$

we can infer that $g(\lambda v^A.\times v\,u)\ 5$ needs its parameter.

6. Conclusions

We have exhibited a method for strictness analysis of a typed language incorporating higher-order functions over flat base domains, which has a sound theoretical foundation. This is an expression based interpretation, which works on program texts, allowing us to decide at each apply node whether an argument is definitely needed or not. The pragmatics of implementing the textual abstraction map in a compiler are currently being investigated by the authors. Some work on the implementation of strictness analysis has been reported in [3].

Much of the material is independent of the interpretation of the base domains, so it should be fairly easy to extend the framework to handle other abstract interpretations. For example, if we were to define strictness analysis for non-flat base domains, then we may be able to extend this to higher order functions by using the structure presented in this paper. (The trivial approach of setting the abstract interpretation of *cons* to $\lambda x.\lambda y.1$ is already supported here.)

Another, highly desirable extension is to a language allowing polymorphic functions [6]. This topic is treated in a paper by one of the authors [1].

Acknowledgment

A number of people have assisted in the development of the ideas presented in this paper. The first author had many useful discussions with David Bevan during the early development of this work. We thank Pete Harrison and Simon Peyton Jones for their careful reading of this paper, and for their helpful comments. Simon in particular made many excellent suggestions and constructive criticisms which have materially improved the presentation.

Geoffrey Burn is partly funded by ESPRIT project 415: Parallel Architectures and Languages for AIP: A VLSI Directed Approach.

References

[1]S. Abramsky, Strictness analysis and polymorphic invariance, *Proc. Workshop on Programs as Data Objects*, Lecture Notes in Computer Science **217** (Springer, Berlin, 1986) 1-23.

[2] M.A. Arbib and E.G. Manes, *Arrows, Structures and Functors: The Categorical Imperative* (Academic Press, New York, 1975).

[3] C. Clack and S.L. Peyton Jones, Generating parallelism from strictness analysis, Department of Computer Science, University College London, Internal Note 1679,

[4] G. Gierz, K.H. Hofmann, K. Keimel, J.D. Lawson, M. Mislove and D.S. Scott, *A Compendium of Continuous Lattices* (Springer, Berlin, 1980).

[5] M. Hennessy and G.D. Plothin, Full abstraction for a simple parallel programming language, *Proc. Mathematical Foundations in Computer Science 1979*, Lecture Notes in Computer Science **74** (Springer, Berlin, 1979) 108-120.

[6] R. Milner, A theory of type polymorphism in programming, *J. Comput. System Sci.* **17** (1978) 348-375.

[7] A. Mycroft, Abstract interpretation and optimising transformations for applicative programs, PhD. Thesis, University of Edinburgh, 1981.

[8] A. Mycroft and F. Nielson, Strong abstract interpretation using power domains (Extended Abstract), *Proc. 10th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science **154** (Springer, Berlin, 1983) 536-547.

[9] G.D. Plotkin, A powerdomain construction, *SIAM J. Comput.* **5**(3) (1976) 452-487.

[10] G.D. Plotkin, A power domain for countable non-determinism (Extended Abstract), *Proc. 9th International Colloquium on Automata, Langauges and Programming*, Lecture Notes in Computer Science **140** (Springer, Berlin, 1982) 418-428.

[11] D. Scott *Lectures on a Mathematical Theory of Computation*, Tech. Monograph PRG-19, Oxford Univ. Computing Lab., Programming Research Group, 1981.

[12] D. Scott, Domains for denotational semantics, *Proc. 9th International Colloquium on Automata, Languages and Programminmg*, Lecture Notes in Computer Science **140** (Springer, Berlin, 1982) 577-613.

[13] J.E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory* (MIT Press, Cambridge, MA, 1977).

[14] C.P. Wadsworth, Semantics and pragmatics of the lambda calculus (Chapter 4), PhD Thesis, University of Oxford, 1971.