Abstract Interpretation and the Parallel Evaluation of Functional Languages.

by

Geoffrey Livingston Burn

August, 1986.

A thesis submitted for the degree of Doctor of Philosophy of the University of London and for the Diploma of Membership of the Imperial College.

Department of Computing
Imperial College
University of London

Abstract

Unrestricted parallel evaluation of functional programs overloads the resources of a machine by evaluating expressions whose values are not needed for producing the result of a program. We present a semantic criterion which restricts the expressions which are evaluated to those which would eventually have been evaluated using a lazy evaluator.

Usually a programming language is designed with one particular meaning, or interpretation, for each of the constructs. By giving a different, or abstract, interpretation which captures some property of interest, we are able to answer questions about programs without running them. Such interpretations are often used by a compiler to optimise the execution of programs.

We develop a framework for the abstract interpretation of functional languages. For the first time we present a framework which supports all the features of functional languages excepting general recursive type definitions. It is applied to give an interpretation which specifies the definedness of a function in terms of the definedness of its arguments.

Given that the semantic criterion allows a certain amount of evaluation of a function application, we are able to use the definedness interpretation to determine how much evaluation of the arguments is permitted. Previous uses of similar interpretations have ignored much of the information that is available. By asking how much information we have to give about arguments in order to produce a certain amount of information about a function application, we are able to take into account the dynamic context of an expression. This simple change in the way we use the abstract interpretation means we can determine many more sources of parallel evaluation. Evaluation of the arguments can then proceed in parallel with the evaluation of the application. The complete analysis may be implemented as one of the passes in a compiler, so that programs can be automatically annotated with the parallelism information.

Acknowledgements

I would like to thank the following people and institutions for making this work possible.

This work was completed while working for GEC Research Ltd at the Hirst Research Centre, who supported me financially by paying my College fees and making it possible to buy the books and obtain the references that I needed. My work was completed for ESPRIT Project 415, entitled "Parallel Architectures and Languages for AIP - A VLSI Directed Approach", and the project was partially funded by the EEC under the ESPRIT scheme. Of the people who have worked with me at Hirst, both past and present, I would like to especially thank Martin Evans who originally encouraged me to do a PhD, and Ray Offen, who was my Lab Manager, and who, besides arranging for Hirst to support me in my research, was my external supervisor and provided much encouragement. Peter Watkins took over my external supervision when Ray left GEC. I have had many technical discussions with David Bevan and Rajeev Karia and David's mathematical skill has been especially helpful.

At Imperial College, Samson Abramsky has been most helpful throughout the development of this work, both in giving encouragement and for his very valuable technical knowledge of the mathematical foundations upon which this thesis has been built. I thank Simon Peyton Jones of University College with whom I have had many fruitful discussions, and who kept on asking questions which were hard to answer.

Most of all I would like to thank my supervisor, Chris Hankin, of Imperial College. He has spent many long hours with me as we thrashed out ideas together, and has been especially helpful with his pointers into the literature and his reading of all that I have written. Without his help and encouragement this work would never have been completed.

Contents

List of Tables

Chapter 1

Introduction

1.1. Functional Languages, Evaluation Strategies and Semantics.

Functional languages of the type we will be considering in this thesis are syntactically sugared versions of the typed λ-calculus with constants. We will formally introduce the language in section 1.5. The λ-calculus can be viewed from two different angles. It can be regarded as a term-rewriting system [Church 1941], [Curry and Feys 1958], [Plotkin 1977], [Barendregt 1984], [Klop 1985] or as a notation for mathematical objects and operations over them [Scott 1981, 1982], [Milne and Strachey 1976], [Stoy 1977]. The former view is used in implementations of functional languages whilst the latter most often finds use in giving semantics to programming languages. This thesis explores the junction between the two views, giving a semantic condition which allows the evaluation strategy to be changed while retaining the correct semantics. By developing and applying a framework for abstract interpretation we are able to give a method for statically checking the semantic condition during compilation. In this and the following two sections, we will discuss the two views of the λ-calculus and their relationship.

If we were to regard the typed λ-calculus with constants as a term rewriting system, then we have to give rewrite rules for the λ-calculus and the constants. For the λ-calculus, the rewrite rules are usually called α-, β-, and η-conversion, while for the constants we have the normal reduction rules. For example, the rule for an expression

$$+\ e_1\ e_2$$

would (recursively) rewrite e_1 and e_2 to integers and then rewrite the whole expression to their sum.

Care must be taken with the constructors of data objects which stand for potentially infinite pieces of data, for example lists. Consider the function

$$ints\_from(n) = cons(n, ints\_from(n+1)).$$

If we chose a rewrite rule for *cons* which (recursively) rewrote both arguments to *cons*, then an application

$$ints\_from(0)$$

would be successively rewritten as shown below :

$$\begin{aligned} ints\_from(0) &\rightarrow cons(0, ints\_from(0+1)) \\ &\rightarrow cons(0, cons(1, ints\_from(1+1))) \\ &\rightarrow cons(0, cons(1, cons(2, ints\_from(2+1)))) \\ &\rightarrow \ldots \end{aligned}$$

This is just producing the infinite list of integers, and the computation will never terminate. While it may be acceptable to do this to the top-level expression, in an application

$$hd(ints\_from(0))$$

one would like the value 0 to be returned. Thus a rewrite rule which caused an infinite computation to first calculate the entire list *ints_from*(0) before being able to take the head of it is unacceptable. The only way to ensure such a situation does not occur is to have no rewrite rule for *cons*, so that we do not initiate potentially infinite computations by evaluating one of the arguments to *cons*. This was the key observation of the papers [Henderson and Morris 1976] and [Friedman and Wise 1976], and allows the programmer to use infinite lists which are a very powerful programming paradigm [Hughes 1984].

We are thus able to say that an evaluator will evaluate an expression as far as head normal form (†), defined by

---

(†) Functional language evaluators often evaluate functions only as far as *weak head normal form*, which differs from head normal form only in the way that it treats functions [Peyton Jones 1986]. An expression which stands for a function is in weak head normal form when there is a λ at the top-level. By only evaluating expressions to weak head normal form we remove the problem of having to rename variables within expressions, making implementation a lot easier [Peyton Jones 1986]. Consider the expression

$$\lambda x.((\lambda y.\lambda x.y)x)$$

which is weak head normal form but not head normal form. To reduce this to head normal form requires a renaming of one of the bound variables to obtain $\lambda x.\lambda z.x$. This is discussed further in section 5.7.

As well, the usual way the implementation of a functional language works is to use an evaluator which evaluates things to weak head normal form and then uses a strict print. That is, if the answer is a list object, it will force the evaluation of the head of the list, print it, and then repeat the process on the tail of the list until it reaches the end of the list (if it exists!). Unfortunately, this no longer ensures that the "expression left at the end of the computation" has the same semantics as the original expression for the quite simple reason that this evaluation strategy now falls into any black holes, and so is not safe. Consider what happens when the answer is an expression which represents $cons(1, cons(\bot, cons(5, nil)))$. The evaluator/print routine will force the evaluation of the first element of the list and print out

$$cons(1, cons($$

but unfortunately that is as far as it will get, for the evaluation of the next element of the list initiates a non-terminating computation. As this gives more information than just printing out

Definition 1.1.1

A λ-expression is in *head normal form* if and only if it is of the form

$$\lambda x_1^{\sigma_1}. \cdots \lambda x_n^{\sigma_n}.(v\ M_1\ \cdots\ M_m)$$

where

(i) $n, m \geq 0$,

(ii) v is a variable (i.e. $x_i^{\sigma_i}$ for some i) or a constant, and

(iii) $(v\ M_1\ \cdots\ M_m)$ is not a redex.

□

So far we have only been discussing the evaluation of an expression, but not saying how we choose which expression to evaluate. For example, in the expression

$$+\ (\times\ 3\ 4)\ (\times\ 5\ 6)$$

there are two reducible expression - $(\times\ 3\ 4)$ and $(\times\ 5\ 6)$. A rule for choosing which set of expressions to reduce next is called an *evaluation strategy* (or *evaluation mechanism* or *computation rule*).

Given several evaluation strategies, it is natural to ask whether they compute the same answer. In the case of the λ-calculus, it is provable that all evaluation strategies give the same answer *if they terminate* [Barendregt 1984]. From a semantic viewpoint, we would like to ensure that the evaluation mechanisms give lazy semantics to a program. Thus we would like to ensure that a computation terminated if the semantics of the original expression was not bottom (undefined), and that the expression left had the correct semantics. A computation then would not terminate only if the semantics of the original expression was bottom.

It can be shown that there are several evaluation strategies for the typed λ-calculus with constants which preserve the semantics of the original expression and terminate when the semantics of the expression is non-bottom [Plotkin 1977], [Klop 1985]. Below we will discuss two such strategies, and in the next section, discuss their potential for parallelism.

The *full substitution rule* selects every redex for reduction at each computation step. For example [Downey and Sethi 1976], if we assume that g and h are constants which are irreducible, and that f is a user-defined function defined by (i.e. has the rewrite rule) :

---

cons(*suspended_expression*, *suspended_expression*), then this is probably acceptable at the top level.

$$f(x) = g(fhfx.hx)$$

then the computation using the full substitution rule would start as follows :

$$f(x) \rightarrow g(fhfx.hx)$$

$$\rightarrow g(g(fhfhg(fhfx.hx).hhg(fhfx.hx)).hx)$$

$$\rightarrow \ldots$$

Another evaluation mechanism which is correct with respect to the mathematical semantics is known variously as the *call-by-name* or *left-most outer-most* or *normal order* strategy. The different names have arisen because it has appeared in several different contexts; call-by-name because of its use in programming languages as a parameter passing mechanism, where the values of arguments are passed as unevaluated expressions and evaluated only when needed [Naur 1963]; left-most outer-most because it reduces the left-most outer-most redex; and normal order from the λ-calculus background because it is the strategy which is guaranteed to find a *normal form* if it exists [Barendregt 1984].

We can illustrate call-by-name by using the same example that was used for illustrating the full substitution rule :

$$f(x) \rightarrow g(fhfx.hx)$$

$$\rightarrow g(g(fhfhfx.hhfx).hx)$$

$$\rightarrow \ldots$$

Call-by-name can be made computationally more efficient by noting that if an expression is substituted into another expression in several places, then it may end up being evaluated several times. By arranging that once the expression is reduced all other copies of the expression share the reduced form, computational effort is saved. This is often termed *call-by-need*. When call-by-need is combined with the rule that expressions are only reduced to head normal form, we have an evaluation strategy which is usually called *lazy evaluation*. It is named this because the evaluator is lazy, not doing any reduction of an expression until it is forced to do so to produce a result.

1.2. Evaluation Strategies and Potential for Parallelism.

Initiating a non-terminating computation in a sequential machine is fatal, because it means that the whole computation is undefined. In a parallel machine, because there is more than one processor, computation may be able to proceed even if there are some infinite computations taking place. However, we will argue that it is still not sensible to evaluate every possible redex in an expression in parallel because this could swamp the resources of a parallel machine. In the third section we will use this intuition to develop a semantic criterion for allowing parallel evaluation.

In the previous section we introduced two evaluation strategies which gave the correct semantics of programs. At first sight, the full substitution rule seems to be the best candidate for implementation on a parallel machine, for we can perform all of the reductions at each step in parallel.

Unfortunately, this computation rule has many disadvantages which render it impractical. Consider the conditional expression :

if condition then e_1 *else* e_2.

The full substitution rule will cause some evaluation to be done on all three of the expressions *condition*, e_1 and e_2. However, we note that only one of e_1 and e_2 will ever be *needed* in the sense that the value of one of them must be determined (according to the truth of the *condition*) to obtain the value of the expression, and so any evaluation of the expression which is not needed is "wasted"(†). While this may not seem a big disadvantage because we have saved some time on the evaluation of the expression which is needed, we must be careful not to overload the finite resources of our machine. Thus, "wasted" means more than just wasting time computing something which is not needed, but also means withholding resources from the evaluation of an expression which we know will need to be evaluated, for example, the *condition* in the above.

A possible way to bypass the problem of wasting resources evaluating expressions which are not needed is to prioritise tasks so that a higher priority is given to expressions whose values we know are needed than for expressions which are more "speculative". Thus, again using the example of the conditional, the evaluation of the *condition* would have higher priority than the evaluation of either e_1 or e_2. If we do this, then we

(†) In fact, we often use the conditional for exactly this reason - to stop the evaluator from evaluating an expression unless it needs it. Consider the function :

factorial(n) = *if* $n = 0$ *then* 1 *else* n**factorial*$(n-1)$

where one does not wish to evaluate *factorial*$(n-1)$ when $n = 0$.

introduce further problems, for the priority of tasks may change over time. For example, if the *condition* was to evaluate to *true* in the above example, then e_1 would become a "vital" task, because we now need its value.

Perhaps more serious than the problem of having to introduce and manage a priority system is what must be done about the computations of expressions whose values we discover are not needed. If we are to make maximum use of the finite resources of our machine, then we would like to stop the evaluation of unneeded expressions. The problem here is that the evaluation of the unwanted expression has generally initiated the evaluation of subexpressions, which also must be stopped, and so on. While this may be possible, we can imagine the situation where the spawning of subtasks proceeds faster than messages can be sent to stop tasks.

One final nail that we will drive into the coffin of the full substitution rule is that it requires more computation steps than other methods [Vuillemin 1974].

If we look carefully at the above discussion, we can see that the source of the problem is that the full substitution rule evaluates expressions whose values are never needed.

Call-by-name is an evaluation mechanism which ensures that we only evaluate expressions when they are needed. However, it has the unfortunate side-effect that only one reduction step is done at a time, and so is a sequential evaluation strategy with no potential for parallelism - hardly useful for a parallel machine! (†) The rest of this thesis allows us to get around this problem - ensuring that we do no more evaluation than is needed, but at the same time allowing parallel evaluation.

1.3. A Safe Evaluation Strategy Which Allows Parallel Evaluation.

Because a sequential evaluator only evaluates one expression at a time, and call-by-name only evaluates expressions whose values are needed, then call-by-name is a sensible evaluation strategy for a sequential machine. However, call-by-name is overly pessimistic about needed expressions when it comes to a parallel architecture. It is intuitively obvious that the values of both e_1 and e_2 are needed in the expression

(†) An astute reader may notice that when the top-level redex is something like $+\ e_1\ e_2$ that call-by-name would evaluate firstly e_1 and then e_2, whereas it is perfectly safe to evaluate both e_1 and e_2 in parallel because the function needs both of the values to be able to return a value. Unfortunately the parallelism generated by such *strict* operators is not sufficient to warrant the building of a parallel machine [Peyton Jones 1984]. However, the method of abstract interpretation can be seen to generalise this idea to finding out similar information about user-defined functions.

$+\ e_1\ e_2$

and so can be evaluated in parallel.

In this section we will give our semantic criterion for changing the evaluation strategy from call-by-name to strategies which involve the parallel evaluation of some of the arguments to functions. Firstly however, we must define some terms.

Suppose we have a language Exp. We form the language Exp_∞ by taking the completion of the language obtained by adding a constant Ω_σ for each type σ to Exp. The constants Ω_σ represent computations which for all finite numbers of steps, and hence in the limit, return no information (or value). For any type σ, Ω_σ is a formal bottom element, and in any interpretation of the type σ will have the bottom element as its interpretation. Completion allows infinite expressions. The semantics of Exp is extended to Exp_∞ in the usual way [Guessarian 1981] so that we can give the semantics of an expression in Exp_∞. Then we can make the following definition :

Definition 1.3.1:

A function

$$\xi : Exp \rightarrow Exp_\infty .$$

is an *evaluator* if it preserves the semantics of e. That is, if $e \in Exp$, then the semantics of e and $\xi(e)$ are identical.

□

It is worth noting why we have included all the things we have in the definition of Exp_∞. In section 1.1 we introduced the idea of not evaluating the expressions which were arguments to *cons*. Thus it is not necessarily true that an evaluator will terminate with an expression containing no redices. The element Ω_σ is needed to represent computations which are trying to evaluate an expression which has semantics $\perp$, that is, a computation which is infinite but returns no information. It can appear embedded in a data structure. Finally we need to extend the language to include infinite expressions (if the language does not already include them) because we can evaluate expressions which represent infinite objects. As an example of this we gave the definition of the function

$$ints\_from(n) = cons(n, ints\_from(n+1))$$

in section 1.1, and saw that the expression

$$ints\_from(0)$$

stood for the infinite list of integers.

Fundamental to our work is the notion of an evaluator preserving an expression.

Definition 1.3.2

An evaluator ξ *preserves* an expression e if $\xi(e) \in Exp$. We will also say that an element in the standard semantics is preserved by ξ if all possible expressions which have it as their standard semantics are preserved by ξ.

□

By insisting that $\xi(e)$ is an element of Exp in the above definition, we are ensuring that it is neither infinite nor contains any subexpressions which are Ω_σ, that is, it is not a member of $Exp_\infty - Exp$. Since these two cases represent the only situations where we have infinite (non-terminating) computations, then an evaluator preserving an expression effectively means that the computation of the expression using that evaluator will be finite.

Let us call the evaluator which does no evaluation ξ_0. Lazy evaluation chooses the left-most outer-most redex and evaluates an expression as far as head normal form. We will call this evaluator ξ_1. It may be possible to detect that an expression which denotes a list can be evaluated further than head normal form. The evaluator which evaluates an expression to head normal form and then, recursively, evaluates the second argument to *cons*, until *nil* is reached (if ever) will be called ξ_2. Such a *cons* is often called a *right-strict cons*, and the process is termed evaluating the *spine* of the list. The evaluator which evaluates the spine of the list and each element of the list to head normal form will be called ξ_3. These last two evaluators were chosen because they treat all of the elements of a list in a uniform way.

The evaluators ξ_0 to ξ_3 have the following properties (where finite lists are defined formally in section 1.5.2):

Fact 1.3.3:

(i) ξ_0 preserves all values;

(ii) ξ_1 preserves all non-bottom values;

(iii) ξ_2 preserves all finite lists; and

(iv) ξ_3 preserves all finite lists which do not contain bottom elements.

□

From the discussion of the last two evaluators, we can see that they were chosen because they treat all of the elements of a list in the same way.

It was argued in section 1.2 that it is wasting computational resources to evaluate expressions whose values are not needed. A lazy evaluator only evaluates expressions whose values are needed. Furthermore, it initiates a non-terminating computation if and only if the semantics of the top-level expression is $\perp$. This prompts us to make the following definition.

Definition 1.3.4:

An evaluation strategy is *safe* if it never initiates an infinite computation unless the semantics of the original expression is $\perp$. We will also say that an evaluator for an expression is safe if the evaluation of the expression using that evaluator maintains a safe evaluation strategy.

□

We can intuitively see that safety implies that only expressions whose values are needed are evaluated. Since the evaluation of any expression allows the possibility of initiating an infinite computation (because the evaluators ξ_1 to ξ_3 all send expressions with semantics $\perp$ to a non-terminating computation), then doing some evaluation of an expression which would not be evaluated by a lazy evaluator may cause a non-terminating computation when the semantics of the top-level expression was not $\perp$, and so it is not a safe evaluation strategy.

We will insist that all evaluation strategies are safe. This then is our semantic criterion. Any evaluation strategy which satisfies it is allowed.

How much potential for parallel evaluation there is when we insist on safe evaluation strategies is a question which must be answered by experimentation. Whether or not there is a weaker condition than safety which allows more parallel evaluation but maintains the property that only expressions whose values are needed are evaluated remains an open question.

To maintain the safety of the evaluation strategy, we need to ensure that the evaluator chosen for an application

$$f\, e_1 \cdots e_n$$

does not initiate a non-terminating computation unless the semantics of the original expression was bottom. Suppose we have such an evaluator, ξ, then we must choose evaluators for each expression e_1 to e_n which are not allowed to initiate non-terminating computations unless the evaluation of $f\, e_1 \cdots e_n$ using ξ does. Sometimes we will have to choose an evaluator which does no evaluation.

In terms of preservation of values, an evaluator ξ preserves a set of elements. In the above example, we have to ensure that whenever the function f can return a value in the set of elements which is preserved by ξ, then we do not initiate a non-terminating computation in evaluating any of the expressions e_1 to e_n. Thus we need to find out the definedness of a function with respect to the definedness of its arguments, and make sure we preserve the values of the arguments for which the value of the function application is a value preserved by ξ. Some examples may help make this clearer.

For the function f defined by

$$f(x,y) = x + y$$

we know that the semantics of an application of f is bottom if and only if the semantics of either of the arguments is bottom. Thus a safe evaluation strategy is one which does not initiate a non-terminating computation unless the semantics of either of the arguments to f is $\perp$. A safe evaluation strategy then is one which preserves the non-bottom integers for both of the arguments of f, that is, evaluate both of the arguments to f in parallel.

If we are required to preserve all non-bottom elements in an application of

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

we can see that it is not safe to do any evaluation of the second argument to g because the application can be defined no matter how defined the second argument is, for the function never needs to evaluate it. The first argument however can be evaluated because the function is undefined whenever the first argument is. Thus a safe evaluation strategy allows the evaluation of the first argument of an application of g and not the second.

The function

$$\mathit{length}\ [\,] = 0$$
$$\mathit{length}\ x{:}xs = 1 + \mathit{length}\ xs$$

is defined whenever the argument list is finite (no matter how defined the elements of the list are). Thus, if we are required to preserve all non-bottom elements in an application of *length*, we must make sure that the evaluator chosen preserves all finite lists. Hence a

safe evaluation strategy may evaluate the argument to *length* using ξ_2.

Suppose that it is safe to use the evaluator ξ_3 in an application of the function

$$\begin{aligned} &append\ []\ L = L \\ &append\ x{:}xs\ L = cons(x, append(xs, L)). \end{aligned} \qquad (\dagger)$$

Then, since ξ_3 preserves all finite lists without bottom elements in them, we must make sure that whenever *append* can return a finite list with no bottom elements in it we do not initiate a non-terminating computation. This can only occur if both of the argument lists are finite and contain no bottom elements. It is thus safe to evaluate both of the arguments to the application of *append* using ξ_3 in this case.

In the examples we have been arguing intuitively about the definedness of functions. The work of this thesis is to develop a framework for abstract interpretation which we apply to finding out the definedness of functions. These results are then used to show how we can change the evaluation strategy as outlined above to allow for parallel evaluation while retaining the safety of the evaluation strategy.

1.4. 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 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 the answering of these and similar questions? 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.

As a more complex example of abstract interpretation which shows most of the essential features we will use later on, we introduce the "rule of signs", which is familiar from school mathematics. Our presentation is modelled on [Hankin 1986]. Let us consider the following abstract syntax of a language of arithmetic expressions:

$$Exp ::= c_n$$

(†)Note that *append* is a curried function even though here, and throughout the thesis, we will sometimes write applications as though a function is given a tuple of arguments.

$$| Exp + Exp$$

$$| Exp \times Exp$$

There is one constant, c_n for each integer n. A normal way to interpret such a language is to firstly interpret the set of constants $\{c_n\}$ as the integers, which we will denote by $\mathbf{Z}^{st}$, so that the constant c_n is interpreted to be the integer n. The symbols + and × are then interpreted as integer addition and multiplication respectively. These induce a standard interpretation function which we will denote by E^{st} :

$$E^{st} : Exp \rightarrow \mathbf{Z}^{st}$$

$$E^{st}\ [[c_n]] = n$$

$$E^{st}\ [[Exp_1 + Exp_2]] = E^{st}\ [[Exp_1]] \bar{+} E^{st}\ [[Exp_2]]$$

$$E^{st}\ [[Exp_1 \times Exp_2]] = E^{st}\ [[Exp_1]] \bar{\times} E^{st}\ [[Exp_2]]$$

where we have put a bar over the + and the × to help us remember that these are the real addition and multiplication functions.

If the property of interest is "Is the value of the expression positive or negative or zero?", then we could use the standard interpretation of our language to determine the answer by doing the calculation and then seeing whether it was indeed positive or negative or zero. For example, for the expression

$$c_{29} \times c_{-33} \times c_{64}$$

we could calculate the answer to be

$$E^{st}\ [[c_{29}]] \bar{\times} E^{st}\ [[c_{-33}]] \bar{\times} E^{st}\ [[c_{64}]]$$

which is

$$29 \bar{\times} -33 \bar{\times} 64 = -61248$$

and then see that the answer is negative. However, we all know a simple way of doing this, for we know that, for example, multiplying a positive number by a negative number always gives a negative number. The way we normally answer the question about the sign of the the answer is to do the "calculation"

$$(+) \times (-) \times (+) = (-)$$

where (+) represents the property of being positive, and similarly (-) the property of

being negative, and then say that the answer to the real calculation would have been negative.

What have we done? We have said that the important thing about the constant c_n was not it magnitude, but its sign, and provided an abstract interpretation, which we will denote by E^{ab}, that says

$$E^{ab}[[c_n]] = sign(E^{st}[[c_n]])$$

where

$$sign(n) = \begin{cases} (+) \text{ if } n>0 \\ (0) \text{ if } n=0 \\ (-) \text{ if } n<0 \end{cases}$$

We also have the rule of signs, where we will write the interpretations of + and × under the rule of signs as $\underline{+}$ and $\underline{\times}$ respectively :

$$\begin{array}{ll} (+)\,\underline{\times}\,(+) = (+) & (0)\,\underline{\times}\,(+) = (0) \\ (+)\,\underline{\times}\,(-) = (-) & (+)\,\underline{\times}\,(0) = (0) \\ (-)\,\underline{\times}\,(+) = (-) & (0)\,\underline{\times}\,(-) = (0) \\ (-)\,\underline{\times}\,(-) = (+) & (-)\,\underline{\times}\,(0) = (0) \\ (0)\,\underline{\times}\,(0) = (0) & \end{array}$$

The rule of signs gives $\underline{\times}$ as an abstract interpretation of ×.

For the abstract interpretation, $\underline{+}$, of +, some of the rules are obvious, for example,

$$\begin{array}{ll} (+)\,\underline{+}\,(+) = (+) & (0)\,\underline{+}\,(+) = (+) \\ (-)\,\underline{+}\,(-) = (-) & (0)\,\underline{+}\,(-) = (-) \\ (+)\,\underline{+}\,(0) = (+) & (0)\,\underline{+}\,(0) = (0) \\ (-)\,\underline{+}\,(0) = (-) & \end{array}$$

When we have one of the expressions :

$$(+)\,\underline{+}\,(-) \text{ or } (-)\,\underline{+}\,(+)$$

then we can no longer say what the result is, because the sign of the result depends on the magnitude of the two values, and we have abstracted away that information. We thus introduce the value $\top$ (pronounced "top") to represent the idea that we do not know what the sign of the calculation is. Another way of representing this would be to use the set $\{(-),(0),(+)\}$, but we prefer this way because then we do not have to introduce sets into the abstract interpretation. The rules for $\top$ are :

$$(-) \underline{\times} \top = \top \qquad (-) \underline{+} \top = \top$$
$$(0) \underline{\times} \top = (0) \qquad (0) \underline{+} \top = \top$$
$$(+) \underline{\times} \top = \top \qquad (+) \underline{+} \top = \top$$

and the other six equations obtained by changing the order of the arguments to the operators and retaining the same answers.

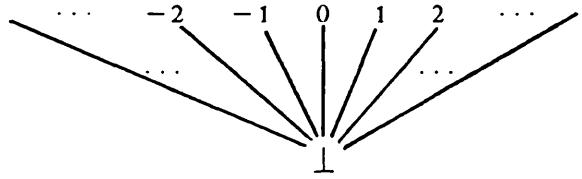
For completeness, we will introduce the value $\perp$ (pronounced "bottom") to represent the undefined integer, even though in our language we will not be able to write an expression which has this as its standard semantics. We have thus extended our domain Z^{st} to $Z_{\perp}^{st}$, where

$$Z_{\perp}^{st} = Z^{st} \cup \{\perp\}$$

and where we give the definedness ordering, $\leqslant$, on the domain

$z_1 \leqslant z_2$ if and only if $z_1 = \perp$ or $z_1 = z_2$.

$Z_{\perp}^{st}$ is an example of a *flat* domain because all of the elements from Z^{st} are on an equal level of definedness. We can draw this in a diagram as :



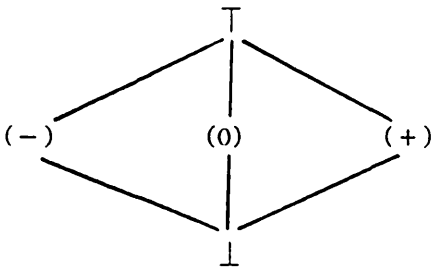
If $\perp$ appears as one of the arguments to $\overline{+}$ or $\overline{\times}$, then we must say what the answer is. We choose *strict* interpretations of these functions, so that the answer to an expression which has $\perp$ as either of the arguments must be $\perp$.

To model the bottom element in the domain $Z_{\perp}^{st}$, we introduce $\perp$ into the elements in our abstract domain. For both $\underline{+}$ and $\underline{\times}$, $\perp$ in either of the argument positions gives $\perp$.

We now have an abstract domain, which we will call Z^{ab}

$$Z^{ab} = \{\perp, (-), 0, (+), \top\}$$

where we define the ordering on the domain so that Z^{ab} is a complete lattice :



In the same way that we defined a standard interpretation for our language, we can provide an abstract interpretation, the semantic function being called E^{ab} :

$$E^{ab} : Exp \rightarrow \mathbf{Z}^{ab}$$

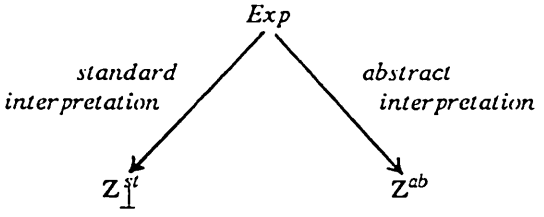
$$E^{ab}\,[[c_n]] = sign(E^{st}\,[[c_n]])$$

$$E^{ab}\,[[Exp_1 + Exp_2]] = E^{ab}\,[[Exp_1]] \mathbin{\underline{+}} E^{ab}\,[[Exp_2]]$$

$$E^{ab}\,[[Exp_1 \times Exp_2]] = E^{ab}\,[[Exp_1]] \mathbin{\underline{\times}} E^{ab}\,[[Exp_2]]$$

Notice that the form of the abstract interpretation is exactly the same as the form of the standard interpretation; all that has changed is the interpretation of the constants c_n, + and ×.

We now have two interpretations :



How can we say that if we get the answer (+) in the abstract interpretation, that the answer in the standard interpretation was really positive? That is, we must find a notion of correctness and prove that the abstract interpretation is correct. We begin by noting that the symbol (+) "represents" any positive integer. To capture this notion, we define a *concretisation* map :

$$Conc : \mathbf{Z}^{ab} \rightarrow \mathbf{P}(\mathbf{Z}^{st}_{\perp})$$

where $\mathbf{P}(X)$ is the powerset of X. It returns an element in the powerset because each element in $\mathbf{Z}^{ab}$ represents (possibly) many elements of $\mathbf{Z}^{st}_{\perp}$.

Rather than define this map directly, we will define it in a manner which is similar

to the way we will define such maps in Chapter 2. We can define an abstraction map :

$$abs : \mathbf{Z}_{\perp}^{st} \rightarrow \mathbf{Z}^{ab}$$

which relates the standard interpretation and the abstract interpretation of the constants. In this case, the abstraction map is just the *sign* map defined earlier, augmented with the rule

$$sign(\perp) = \perp$$

to cope with the fact that we now are dealing with $\mathbf{Z}_{\perp}^{st}$ rather than just $\mathbf{Z}^{st}$.

We can now define a map

$$Abs : \mathbf{P}(\mathbf{Z}_{\perp}^{st}) \rightarrow \mathbf{Z}^{ab}$$

which will allow us to find the abstract interpretation of sets of elements. When we have a set of elements, we have the possibility of having elements of differing sign in the same set. Suppose we had the set $\{-3,4\}$ which we wanted to abstract. Then we could apply the abstraction map *abs* to each element in the set to obtain the set $\{(-),(+)\}$. We added the point $\top$ to represent the fact that we were not sure what the sign of a result of a computation was. Here we can give it another reading, where it says that it represents sets of elements which have more than one sign in them. Because of the ordering we introduced on our domain, we can obtain this result by taking the least upper bound of the sets of elements we get by abstracting each element in the set. (Note that $\bigsqcup\{(-),(+)\}$ is $\top$.) Thus we define *Abs* by

$$Abs(S) = \bigsqcup\{abs(n) \mid n \in S\}.$$

Finally we are able to define the concretisation map. For $z \in \mathbf{Z}^{ab}$,

$$Conc(z) = \bigcup \{T \mid Abs(T) \leqslant z\}$$

The concretisation map collects together all of the elements which abstract to something at most as defined as z. If we calculate what this means for each of the elements of $\mathbf{Z}^{ab}$, then we find that :

$$Conc(\perp) = \{\perp\}$$

$$Conc((-)) = \{n \mid n<0\} \cup \{\perp\}$$

$$Conc((0)) = \{0\} \cup \{\perp\}$$

$$Conc((+)) = \{n \mid n>0\} \cup \{\perp\}$$

$$Conc(\top) = Z_{\perp}^{st}$$

and so see that indeed concretisation captures our notion of an element "representing" a set of values.

We can now state what we mean by correctness. For any evaluation

$$z_1 \underline{op} z_2 \Rightarrow z_3$$

where $z_1, z_2 \in Z^{ab}$ and $\underline{op}$ is either $\underline{+}$ or $\underline{\times}$, then we have that for all $n_1 \in Conc(z_1)$, $n_2 \in Conc(z_2)$,

$$n_1 \overline{op} n_2 \in Conc(z_3)$$

where $\overline{op}$ is $\mp$ if $\underline{op}$ was $\underline{+}$ and $\overline{\times}$ otherwise. It can be shown that the abstract interpretation we have defined satisfies this property.

As an example of this, let us return to the previous example and ask what the sign of the expression :

$$c_{29} \times c_{-33} \times c_{64}$$

is. Our previous calculation showed that the real answer was -61248 and so the answer is negative: Doing the calculation in the abstract domain we obtain :

$$E^{ab}[[c_{29}]] \underline{\times} E^{ab}[[c_{-33}]] \underline{\times} E^{ab}[[c_{64}]]$$

which is

$$(+) \underline{\times} (-) \underline{\times} (+) = (-)$$

From this we conclude that the answer calculated in the standard interpretation is negative. If we concretise $(-)$, we obtain the set of negative integers (plus $\perp$), and so we are able to conclude (because of the correctness of our abstract interpretation) that the result really was negative.

There is another feature of abstract interpretation that can be shown with this example. We note that the abstract interpretation does not give exact answers. For example,

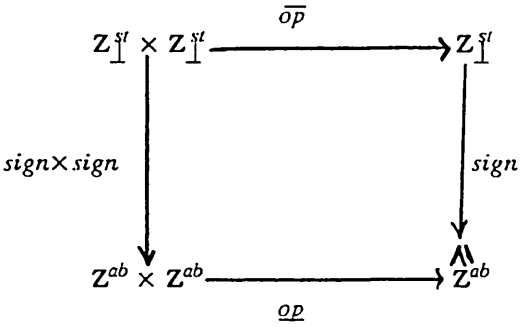
$$\begin{aligned} E^{ab}[[c_{-10} + c_{11}]] &= E^{ab}[[c_{-10}]] \underline{+} E^{ab}[[c_{11}]] \\ &= (-) \underline{+} (+) \\ &= \top \end{aligned}$$

whereas the sign of the real answer

$$sign(E^{st}\ [[c_{-10} + c_{11}]]) = (+)$$

is (+), and so the abstract interpretation loses information, but in a safe way; the abstract interpretation says that it does not know what the sign of the answer is, but does not wrongly conclude that it is negative (or some other wrong answer).

Those familiar with universal algebra will notice that we almost have that *sign* is a homomorphism. We have the following diagram :



where in the bottom right-hand corner we have $\leqslant$ rather than equality. Our abstraction map, *sign*, is then a *semi-homomorphism*. This is the general case for abstraction maps when we are working in the world of domains rather than just sets.

We can now summarise the key features of abstract interpretation. Given a set of symbols, we must give them an interpretation. Thus, if we have a language, we must give an interpretation of the language. Usually the symbols are chosen with one particular interpretation in mind. However, sometimes there are questions one wishes to ask which are hard to answer using the standard interpretation of the language; in fact, the questions we will ask about the definedness of functions are not recursive. It is sometimes possible to give an interpretation of the language which answers the questions we want to ask, but requiring significantly less work to do so. Such an interpretation is called an abstract interpretation. The example we have given here is to ask what the sign of the result of a calculation is. For an abstract interpretation to be of any use, it must be correct, in that anything we conclude from it must really be true. Finally, answers given by abstract interpretations are not exact, but they are safe, that is, any conclusion we draw from the abstract interpretation will be weaker than a conclusion we could obtain from the standard interpretation.

1.5. The Language Used Throughout This Thesis.

For the development of this thesis, we need a functional language. The simplest functional language that we know is the typed λ-calculus. We use the typed λ-calculus for the development of this thesis for both aesthetic and pragmatic reasons. Firstly, we believe that programs should be finitely typable. Secondly, the fact that our language is finitely typable means that we are able to give computable abstract interpretations for our language. In this section we introduce both the abstract syntax of the language, and the idea of interpretations of the language which give different semantics to the language. Having more than one interpretation for a language is fundamental in the work of abstract interpretation. The presentation of this section is after the style of [Abramsky 1985b].

1.5.1. Syntax

Given a set of base types $\{A_1, A_2 \cdots\}$, we define type expressions $\sigma, \tau, \cdots$ by

$$\sigma ::= A_i \mid \sigma \to \sigma \qquad (\dagger)$$

In Chapter 5 we will extend the type system of our language to include finite combinations of sums, products and lifting. At the moment, such constructed types are handled by making them "base types".

The language has a set of typed constants, denoted by $\{c_\sigma\}$, and we will choose for our typed constants the following :

- *integers* e.g. 0,5
- *booleans* i.e. *true* , *false*
- *Alist* - lists of elements of type A, i.e. elements of the recursive type $A \cong 1 + A \times Alist$
- arithmetic functions e.g. $+$, $-$, $\times$
- boolean functions e.g. **and** , **or**
- a conditional for each type σ denoted by $if_{bool \to \sigma \to \sigma \to \sigma}$ (or just *if*)

(†) Note that we have no type variables in the syntax of our type expressions, and so we are using a mono-typed λ-calculus and not a polymorphically-typed λ-calculus.

- list processing functions i.e. *hd* , *tl* , *cons* and *case*

For each type σ, we will assume an infinite supply of typed variables $Var_\sigma = \{x^\sigma, \cdots\}$. The terms in the language *Exp* then consist of typed terms $e{:}\sigma$ formed according to the following rules :

(1) $x^\sigma : \sigma$ variables

(2) $c_\sigma : \sigma$ constants

(3) $\dfrac{e : \tau}{\lambda x^\sigma . e : \sigma \to \tau}$ λ-abstractions

(4) $\dfrac{e_1 : \sigma \to \tau \quad e_2 : \sigma}{e_1\, e_2 : \tau}$ application

(5) $\dfrac{e : \sigma \to \sigma}{fix(e) : \sigma}$ fixed points

(Note that fix is part of the syntax and not a constant.)

1.5.2. Interpretations.

So far we have only given the syntactic constructions of our language. We need to give interpretations for our language. An interpretation, I, is given by

$$I = (\{D_A^I\}, \{c_\sigma^I\})$$

that is, interpretations for the base types and interpretations for each of the constants. For each base type A, we require that D_A^I must be a bounded-complete, ω-algebraic cpo [Scott 1981].

This is extended to the interpretation of the type $\sigma \to \tau$ by defining $D_{\sigma\to\tau}^I$ to be the domain of continuous maps $D_\sigma^I \to D_\tau^I$ [Scott 1981]. Each c_σ^I is given interpretation in D_σ^I. In particular, for each type σ, the interpretation of Ω_σ is $\perp_{D_\sigma^I}$.

The interpretation of base types and constants induces a semantic function

$$E^I : Exp \to Env^I \to \bigcup D_\sigma^I$$

where $Env^I = \{Env_\sigma^I\}$ and $Env_\sigma^I = Var_\sigma \to D_\sigma^I$.

$$E^I\, [[x^\sigma]]\, \rho^I = \rho^I(x^\sigma)$$

$$E^I\, [[c_\sigma]]\, \rho^I = c_\sigma^I$$

$$E^I\, [[\lambda x^\sigma . e]]\, \rho^I = \lambda y^{D_\sigma^I} . E^I\, [[e]]\, \rho^I[y/x^\sigma]$$

$$E^I\ [[e_1\ e_2\ ]]\ \rho^I\ =\ (E^I\ [[e_1]]\ \rho^I)\ (E^I\ [[e_2]]\ \rho^I)$$

$$E^I\ [[fix\ e]]\ \rho^I\ =\ fix(E^I\ [[e]]\ \rho^I)$$

(Note that fix, application and abstraction are interpreted the same way in all interpretations.)

Throughout the rest of the thesis we will have a *standard* interpretation $(\{D_A^{st}\}, \{c_\sigma^{st}\})$ where we have the usual flat domains for things like integers and booleans. The standard interpretation for *Alist* is obtained by solving the isomorphism

$$Alist \cong 1 + A \times Alist$$

over the category of domains [Smyth and Plotkin 1982] to obtain

$$D_{Alist}^{st} = D_A^{st}{}^*.nil \cup D_A^{st}{}^*.\perp_{D_{Alist}^{st}} \cup D_A^{st\,\omega}.$$

Here * is the Kleene star, denoting a finite sequence (possibly empty) of elements from the set which is starred, and so $D_A^{st}{}^*.nil$ are finite lists of elements of D_A^{st} (i.e. integers or booleans etc.), and $D_A^{st}{}^*.\perp_{D_{Alist}^{st}}$ are lists which have a finite number of elements from D_A^{st} and then have an undefined tail. The set of infinite lists is denoted by $D_A^{st\,\omega}$

It is useful to have some terminology to refer to various types of lists.

Definition 1.5.2.1:

A list, L, is

(*i*) *finite* if $L \in D_A^{st}{}^*.nil$.

(*ii*) *partial* if $L \in D_A^{st}.D_A^{st}{}^*.\perp_{D_{Alist}^{st}}$.

(*iii*) *infinite* if $L \in D_A^{st\,\omega}$.

□

Note that although the bottom list is normally called a partial list, we have separated it out because in Chapter 3 we will need to consider the set containing only the bottom list and the set containing all other partial lists. Thus, for our purposes, a partial list has at least one element of D_A^{st} in it, that is, can be written as $cons(e_1, e_2)$ for some e_1 and e_2.

We will call the induced semantic function E^{st}, and we will always use the environment ρ^{st} for the standard interpretation.

For the standard interpretation of constants, we will have the strict versions of all of the arithmetic and boolean functions. The conditional has the following interpretation :

$$(E^{st}\ [[\mathrm{if}_{bool\to\sigma\to\sigma\to\sigma}]]\ \rho^{st})\ \perp_{D_{bool}^{st}}\ x\ y = \perp_{D_\sigma^{st}}$$

$(E^{st}\ [[\text{if}_{bool\to\sigma\to\sigma\to\sigma}]]\ \rho^{st})\ tt\ x\ y = x$

$(E^{st}\ [[\text{if}_{bool\to\sigma\to\sigma\to\sigma}]]\ \rho^{st})\ ff\ x\ y = y$

The standard semantics of the *case* function is :

$$E^{st}\ [[case\ s\ f\ L]]\ \rho^{st} = E^{st}\ [[if\ null(L)\ then\ s\ else\ f(hd(L),tl(L))]]\ \rho^{st}$$

It is meant to be a translation of the more user-friendly pattern matching style of writing programs, being a case on the structure of the list. Thus

sumlist [] = 0
sumlist x:xs = x + *sumlist* xs

is translated into our language as :

$$sumlist = fix(\lambda f^{Alist\to int}.\lambda L_1^{Alist}.case(0,\lambda x^{A}.\lambda L_2^{Alist}.x+f(L_2)),L_1) \qquad (\dagger\dagger)$$

For *abstract* interpretations we will allow any finite, complete lattices, D_A^{ab} for each base type A, and these properties of finiteness and completeness are preserved by the interpretation of the type structure. In the examples of the use of the framework in this thesis, we will use in particular two abstract domains, namely the two point domain and the four point domain.

We will induce the interpretations of the constants as abstractions of their standard interpretation.

The induced semantic function will be called E^{ab} and we will always denote the environment used in the abstract interpretation by ρ^{ab}.

Note that in our abstract interpretation we have only parameterised out the interpretation of the base types and the constants. This is because we only need to change the interpretations of these things to answer the questions about evaluation strategies that we wish to ask. A framework for the abstract interpretation of the pure λ-calculus is provided in [Mycroft and Jones 1985], where the meaning of λ-abstraction and application are also part of the interpretation.

1.6. Domains, Powerdomains, Functions and Algebraic Relationships (†).

Because we will be developing a framework for abstract interpretation which is semantically well-founded, we will of necessity have to delve into the world of domain

(†) This section is lifted almost verbatim from some material in [Burn, Hankin and Abramsky 1985a], and we are indebted to Samson Abramsky for his original presentation of it.

(††) This notation means that the name *sumlist* stands for the expression on the right hand side of the equality symbol. This shorthand way of referring to expressions in the language will be used throughout the rest of this thesis.

theory in which the semantics of programming languages exists. It is possible to understand the technical details of this thesis using only basic concepts from domain theory [Scott 1981, 1982], category theory [Arbib and Manes 1975] and powerdomains [Plotkin 1976], [Smyth 1978]. The main results that we use can be summarised as a set of algebraic rules, which we will now develop. The proofs of the basic facts cited below are either directly in the literature, or obtainable by minor modifications therefrom; see [Plotkin 1976], [Hennessy and Plotkin 1979].

We shall be working over the category of domains described in [Scott 1981, 1982]. 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 $f:D\rightarrow E$, $g:E\rightarrow F$ is written thus :

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

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

$f \leqslant g$ iff 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(†) 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 $\mathbf{LC}(X) = \{y \mid$ 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 :

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

and for all $y \in Y$, there exists $x \in X$, $x \leqslant y$

(†) This terminology is due to the fact that these are the closed sets with respect to the Scott topology (c.f. for example [Gierz et. al. 1980]).

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:

$$(Pf)(X) = \{f(x) \mid x\in X\}^\circ$$

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 it 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 $\mathbf{P}(\bigsqcup f_i) = \bigsqcup \mathbf{P}f_i$.

Whenever we write $\mathbf{P}$ from now on we will mean the Hoare power domain functor.

Why are we using the Hoare powerdomain construction? The Hoare power domain naturally captures the idea of sets of elements with a certain level of definedness which is what is needed for our applications. It is also pleasant to work with from a technical point of view.

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

$$\{\!|.|\!\}_D : D\to\mathbf{P}D$$

defined by :

$$\{\!|d|\!\}_D = \mathrm{LC}(\{d\}).$$

This satisfies the following properties :

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

This says that $\{\!|.|\!\}$ 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

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

by

$$\biguplus_D(\Theta) = \{x \mid \text{for some } X \in \Theta.\ x \in X\} = \bigcup \Theta. \qquad (\dagger)$$

This satisfies :

(P7) $\biguplus_D$ is continuous.

(P8) for $f:D \to E$, $\biguplus_E \circ \mathbf{P}\mathbf{P}f = \mathbf{P}f \circ \biguplus_D$.

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

Now $(\mathbf{P}, \{\!|.|\!\}, \biguplus)$ 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) $\bigsqcup \circ \{\!|.|\!\}_D = id_D$

(P11) $\bigsqcup \circ \mathbf{P}(\bigsqcup) = \bigsqcup \circ \biguplus_D$.

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

Henceforth, we shall omit the subscripts from instances of $\{\!|.|\!\}$, $\biguplus$ 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 many results.

1.7. Overview of Thesis.

In Chapter 2 we develop a framework for safe abstract interpretations. The aim of abstract interpretation is to provide suitable computable abstract interpretations which describe properties of interest and from which we can make assertions about the standard interpretation.

It will be developed in such a way that the user of the framework has only to provide three things. The first two are

---

(†) The last equality says that $\biguplus$ is just the same as the ordinary set-theoretic union in the case of the Hoare powerdomain. This is not true in general for other powerdomain constructions.

(*i*) a finite, complete lattice as an abstract domain for each base type; and

(*ii*) a strict, continuous abstraction map for each base type from the standard interpretation of the type to the abstract interpretation of that type.

These should make the distinctions in the standard interpretation of the base type that are desired. Abstraction maps for the higher types are then induced in a natural way. Thirdly,

(*iii*) each of the constants must be provided with an abstract interpretation which satisfies a safety relationship with respect to the standard semantics of the constant.

The framework is proved correct, and some theorems are given which can be used as practical tests in the use of the theory.

The framework of Chapter 2 is applied in Chapter 3 to give an abstract interpretation for the definedness of functions. This is used to show when the evaluation strategy can be safely changed to allow some evaluation of arguments to functions in parallel with, or before, applying the function.

Abstract domains for each base type are determined using two different intuitions, both of which end up giving the same abstract domains. The first intuition is to look at the definedness level of elements in the standard interpretation, while the second is to look at how the sensible evaluators behave on expressions standing for elements in the standard interpretation. The abstract domains chosen reflect the divisions made by the differing amounts of evaluation. We define best approximations of the constants in our language by deriving them as abstractions of the standard interpretation of the constants. Having defined the abstract interpretations of the constants, we give some examples of determining the abstract interpretation of user-defined functions.

The correctness of the abstract interpretation follows directly from Chapter 2. It turns out that we need to be able to determine the evaluation information which is true in a particular context (context-sensitive) as well as that which is true in all contexts (context-free). The two necessary theorems again follow directly from Chapter 2.

Finally, we show how both of these sorts of information can be used to safely change the evaluation strategy and how this information can be encoded on the graphs of functions. [Wadsworth 1971]

In applying the abstract interpretation developed in Chapter 3, we will see when a function application is undefined, and from that deducing how much evaluation is safe for each of the arguments. If the arguments are function applications, then instead of just assuming that it is safe to evaluate the applications to head normal form, which is how the theory is used in Chapter 3, we may know that it is safe to do more evaluation.

It would be better if we could carry this information inwards and so we may be able to more evaluation of arguments to this function than we would have otherwise have been able to do. Chapter 4 begins by giving a simple example of where this is the case.

Using the abstract interpretation developed in Chapter 3, we are able to give two more theorems in Chapter 4 which answer the question : Given a safe evaluator for a function application, what are the safe evaluators for each of the arguments in the function application? The theorems allow us to develop *evaluation transformers*, one for each argument, which transform a safe evaluator for a function application to a safe evaluator for that argument. They can be determined directly from the abstract interpretation.

The theoretical framework developed and used up to the end of Chapter 4 has the function space as the only type constructor. Complex data types such as infinite lists (infinite sums of products) were handled by putting them in a black box and treating them as a base type, giving them a standard and abstract interpretation. In Chapter 5 we are slightly more imaginative, adding types which are constructed from finite combinations of sums, products and lifting. We define in a natural way abstract interpretations and abstraction maps for the structured types from the corresponding abstract interpretations and abstraction maps of their components. We show that the vital safety relationship between the standard and abstract interpretation still holds, and thus the correctness of the abstract interpretation follows as in Chapter 2.

We draw our work together in Chapter 6, and point out some areas in which more work must be done, both theoretical in terms of extending results to a polymorphically typed framework, and pragmatic in terms of implementing the theory in compilers and computer architectures.

Chapter 2

A Framework For the Safe Abstract Interpretation of Functional Languages

We develop a framework for safe abstract interpretations. The aim of abstract interpretation is to provide suitable computable interpretations which describe properties of interest and from which we can make assertions about the standard interpretation.

In the first section we explain what we mean be the *correctness* of an abstract interpretation. The main thrust of the rest of the chapter is to provide a framework for the abstract interpretation of functional languages which is correct. It will be developed in such a way that the user of the framework has only to provide three things. The first two are a finite, complete lattice as an abstract domain for each base type, and a strict, continuous abstraction map from the standard interpretation of that type to the abstract interpretation. These should make the distinctions in the standard interpretation of the base type that are desired. Abstraction maps for the higher types are then induced in a natural way. Thirdly, each of the constants must be provided with an abstract interpretation which satisfies a safety relationship.

As a more detailed survey of this chapter, in the first section we motivate and give a definition of the correctness of an abstract interpretation as well as motivating the need for, and form of the definition of various functions that are needed in the subsequent development. We formally define these maps and prove they are well-behaved in the second section. Some more useful forms of the definition of the abstraction maps are given in the third section. Properties such as strictness and bottom-reflexivity are often useful to have, and it is shown that these properties are inherited from the abstraction maps on the base types. We defined abstraction maps and concretisation maps in such a way that they are adjoined functions; this is proved to be true in the fifth section, and in the following section we prove two propositions which are useful in proving a safety relation between the abstract interpretation and standard interpretation which will allow us to prove the correctness of the abstract interpretation.

In the application of this theory in Chapter 3 where a definedness interpretation is given, we will see that the definedness of a higher-order function depends on the definedness of any argument which is a function. If we were to be totally pessimistic about the definedness level of functions, then we would have to give information about the definedness of a function which was true in every application of that function, that is,

information which is true regardless of the context in which the function appears. However, if we take into account the contextual information of an application of a function we are able to sometimes to find out more about the definedness of the function in this particular application. It turns out that we cannot do without either of the types of information. The final two theorems in this chapter give practical tests for finding out both of these sorts of information.

The work can be seen as a generalisation of the theory of [Burn, Hankin and Abramsky 1985a] and [Hankin, Burn and Peyton Jones 1986] to situations where we have more complex abstract domains than the two point domain for the abstract interpretations of base types.

2.1. Motivation for the Definition of Abstraction and Concretisation Maps.

In section 1.5 of Chapter 1 we introduced the idea of interpretations. We were particularly interested in providing a computable abstract interpretation with which we could do calculations and make assertions about computations in the standard interpretation. Suppose that f is strict and that we have the abstract interpretation of f:

$$D_{\sigma}^{ab} \xrightarrow{E^{ab}\,[[f]]\,\rho^{ab}} D_{\tau}^{ab}$$

and wish to make assertions about a calculation using the standard interpretation :

$$D_{\sigma}^{st} \xrightarrow{E^{st}\,[[f]]\rho^{st}} D_{\tau}^{st}$$

Given that

$$(E^{ab}\,[[f]]\,\rho^{ab})\ \bar{s} = \bar{t}$$

what do we wish to conclude? A reasonable statement would be that for all s "represented by" $\bar{s}$, the value $\bar{t}$ "represents" the calculation $(E^{st}\ [[f]]\,\rho^{st})\ s$.

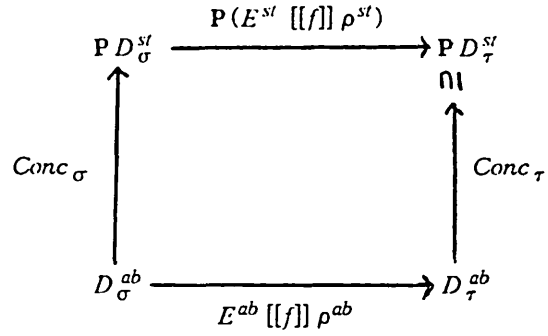
If we call the process of going from $\bar{s}$ to the set of things that $\bar{s}$ "represents" *concretisation*, and assume that for each type σ that we have a map $Conc_{\sigma}$ which does this, then we can state the above condition formally, and give it the status of a definition :

Definition 2.1.1:

An abstract interpretation is *correct* if $(E^{ab}\ [[f]]\ \rho^{ab})\ \bar{s} = \bar{t}$ implies that for all $s \in Conc_\sigma(\bar{s})$, $(E^{st}\ [[f]]\ \rho^{st})\ s \in Conc_\tau(\bar{t})$.

□

We shall see in section 2.8 that the correctness of an abstract interpretation is implied by the following *safety diagram*, which is essentially an adaptation to the world of the Hoare power domain of diagrams appearing elsewhere, for example, [Cousot and Cousot 1979], [Mycroft 1981] and [Mycroft and Nielson 1983]:



i.e. $Conc_\tau \circ (E^{ab}\ [[f]]\ \rho^{ab}) \supseteq \mathbf{P}(E^{st}\ [[f]]\ \rho^{st}) \circ Conc_\sigma$

The use of $\supseteq$ in the above diagram captures the notion of safety, for it implies that the result obtained in a calculation in the abstract interpretation represents a superset of the possible results in the standard interpretation.

So far we have been assuming that we have a concretisation map $Conc_\sigma$ for each type σ; we now turn our attention to how we can define such concretisation maps. Initially we look at the question the opposite way around. For each type σ, there are possibly many things in D_σ^{st} which have equal levels of definedness. As an example, consider $\lambda x^{D_{int}^{st}}.x+1$ and $\lambda x^{D_{int}^{st}}.x+2$ which are in $D_{int\to int}^{st}$ and both of which are defined everywhere except at $\perp_{D_{int}^{st}}$. It would be useful to be able to define a map

$$Abs_\sigma : \mathbf{P} D_\sigma^{st} \to D_\sigma^{ab} \qquad (\dagger)$$

which mapped sets of things of equal definedness to the same element in the abstract domain.

(†) We note that Abs_σ is a map which takes an element of the Hoare power domain and so a more correct intuition is that $Abs_\sigma(S)$ represents the most defined elements in a (left-closed) set S.

A natural way of defining maps is to define them inductively over the type structure. Suppose therefore that Abs_σ, Abs_τ and $Conc_\sigma$ and $Conc_\tau$ have been defined and that we wish to define

$$Abs_{\sigma\to\tau} : \mathbf{P}(D_\sigma^{st} \to D_\tau^{st}) \to (D_\sigma^{ab} \to D_\tau^{ab})$$

$$Conc_{\sigma\to\tau} : (D_\sigma^{ab} \to D_\tau^{ab}) \to \mathbf{P}(D_\sigma^{st} \to D_\tau^{st})$$

Given a set $S \in \mathbf{P}(D_\sigma^{st} \to D_\tau^{st})$, remember that Abs_σ says that the maximum level of definedness of the elements of S is a particular value in the abstract domain. Thus we need to define a map

$$abs_{\sigma\to\tau} : D_{\sigma\to\tau}^{st} \to D_{\sigma\to\tau}^{ab}$$

so that we can test the level of definedness of each element of S and then choose the maximum one of these for the value of $Abs_{\sigma\to\tau}(S)$.

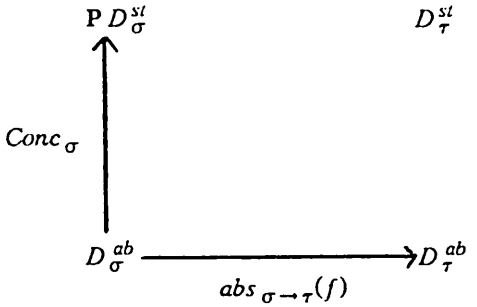
Given an $f \in D_\sigma^{st} \to D_\tau^{st}$:

$$D_\sigma^{st} \xrightarrow{\quad f \quad} D_\tau^{st}$$

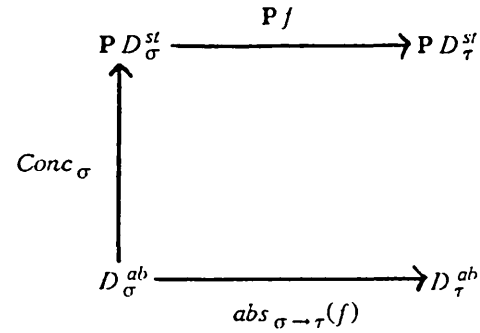
how do we define

$$D_\sigma^{ab} \xrightarrow{\quad abs_{\sigma\to\tau}(f) \quad} D_\tau^{ab} ?$$

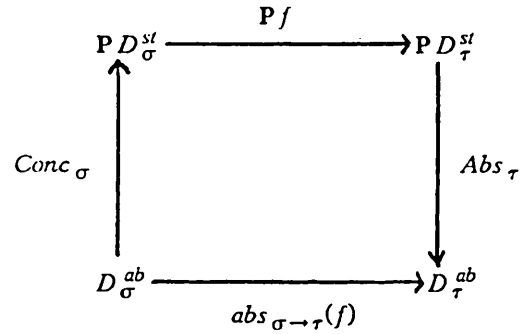
We know that $abs_{\sigma\to\tau}(f)$ will be applied to $\bar{s} \in D_\sigma^{ab}$, and that $\bar{s}$ "represents" all values in $Conc_\sigma(\bar{s})$. Furthermore, concretisation returns a set of values, that is, an element of $\mathbf{P}\, D_\sigma^{st}$. So we have the following situation :



We wish to say that $f \in D_\sigma^{st} \to D_\tau^{st}$ is "at most as defined as" something, by saying that $(abs_{\sigma\to\tau}(f))\,\bar{s} = \bar{t}$ where $\bar{t}$ represents the most defined values that f can take for values represented by $\bar{s}$. This implies that we must apply f to all of the elements s in $Conc_\sigma(\bar{s})$ and we can do this by applying $\mathbf{P}$ to f to obtain the diagram :



All that we need to find now is a map from $\mathbf{P}\,D_\sigma^{st}$ to D_τ^{ab}, which maps a set of values down to the element in the abstract domain that represents that set, and we already have such a map in Abs_τ. Thus we can complete the diagram for the definition of $abs_{\sigma\to\tau}(f)$:



and write it out as a formula :

$$abs_{\sigma\to\tau}(f) = Abs_\tau \circ \mathbf{P} f \circ Conc_\sigma$$

We have so far said that Abs_σ picks out a value which represents the most defined elements in a set of things, and now we have a map, abs_σ which works out the definedness level of a single element. Thus we can find out the definedness of a set of things by applying abs_σ to each element in the set. Furthermore, since we have insisted our abstract domains are finite, complete lattices, we can take the least upper bound of the set of results from applying abs_σ to a set of values in the standard interpretation to model the idea of choosing the maximum value. Thus a reasonable definition of Abs_σ is

$$Abs_\sigma = \bigsqcup \circ \mathbf{P}\, abs_\sigma$$

The least upper bound in the definition is capturing two intuitions. Firstly, an upper bound is necessitated by the fact that we are developing a safety analysis, and secondly, having the least upper bound means that we have the best safe representation of the

value.

We will also define $Conc_\sigma$ so that Abs_σ and $Conc_\sigma$ are adjoined functions for each type σ. Adjointness is a very important mathematical property [Gierz et. al. 1980], and can be interpreted in this context to mean that the abstract domain models the standard domain as closely as possible.

Furthermore, if we can provide an abstraction map which satisfies that for all $e{:}\sigma$ that

$$E^{ab}[[e]]\rho^{ab} \geqslant abs_\sigma(E^{st}[[e]]\rho^{st}) \qquad (\S)$$

and the maps Abs_σ and $Conc_\sigma$ are adjoined functions, then we can prove the correctness of abstract interpretation. (See the progression of the argument from Theorem 2.7.1 to Theorem 2.8.2.)

It is shown in [Abramsky 1985b] that the definition of abs_σ we have motivated above is the "best" possible abstraction map in that it loses least information while still satisfying (§).

One final thing that is worth noting is that we have a dual reading of the elements of the abstract domain. Firstly they can be seen as the abstraction of one particular element of the standard interpretation (using abs_σ) and secondly as saying the elements of a set of things from the standard interpretation are at most as defined as a certain value. Concretisation captures the second reading by returning the set of all values which abstract to something less than (or equal to) the element which we are concretising.

2.2. Formal Definition of Abstraction and Concretisation Maps.

Having motivated the need for the maps abs_σ, Abs_σ and $Conc_\sigma$ for each type σ, and the form of their definition in the previous section, we now proceed with the formal definition. At the base types we presume that we are given abstract domains D_A^{ab}, for each base type A, which are finite, complete lattices, and strict, continuous abstraction maps

$$abs_A : D_A^{st} \to D_A^{ab}$$

where these make the distinctions on the base types that the user wishes to make. The abstract domains need to be complete lattices because we will need to take least upper bounds, and they need to be finite so that the testing of equality of functions when taking least fixed points is an effective procedure. Strictness of the abstraction maps is required so that concretisation is well-defined and for the proof of Proposition 2.6.2.

We can then define

Definition 2.2.1:

$$Abs_A : \mathbf{P}\, D_A^{st} \to D_A^{ab}$$

$$Abs_A = \bigsqcup \circ \mathbf{P}\, abs_A$$

□

Definition 2.2.2:

$$Conc_A : D_A^{ab} \to \mathbf{P}\, D_A^{st}$$

$$Conc_A(\bar{a}) = \biguplus \{T \mid Abs_A(T) \leqslant \bar{a}\} \qquad (\dagger)$$

□

From the previous section we saw that the way to define the abstraction map for higher types was :

Definition 2.2.3:

$$abs_{\sigma\to\tau} : D_{\sigma\to\tau}^{st} \to D_{\sigma\to\tau}^{ab}$$

$$abs_{\sigma\to\tau}(f) = Abs_\tau \circ \mathbf{P}\, f \circ Conc_\sigma$$

□

and we also define the maps $Abs_{\sigma\to\tau}$ and $Conc_{\sigma\to\tau}$ in an analogous way to the definitions on the base types :

Definition 2.2.4:

$$Abs_{\sigma\to\tau} : \mathbf{P}\, D_{\sigma\to\tau}^{st} \to D_{\sigma\to\tau}^{ab}$$

$$Abs_{\sigma\to\tau} = \bigsqcup \circ \mathbf{P}\, abs_{\sigma\to\tau}$$

□

Definition 2.2.5:

$$Conc_{\sigma\to\tau} : D_{\sigma\to\tau}^{ab} \to \mathbf{P}\, D_{\sigma\to\tau}^{st}$$

(†) Note how we have defined $Conc_A$ so that it will be an upper adjoint to Abs_A [Gierz et. al. 1980].

$$Conc_{\sigma\to\tau}(\bar{f}) = \bigsqcup\{T \mid Abs_{\sigma\to\tau}(T) \leqslant \bar{f}\}$$

□

We must of course show that these maps are well-defined and continuous, but first a definition and a subsidiary Lemma.

Definition 2.2.6:

A function $f \in D\to E$ is *strict* if $f(\bot_D) = \bot_E$.

□

Lemma 2.2.7:

If $f \in D^I_\sigma \to D^I_\tau$ is strict and D^I_τ is a complete lattice, then $\bigsqcup \circ \mathbf{P} f$ is strict.

Proof :

$$(\bigsqcup \circ \mathbf{P} f)\{\bot_{D^I_\sigma}\} = \bigsqcup\{f\,x \mid x \in \{\bot_{D^I_\sigma}\}\}^\circ \qquad \text{definition of } \mathbf{P} \text{ on morphisms}$$

$$= \bigsqcup\{f \bot_{D^I_\sigma}\}^\circ$$

$$= \bigsqcup\{\bot_{D^I_\tau}\}^\circ \qquad \text{since } f \text{ is strict}$$

$$= \bot_{D^I_\tau} \qquad \text{(P10)}$$

□

Lemma 2.2.8:

If for each base type A, we are given a strict, continuous abstraction map $abs_A : D^{st}_A \to D^{ab}_A$, then for all types σ

(*i*) abs_σ is continuous.

(*ii*) Abs_σ is continuous.

(*iii*) abs_σ and Abs_σ are strict.

(*iv*) $Conc_\sigma$ is well-defined and continuous.

Proof :

We prove this by induction on the type structure.

(*i*) This is true of the base types by our condition of the continuity of the abstraction maps on base types.

(*ii*) On base types we have that this is true since $\bigsqcup$ is continuous, $\mathbf{P}f$ is continuous if f is (P2), and the composition of continuous functions is continuous.

(*iii*) abs_A is strict by the condition of the lemma and thus Abs_A is strict by Lemma 2.2.7.

(*iv*) We have to prove that $Conc_A$ is well-defined and monotonic, for its source is a finite domain and hence is continuous if it is monotonic.

To prove well definedness, we must show that $\{T \mid Abs_A(T) \leqslant \bar{a}\}$ is a non-empty Scott-closed subset of $\mathbf{P}\mathbf{P}D_A^{st}$. Since we have that Abs_A is strict (part (*iii*) of induction), we have that the set $\{T \mid Abs_A(T) \leqslant \bar{a}\}$ is non-empty. Denoting $\{T \mid Abs_A(T) \leqslant \bar{a}\}$ by Θ, to show that Θ is Scott-closed we need to show that (*a*) Θ is left-closed and that (*b*) Θ is closed under least upper bounds of directed sets. The first is true since if $Y \leqslant X \in \Theta$, then $Abs_A(Y) \leqslant Abs_A(X) \leqslant \bar{a}$ and so $Y \in \Theta$. The second is true for if $\Delta \subseteq \Theta$ is a directed set, then $Abs_A(\bigsqcup \Delta) = \bigsqcup\{Abs_A(X) \mid X \in \Delta\}$ and since $Abs_A(X) \leqslant \bar{a}$ for all $X \in \Delta$, $\bigsqcup\{Abs_A(X) \mid X \in \Delta\} \leqslant \bar{a}$.

To show monotonicity of $Conc_A$, let $s_1, s_2 \in D_A^{ab}$, $s_1 \leqslant s_2$. Then

$$Conc_A(s_1) = \biguplus\{T \mid Abs_A(T) \leqslant s_1\} \text{ and } Conc_A(s_2) = \biguplus\{T \mid Abs_A(T) \leqslant s_2\}.$$

Clearly, $\{T \mid Abs_A(T) \leqslant s_1\} \subseteq \{T \mid Abs_A(T) \leqslant s_2\}$ since $s_1 \leqslant s_2$ and so $Conc_A(s_1) \leqslant Conc_A(s_2)$. Thus $Conc_A$ is monotonic and hence continuous.

Assuming (*i*) to (*iv*) are true for types σ and τ we prove them for type $\sigma \to \tau$.

(*i*) $abs_\sigma(f) = Abs_\tau \circ \mathbf{P}f \circ Conc_\sigma$ and is thus continuous because by the induction hypothesis it is the composition of continuous functions.

(*ii*) Follows as for the base case.

(*iii*) $abs_{\sigma\to\tau}(\bot_{D^{st}_{\sigma\to\tau}})\,\bar{s} = (Abs_\tau \circ \mathbf{P}(\bot_{D^{st}_{\sigma\to\tau}}) \circ Conc_\sigma)(\bar{s})$

$= \bigsqcup(\mathbf{P}(abs_\tau \circ \bot_{D^{st}_{\sigma\to\tau}}) \circ Conc_\sigma)(\bar{s})$

$= \bigsqcup(\mathbf{P}(abs_\tau \circ \bot_{D^{st}_{\sigma\to\tau}}))\{s \mid s \in Conc_\sigma(\bar{s})\}$

$= \bigsqcup\{abs_\tau(\bot_{D^{st}_{\sigma\to\tau}}(s)) \mid s \in Conc_\sigma(\bar{s})\}^\circ$

$= \bigsqcup\{abs_\tau(\bot_{D^{st}_\tau})\}^\circ$

$= \bigsqcup\{\bot_{D^{ab}_\tau}\}^\circ$ by induction hypothesis (*iii*)

$= \bot_{D^{ab}_\tau}$ (P10)

The result holds for $Abs_{\sigma\to\tau}$ by Lemma 2.2.7.

(*iv*) Follows as for the base case.

□

In the proof of Lemma 2.2.8 we have made use of all of the properties of the abstraction maps and abstract domains for base types which we have insisted upon. Firstly, continuity is needed for the abstraction maps on the base types so that all of the maps we use are continuous, which is needed for proving the safety of the framework. The strictness of the abstraction maps on base types, which is preserved by the abstraction maps on higher types, is needed to prove the well-definedness of the concretisation maps. To guarantee the existence of least upper bounds for the definition of the abstraction maps, we need the property that the abstract domains are complete lattices. Finally, finiteness of the abstract domains is needed not only so that we will develop an effective analysis, but because we were able to use the fact that we needed only to prove that the concretisation maps were monotonic for them to be continuous. Abramsky [Abramsky 1985b] has shown that if the abstract domains are any complete lattices (including infinite ones), then the concretisation maps which are induced from the abstraction maps are not in general continuous. If the abstraction maps also map finite elements [Scott 1981] to finite elements, then the concretisation maps will be continuous [Abramsky 1985b]. Since our abstract domains are finite, this condition trivially holds.

It is useful to have a relationship between $\mathbf{P}abs_\sigma$ and Abs_σ. This is stated in the following Lemma.

Lemma 2.2.9:

$$\mathbf{P}abs_\sigma \leq \lambda.\beta \circ Abs_\sigma$$

Proof :

$$\mathbf{P}abs_\sigma \leq \lambda.\beta \circ \amalg \circ \mathbf{P}abs_\sigma \qquad \text{since } \lambda.\beta \circ \amalg \geq id$$

$$= \lambda.\beta \circ Abs_\sigma \qquad \text{Definition of } Abs_\sigma$$

□

2.3. Two Useful Forms of the Definition of the Abstraction Map.

We present two alternative forms for the abstraction map $abs_{\sigma\to\tau}$ when τ is a function space.

Proposition 2.3.1:

Suppose $f \in D^{st}_{\sigma_1} \to \cdots \to D^{st}_{\sigma_n} \to D^{st}_{\tau}$. Then

$$abs_{\sigma_1\to\cdots\to\sigma_n\to\tau}(f) = \lambda\overline{x_1}^{D^{ab}_{\sigma_1}} \cdots \lambda\overline{x_n}^{D^{ab}_{\sigma_n}} . \bigsqcup \{abs_\tau(f\, x_1 \cdots x_n) \mid \operatorname*{and}_{i=1}^{n} abs_{\sigma_i}(x_i) \leqslant \overline{x_i}\}^\circ$$

Proof :

Suppose $f \in D^{st}_{\sigma_1} \to \cdots \to D^{st}_{\sigma_n} \to D^{st}_{\tau}$. Then from [Abramsky 1985b, Lemma 6.24] we have that

$$abs_{\sigma_1\to\cdots\to\sigma_n\to\tau}(f) = \lambda\overline{x_1}^{D^{ab}_{\sigma_1}} \cdots \lambda\overline{x_n}^{D^{ab}_{\sigma_n}} . \bigsqcup \{abs_\tau(f\, x_1 \cdots x_n) \mid \operatorname*{and}_{i=1}^{n} abs_{\sigma_i}(x_i) \leqslant \overline{x_i}\}$$

$$= \lambda\overline{x_1}^{D^{ab}_{\sigma_1}} \cdots \lambda\overline{x_n}^{D^{ab}_{\sigma_n}} . \biguplus \{abs_\tau(f\, x_1 \cdots x_n) \mid \operatorname*{and}_{i=1}^{n} abs_{\sigma_i}(x_i) \leqslant \overline{x_i}\}^\circ$$

since $\biguplus X^\circ = \bigsqcup X$ for finite, complete lattices [Abramsky 1985b].

□

The second form of the abstraction map, which we will give in Proposition 2.3.3, is a direct consequence of the above Proposition and the following Lemma.

Lemma 2.3.2:

$$abs_\sigma(s) \leqslant \bar{s} \Rightarrow s \in Conc_\sigma(\bar{s})$$

Proof :

Suppose $abs_\sigma(s) \leqslant \bar{s}$. Then

$$Conc_\sigma(\bar{s}) = \biguplus \{T \mid Abs_\sigma(T) \leqslant \bar{s}\}$$

$$= \biguplus \{T \mid \biguplus \{abs_\sigma(t) \mid t \in T\}^\circ \leqslant \bar{s}\}$$

definition of Abs_σ and definition of P on morphisms

and certainly $\biguplus \{abs_\sigma(t) \mid t \in \{\!|s|\!\}\}^\circ \leqslant \bar{s}$ since $abs_\sigma(s) \leqslant \bar{s}$, and thus $s \in Conc_\sigma(\bar{s})$.

□

Proposition 2.3.3:

Suppose $f \in D^{st}_{\sigma_1} \to \cdots \to D^{st}_{\sigma_n} \to D^{st}_{\tau}$. Then

$$abs_{\sigma_1 \to \cdots \to \sigma_n \to \tau}(f) = \lambda \overline{x_1}^{D^{alt}_{\sigma_1}} \cdots \lambda \overline{x_n}^{D^{alt}_{\sigma_n}} . \bigsqcup \{abs_\tau(f\, x_1 \cdots x_n) \mid \operatorname{and}_{i=0}^{n} x_i \in Conc_{\sigma_i}(\overline{x_i})\}^\circ$$

Proof :

This follows directly from Proposition 2.3.2 and Lemma 2.3.3.

□

2.4. Properties of Abstraction and Concretisation Maps of Higher Types.

Many useful properties of abstraction maps of the base types are carried over to the abstraction maps for the higher types. Although these properties were used in the development of [Burn, Hankin and Abramsky 1985a], we note that it is in [Abramsky 1985b] that the first record is made of separating out the properties of abstraction maps on base types which are preserved on the higher types. We record here three such properties which are useful in the ensuing development.

Fact 2.4.1:

If abs_A is strict for all base types A, then abs_σ and Abs_σ are strict for all types σ (Lemma 2.2.8 (*iii*)).

□

Definition 2.4.2:

A function $f \in D \to E$ is $\perp$-*reflecting* if $f(d) = \perp_E \Rightarrow d = \perp_D$.

□

Lemma 2.4.3:

If $f \in D^l_\sigma \to D^l_\tau$ is $\perp$-reflecting and D^l_τ is a complete lattice, then $\bigsqcup \circ \mathbf{P} f$ is $\perp$-*reflecting*.

Proof :

Suppose $\bigsqcup \{f\, s \mid s \in S\}^\circ = (\bigsqcup \circ \mathbf{P} f)\, S = \perp_{D^l_\tau}$. Then we must have for each $s \in S$ that $f\, s = \perp_{D^l_\tau}$ (P10), and since f is $\perp$-reflecting, this means that $s = \perp_{D^l_\sigma}$ and so $S = \{\perp_{D^l_\sigma}\}$.

□

Lemma 2.4.4:

If for each base type A abs_A is $\perp$-reflecting, then abs_σ and Abs_σ are $\perp$-reflecting for each type σ.

Proof :

We prove this by induction over the type structure, where the base case is true for abs_A by hypothesis and for Abs_A by Lemma 2.4.3. Assuming that the result holds for abs_σ, abs_τ, Abs_σ and Abs_τ,

$$abs_{\sigma\to\tau}(f) = \perp_{D^{ab}_{\sigma\to\tau}}$$

$$\Rightarrow abs_{\sigma\to\tau}(f)\,\top_{D^{ab}_{\sigma}} = \perp_{D^{ab}_{\tau}}$$

$$\Rightarrow \bigsqcup\{abs_\tau(f\,s) \mid s\in Conc_\sigma(\top_{D^{ab}_{\sigma}})\}^\circ = \perp_{D^{ab}_{\tau}} \qquad \text{Proposition 2.3.3}$$

$$\Rightarrow abs_\tau(f\,s) = \perp_{D^{ab}_{\tau}} \text{ for all } s\in D^{st}_{\sigma}$$

Since by the induction hypothesis, abs_τ is $\perp$-reflecting, we have that for all $s\in D^{st}_{\sigma}$

$$f\,s = \perp_{D^{st}_{\tau}}$$

$$\Rightarrow f = \perp_{D^{st}_{\sigma\to\tau}}$$

Abs_σ is $\perp$-reflecting by Lemma 2.4.3.

□

While ontoness is not required for the technical development of this thesis, if we have onto abstraction functions on the base types, then there is no useless (elements) in the abstract domains. The fact that this can often be shown to be true for higher types means that there is no junk in the abstract domains for higher types.

Lemma 2.4.5:

If $f \in D^I_\sigma \to D^I_\tau$ is onto and D^I_τ is a complete lattice, then $\bigsqcup \circ \mathbf{P} f$ is onto.

Proof :

From [Burn, Hankin and Abramsky 1985a, Lemma 1] we have that for such an f, $\mathbf{P}f$ is onto. This implies that for all $\bar{t} \in D_\tau^I$ that there is an $S \in \mathbf{P}D_\sigma^I$ such that $\mathbf{P}f(S) = \{\!|\bar{t}|\!\}$. Taking the least upper bound of both sides, we find that $\bigsqcup(\mathbf{P}f(S)) = \bar{t}$ by (P10), and so we have exhibited an S for every $\bar{t} \in D_\tau^I$ and so $\bigsqcup \circ \mathbf{P}f$ is onto.

□

Lemma 2.4.6:

If abs_A is onto for each base type, and for each base type A we can define a continuous function $abs_A^{-1} : D_A^{ab} \to D_A^{st}$ which is a right inverse of abs_A, that is, $abs_A \circ abs_A^{-1} = id_{D_A^{ab}}$, then for all types σ

(*i*) there is a continuous function $abs_\sigma^{-1} : D_\sigma^{ab} \to D_\sigma^{st}$ which is a right inverse of abs_σ.

(*ii*) abs_σ and Abs_σ are onto.

(*iii*) $Abs_\sigma \circ Conc_\sigma = id_{D_\sigma^{ab}}$

Proof :

We prove this by induction over the type structure.

(*i*) This is true on base types by hypothesis.

(*ii*) This is true for the base case for abs_A by hypothesis and for Abs_A by Lemma 2.4.5.

(*iii*) $Abs_A(Conc_A(\bar{a})) = Abs_A(\bigsqcup\{T \mid Abs_A(T) \leqslant \bar{a}\})$

$= \bigsqcup(\mathbf{P}\,abs_A(\bigsqcup\{T \mid Abs_A(T) \leqslant \bar{a}\}))$

$= \bigsqcup(\bigcup(\mathbf{P}\,\mathbf{P}\,abs_A(\{T \mid Abs_A(T) \leqslant \bar{a}\})))$ (P8)

$= \bigsqcup(\bigcup(\{\mathbf{P}\,abs_A(T) \mid Abs_A(T) \leqslant \bar{a}\}^\circ))$ definition of P on morphisms

$= \bigsqcup(\bigcup\{\mathbf{P}\,abs_A(T) \mid Abs_A(T) \leqslant \bar{a}\})^\circ$

by a simple adaptation of a result in [Plotkin 1976] p. 477

Let $S = \bigcup\{\mathbf{P}\,abs_A(T) \mid Abs_A(T) \leqslant \bar{a}\}$. Since abs_A is onto (induction (*ii*)), there exists $a \in D_A^{st}$ such that $abs_A(a) = \bar{a}$, and so $a \in \mathbf{P}\,abs_A(\{\!|a|\!\})$ and is in S since $\bigsqcup \mathbf{P}\,abs_A(\{\!|a|\!\}) \leqslant \bar{a}$. Furthermore, $\bar{a}$ is the least upper bound of all the elements in S, and hence the result.

Assume that the three results hold for types σ and τ.

(*i*) Define $abs_{\sigma\to\tau}^{-1} = \lambda \bar{f}^{D_{\sigma\to\tau}^{ab}}.\, abs_\tau^{-1} \circ \bar{f} \circ abs_\sigma$. This is certainly continuous because it is the composition of continuous functions. Let $\bar{f} \in D_{\sigma\to\tau}^{ab}$. Then,

$$\begin{aligned}
abs_{\sigma\to\tau}(abs_{\sigma\to\tau}^{-1}(\bar{f})) &= abs_{\sigma\to\tau}(abs_\tau^{-1} \circ \bar{f} \circ abs_\sigma) \\
&= Abs_\tau \circ \mathbf{P}(abs_\tau^{-1} \circ \bar{f} \circ abs_\sigma) \circ Conc_\sigma \\
&= \bigsqcup \circ \mathbf{P}(abs_\tau \circ abs_\tau^{-1} \circ \bar{f} \circ abs_\sigma) \circ Conc_\sigma \qquad \text{(P3)} \\
&= \bigsqcup \circ \mathbf{P}(\bar{f} \circ abs_\sigma) \circ Conc_\sigma \qquad \text{by induction } (i) \\
&= \bigsqcup \circ \mathbf{P}\bar{f} \circ \mathbf{P}\, abs_\sigma \circ Conc_\sigma \qquad \text{(P3)} \\
&= \bar{f} \circ \bigsqcup \circ \mathbf{P}\, abs_\sigma \circ Conc_\sigma \qquad \text{since } \bar{f} \text{ is continuous}
\end{aligned}$$

$$\begin{aligned}
&\bar{f} \circ Abs_\sigma \circ Conc_\sigma \\
&= \bar{f} \qquad \text{by induction } (iii)
\end{aligned}$$

and so, by the principle of extensionality, $abs_{\sigma\to\tau} \circ abs_{\sigma\to\tau}^{-1} = id_{D_{\sigma\to\tau}^{st}}$

(*ii*) For any $\bar{f}$ in $D_{\sigma\to\tau}^{ab}$, choose $f \in D_{\sigma\to\tau}^{st}$ such that $f = abs_{\sigma\to\tau}^{-1}(\bar{f})$. Then $abs_{\sigma\to\tau}(f) = \bar{f}$ by induction (*i*), and so $abs_{\sigma\to\tau}$ is onto. $Abs_{\sigma\to\tau}$ is then onto by Lemma 2.4.5.

(*iii*) This follows as for the base case by putting the appropriate type subscripts on the abstraction and concretisation maps.

□

The following Lemma is often useful in the applications for which we will use this framework.

Lemma 2.4.7:

If abs_A is strict and bottom-reflexive for each base type A, then for all types σ,

$$Conc_\sigma(\bot_{D_\sigma^{ab}}) = \{\bot_{D_\sigma^{st}}\}.$$

Proof :

$$Conc_\sigma(\bot_{D_\sigma^{ab}}) = \biguplus \{T \mid Abs_\sigma(T) \leqslant \bot_{D_\sigma^{ab}}\}$$

$$= \biguplus \{T \mid Abs_\sigma(T) = \bot_{D_\sigma^{ab}}\}$$

$$= \biguplus \{\{\bot_{D_\sigma^{st}}\}\} \qquad \text{since from Lemma 2.4.4 } Abs_\sigma \text{ is bottom-reflexive}$$

$$= \{\bot_{D_\sigma^{st}}\}$$

□

2.5. Adjointness of Abs_σ and $Conc_\sigma$.

The maps Abs_σ and $Conc_\sigma$ are adjoined functions. This ensures that the abstract domain closely models the sets of elements of equal definedness in the standard interpretation.

Proposition 2.5.1:

Abs_σ and $Conc_\sigma$ are a pair of adjoined functions. i.e.

(*i*) $Conc_\sigma \circ Abs_\sigma \geqslant id_{\mathbf{P}D_\sigma^{st}}$.

(*ii*) $Abs_\sigma \circ Conc_\sigma \leqslant id_{D_\sigma^{ab}}$

Furthermore, if Abs_σ is onto for all σ, then

(*iii*) $Abs_\sigma \circ Conc_\sigma = id_{D_\sigma^{ab}}$.

Proof :

(*i*) Let $S \in \mathbf{P}D_\sigma^{st}$.

$$Conc_\sigma(Abs_\sigma(S)) = \biguplus \{T \mid Abs_\sigma(T) \leqslant Abs_\sigma(S)\}$$

and $Abs_\sigma(S) \leqslant Abs_\sigma(S)$, so $S \in \{T \mid Abs_\sigma(T) \leqslant Abs_\sigma(S)\}$. Hence the result follows since $\geqslant$ is just $\supseteq$ in the Hoare powerdomain.

(*ii*) This follows as in the proof of Lemma 2.4.6 (*iii*) until the second last line where we replace $=$ by $\leqslant$ because we do not insist on ontoness in the conditions of this proposition.

(*iii*) This is Lemma 2.4.6 (*iii*).

□

2.6. Semi-homomorphic Property of abs_σ and fix.

For each type σ we have that abs_σ is a semi-homomorphism of function application, that is, if $f \in D^{st}_{\sigma\to\tau}$ then

$$abs_{\sigma\to\tau}(f) \circ abs_\sigma \geqslant abs_\tau \circ f$$

or, in terms of elements, if $s \in D^{st}_\sigma$ then

$$abs_{\sigma\to\tau}(f)\,(abs_\sigma(s)) \geqslant abs_\tau(f(s))$$

We will need this to prove Theorem 2.7.1. It can be seen to be intuitively true by looking at the definition of $abs_{\sigma\to\tau}(f)(abs_\sigma(s))$:

$$abs_{\sigma\to\tau}(f)\,(abs_\sigma(s)) = \bigsqcup\{abs_\tau(f\,s') \mid abs_\sigma(s') \leqslant abs_\sigma(s)\}^\circ \qquad \text{Proposition 2.3.1}$$

We see that $abs_{\sigma\to\tau}(f)\,(abs_\sigma(s))$ applies f to all the values in D^{st}_σ which abstract to something less than or equal to what s abstracts to, some of which may give a more defined answer, and hence may be abstracted to a greater value than $f(s)$ would. As an example, we will be able to show later in Chapter 3, with the definedness interpretation, that

$$abs_{int\to int}(E^{st}\ [[\lambda x^{int}.\mathit{if}\ x = 5\ \mathit{then}\ \bot\ \mathit{else}\ 1]]\,\rho^{st}) = \lambda x^{D^{ab}_{int}}.x$$

and that

$$abs_{int}(5) = 1$$

and so the left-hand side of the above inequality is 1, while the right-hand side is

$$abs_{int}(E^{st}\ [[(\lambda x^{int}.\mathit{if}\ x = 5\ \mathit{then}\ \bot\ \mathit{else}\ 1)\ (5)]]\,\rho^{st}) = abs_{int}(\bot_{D^{st}_{int}}) = 0$$

We now state and prove that abs_σ is a semi-homomorphism of function application.

Proposition 2.6.1:

For all types σ, abs_σ is a semi-homomorphism of function application. i.e. if $f \in D^{st}_{\sigma\to\tau}$ then $abs_{\sigma\to\tau}(f) \circ abs_\sigma \geqslant abs_\tau \circ f$ (or in terms of elements, if $s \in D^{st}_\sigma$ then $abs_{\sigma\to\tau}(f)(abs_\sigma(s)) \geqslant abs_\tau(f(s))$).

Proof :

$$abs_{\sigma\to\tau}(f) \circ abs_\sigma = Abs_\tau \circ \mathbf{P} f \circ Conc_\sigma \circ abs_\sigma$$

$$= \bigsqcup \circ \mathbf{P}\, abs_\tau \circ \mathbf{P} f \circ Conc_\sigma \circ abs_\sigma \qquad \text{definition of } Abs_{\sigma\to\tau}$$

$$= \bigsqcup \circ \mathbf{P}\, abs_\tau \circ \mathbf{P} f \circ Conc_\sigma \circ \bigsqcup \circ \{\!|.|\!\} \circ abs_\sigma \qquad \text{(P10)}$$

$$= \bigsqcup \circ \mathbf{P}\, abs_\tau \circ \mathbf{P} f \circ Conc_\sigma \circ \bigsqcup \circ \mathbf{P}\, abs_\sigma \circ \{\!|.|\!\} \qquad \text{(P6)}$$

$$= \bigsqcup \circ \mathbf{P}\, abs_\tau \circ \mathbf{P} f \circ Conc_\sigma \circ Abs_\sigma \circ \{\!|.|\!\} \qquad \text{definition of } Abs_\sigma$$

$$\geqslant \bigsqcup \circ \mathbf{P}\, abs_\tau \circ \mathbf{P} f \circ \{\!|.|\!\} \qquad \text{Proposition 2.5.1 } (i)$$

$$= \bigsqcup \circ \mathbf{P}(abs_\tau \circ f) \circ \{\!|.|\!\} \qquad \text{(P3)}$$

$$= \bigsqcup \circ \{\!|.|\!\} \circ abs_\tau \circ f \qquad \text{(P6)}$$

$$= abs_\tau \circ f \qquad \text{(P10)}$$

□

As a consequence of the semi-homomorphic property of abstraction, and the fact that the abstraction maps are continuous, we have that *fix* is a semi-homomorphism of abstraction, which is also needed in the proof of Theorem 2.7.1.

Proposition 2.6.2:

fix is a semi-homomorphism of abstraction. i.e.

$$fix \circ abs_{\sigma\to\sigma} \geqslant abs_\sigma \circ fix$$

Proof :

Let $f \in D^{St}_{\sigma\to\sigma}$, and let h_i be the approximations to $fix(abs_{\sigma\to\sigma}(f))$ and f_i be the approximations to $fix(f)$. Then $h_0 = \bot_{D^{St}_\sigma} = abs_\sigma(\bot_{D^{St}_\sigma})$ (since we have insisted that the abstraction maps on base types are strict, and by Lemma 2.2.8 (*iii*) the abstraction maps for all types are strict) $= abs_\sigma(f_0)$. Assume that $h_k \geqslant abs_\sigma(f_k)$ for all $k \leqslant i$. Then

$$h_{i+1} = (abs_{\sigma\to\sigma}(f))(h_i)$$

$$\geqslant (abs_{\sigma\to\sigma}(f))(abs_\sigma(f_i)) \qquad \text{induction hypothesis and monotonicity of } abs_{\sigma\to\sigma}$$

$$\geqslant abs_\sigma(f(f_i)) \qquad \text{Proposition 2.6.1}$$

$$= abs_\sigma(f_{i+1})$$

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

$$\begin{aligned} fix(abs_{\sigma\to\sigma}(f)) &= \bigsqcup\{h_i\} \\ &\geqslant \bigsqcup\{abs_\sigma(f_i)\}) && \text{by above induction} \\ &= abs_\sigma(\bigsqcup\{f_i\}) && \text{since } abs_\sigma \text{ is continuous and } \{f_i\} \text{ is directed} \\ &= abs_\sigma(fix(f)) \end{aligned}$$

□

2.7. A Result Relating the Abstract and Standard Interpretations.

The following result is crucial for proving the correctness of the framework for abstract interpretation(c.f. for example [Nielson 1984 Theorem 3.3:14).

Theorem 2.7.1:

Suppose that we have that $E^{ab}\,[[c_\sigma]]\,\rho^{ab} \geqslant abs_\sigma(E^{st}\,[[c_\sigma]]\,\rho^{st})$ for all constants c_σ. Then for all $\rho^{st} \in Env^{st}$, $\rho^{ab} \in Env^{ab}$ such that for all x^τ, $\rho^{ab}(x^\tau) \geqslant abs_\tau(\rho^{st}(x^\tau))$, we have for all $e : \sigma$:

$$E^{ab}\,[[e]]\,\rho^{ab} \geqslant abs_\sigma(E^{st}\,[[e]]\,\rho^{st})$$

Proof :

We prove this by structural induction over the terms in our language (see section 1.5.1).

(1) $E^{ab}\,[[x^\sigma]]\,\rho^{ab} = \rho^{ab}(x^\sigma)$

$\geqslant abs_\sigma(\rho^{st}(x^\sigma))$ condition of Theorem

$= abs_\sigma(E^{st}\,[[x^\sigma]]\,\rho^{st})$

(2) $E^{ab}\,[[c_\sigma]]\,\rho^{ab} \geqslant abs_\sigma(E^{st}\,[[c_\sigma]]\,\rho^{st})$ by condition of Theorem

(3) Let $\bar{s} \in D_\sigma^{ab}$. Then

$$(E^{ab}\,[[\lambda x^\sigma.e]]\,\rho^{ab})\,\bar{s} = (\lambda \bar{y}^{D_\sigma^{ab}}.E^{ab}\,[[e]]\,\rho^{ab}[\bar{y}/x^\sigma])\,\bar{s}$$

$$= E^{ab}\,[[e]]\,\rho^{ab}[\bar{s}/x^\sigma]$$

and

$$(abs_{\sigma\to\tau}(E^{st}\ [[\lambda x^{\sigma}.e]]\ \rho^{st}))\ \bar{s} = (\lambda \bar{s}^{D_{\sigma}^{ab}}.\bigsqcup\{abs_{\tau}((E^{st}\ [[\lambda x^{\sigma}.e]]\ \rho^{st})\ s) \mid abs_{\sigma}(s)\leqslant \bar{s}\}^{\circ})\ \bar{s}$$

$$= \bigsqcup\{abs_{\tau}((\lambda y^{D_{\sigma}^{st}}.E^{st}\ [[e]]\ \rho^{st}[y/x^{\sigma}])\ s) \mid abs_{\sigma}(s)\leqslant \bar{s}\}^{\circ}$$

$$= \bigsqcup\{abs_{\tau}(E^{st}\ [[e]]\ \rho^{st}[s/x^{\sigma}]) \mid abs_{\sigma}(s)\leqslant \bar{s}\}^{\circ}$$

Now $\rho^{ab}[\bar{s}/x^{\sigma}]$ and $\rho^{st}[s/x^{\sigma}]$ still satisfy the conditions on the environment since $abs_{\sigma}(s) \leqslant \bar{s}$, and so by the induction hypothesis, every element in the set $\{abs_{\tau}(E^{st}\ [[e]]\ \rho^{st}[s/x^{\sigma}]) \mid abs_{\sigma}(s)\leqslant \bar{s}\}$ approximates $E^{ab}\ [[e]]\ \rho^{ab}[\bar{s}/x^{\sigma}]$ and hence the required result holds (by the definition of the *least* upper bound).

(4) $E^{ab}\ [[e_1\ e_2]]\ \rho^{ab} = E^{ab}\ [[e_1]]\ \rho^{ab}\ E^{ab}\ [[e_2]]\ \rho^{ab}$

$\geqslant abs_{\sigma\to\tau}(E^{st}\ [[e_1]]\ \rho^{st})(abs_{\sigma}(E^{st}\ [[e_2]]\ \rho^{st}))$ induction hypothesis

$\geqslant abs_{\tau}(E^{st}\ [[e_1]]\ \rho^{st}\ E^{st}\ [[e_2]]\ \rho^{st})$ Proposition 2.6.1

$= abs_{\tau}(E^{st}\ [[e_1\ e_2]]\ \rho^{st})$

(5) $E^{ab}\ [[fix\ e]]\ \rho^{ab} = fix(E^{ab}\ [[e]]\ \rho^{ab})$

$\geqslant fix(abs_{\sigma\to\sigma}(E^{st}\ [[e]]\ \rho^{st}))$ induction hypothesis

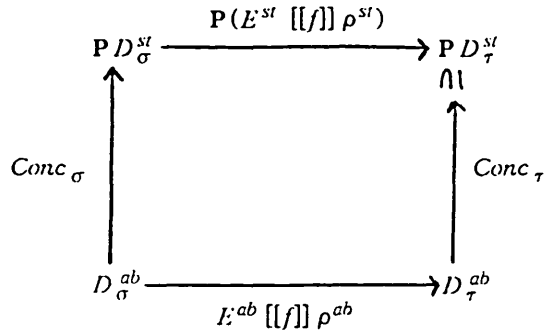
$\geqslant abs_{\sigma}(fix(E^{st}\ [[e]]\ \rho^{st}))$ Proposition 2.6.2

$= abs_{\sigma}(E^{st}\ [[fix\ e]]\ \rho^{st})$

□

2.8. Correctness of Abstract Interpretation

Here we prove the correctness of abstract interpretation. We prove it as a corollary of the fact that the following safety diagram (introduced in section 2.1) holds :



Theorem 2.8.1:

The above diagram holds. i.e. for environments satisfying the conditions of Theorem 2.7.1, if $f : \sigma \to \tau$ then

$$Conc_\tau \circ (E^{ab}\ [[f]]\ \rho^{ab}) \supseteq \mathbf{P}\ (E^{st}\ [[f]]\ \rho^{st}) \circ Conc_\sigma$$

Proof :

$$Conc_\tau \circ (E^{ab}\ [[f]]\ \rho^{ab}) \supseteq Conc_\tau \circ abs_{\sigma\to\tau}(E^{st}\ [[f]]\ \rho^{st})$$

Theorem 2.7.1 and monotonicity of $Conc_\tau$

$$\supseteq Conc_\tau \circ Abs_\tau \circ \mathbf{P}(E^{st}\ [[f]]\ \rho^{st}) \circ Conc_\sigma \quad \text{definition of } abs_{\sigma\to\tau}$$

$$\supseteq \mathbf{P}(E^{st}\ [[f]]\ \rho^{st}) \circ Conc_\sigma \quad \text{Proposition 2.5.1 } (i)$$

□

Theorem 2.8.2: (Correctness Theorem for Abstract Interpretation)

The abstract interpretation we have developed is correct. That is, given $f : \sigma \to \tau$, environments satisfying the conditions of Theorem 2.7.1, and interpretations of constants satisfying the conditions of Theorem 2.7.1, we have that if $\bar{s} \in D^{ab}_\sigma$ and $(E^{ab}\ [[f]]\ \rho^{ab})\ (\bar{s}) = \bar{t}$ then for all $s \in Conc_\sigma(\bar{s})$, $(E^{st}\ [[f]]\ \rho^{st})\ (s) \in Conc_\tau(\bar{t})$.

Proof :

This is a direct corollary of Theorem 2.8.1.

$$Conc_\tau(\bar{t}) = Conc_\tau((E^{ab}\ [[f]]\ \rho^{ab})\ (\bar{s})) \quad \text{by hypothesis}$$

$$\supseteq \mathbf{P}(E^{st}\ [[f]]\ \rho^{st})\ (Conc_\sigma(\bar{s})) \quad \text{Theorem 2.8.1}$$

$$= \{(E^{st}\ [[f]]\ \rho^{st})\ (s) \mid s \in Conc_\sigma(\bar{s})\}^\circ \quad \text{definition of } \mathbf{P} \text{ on morphisms}$$

and hence the result.

□

2.9. Context-free and Context-sensitive Issues

In the world of first-order functions and strictness analysis, the information about arguments to functions is true in all applications of a function. When we move from the domain of first-order functions to higher-order functions, or to using more complex analyses than strictness analysis, we no longer have that the information about a function is constant in all applications of that function. In the case of a higher-order function for instance, the information can vary according to the information about a functional parameter. Consider the function

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

which has abstract interpretation

$$\lambda f^{D_A^{ab} \to D_A^{ab}}.\lambda x^{D_A^{ab}}.f(x).$$

Clearly any information in an application of g is going to depend on the information given by its first argument in an application.

In our application of the theory in the next chapter, we will label arguments of a function to indicate how much evaluation it is safe to do of the arguments. From the above example, it may be desirable to try and carry around information which said, for example, that if a function was given as a parameter a function which was strict in its argument, then the other function was strict in another of its parameters. This would mean that definedness information would have to be available at run-time, and that some of the interpretation of the abstract interpretation would also have to occur at run-time, and we would rather try and avoid the problems this causes.

An attempt to solve this problem was made in [Burn, Hankin and Abramsky 1985a], where it was suggested that apply nodes be labelled with information rather than the arguments to functions, and so the abstract interpretation could take into account the contextual information of the function application. Thus, in an application

$$g\ h\ e$$

of the above function g, we could take into account the information about h in determining information about the second parameter to g in this application. It is shown in [Hankin, Burn and Peyton Jones, 1986] that in the case of strictness analysis this gives

more information than if we had have just labelled the arguments to functions with strictness information. We term such information *context-sensitive*.

However, the evaluation of functional programs dynamically creates function applications which were not present in the program text, and so could not be analysed statically using abstract interpretation. For example, the expression

$$(\mathit{if\ condition\ then}\ f_1\ \mathit{else}\ f_2)\ e$$

will create the application $f_1\ e$ or $f_2\ e$ depending on the truth of the *condition*. As a first approximation to the solution of the problem of finding out information about such applications, we can determine the strictness information about the arguments to a function which is true in any application, and so is *context-free* information. Clearly context-sensitive information will be stronger than context-free information for applications that appear explicitly in the program.

We will discuss these issues more fully in the context of a particular abstract interpretation in section 3.4. It will be shown that each type of information gives something which the other lacks, and so each is indispensable. The following two theorems which allow us to test respectively for context-free and context-sensitive information.

Theorem 2.9.1: (Context-Free Information Theorem)

If $f : \sigma_1 \rightarrow \cdots \sigma_n \rightarrow \tau$ and

$$(E^{ab}\ [[f]]\ \rho^{ab})\ \top_{D^{ab}_{\sigma_1}} \cdots \top_{D^{ab}_{\sigma_{i-1}}} \overline{s_i}\ \top_{D^{ab}_{\sigma_{i+1}}} \cdots \top_{D^{ab}_{\sigma_n}} = \overline{t}$$

then for all $e_j : \sigma_j$, $j \neq i$, for all $s_i \in Conc_{\sigma_i}(\overline{s_i})$, we have

$$E^{st}\ [[f]]\ \rho^{st}\ E^{st}\ [[e_1]]\ \rho^{st} \cdots E^{st}\ [[e_{i-1}]]\ \rho^{st}\ s_i\ E^{st}\ [[e_{i+1}]]\ \rho^{st} \cdots E^{st}\ [[e_n]]\ \rho^{st} \in Conc_\tau(\overline{t})$$

Proof :

$$\overline{t} = (E^{ab}\ [[f]]\ \rho^{ab})\ \top_{D^{ab}_{\sigma_1}} \cdots \top_{D^{ab}_{\sigma_{i-1}}} \overline{s_i}\ \top_{D^{ab}_{\sigma_{i+1}}} \cdots \top_{D^{ab}_{\sigma_n}}$$

$$\geqslant E^{ab}\ [[f]]\ \rho^{ab}\ E^{ab}\ [[e_1]]\ \rho^{ab} \cdots E^{ab}\ [[e_{i-1}]]\ \rho^{ab}\ \overline{s_i}\ E^{ab}\ [[e_{i+1}]]\ \rho^{ab} \cdots E^{ab}\ [[e_n]]\ \rho^{ab}$$

since $\top_{D^{ab}_{\sigma_j}} \geqslant E^{ab}\ [[e_j]]\ \rho^{ab}$ for all $j \neq i$.

$$= (\lambda \bar{x}^{D^{ab}_{\sigma_i}}\ E^{ab}\ [[f]]\ \rho^{ab}\ E^{ab}\ [[e_1]]\ \rho^{ab} \cdots E^{ab}\ [[e_{i-1}]]\ \rho^{ab}\ \bar{x}\ E^{ab}\ [[e_{i+1}]]\ \rho^{ab}$$

$\cdots\ E^{ab}\ [[e_n]]\ \rho^{ab})\ \overline{s_i}$

where $\bar{x}^{D^{ab}_{\sigma_i}}$ is a new variable

$= (\lambda \bar{x}^{D^{ab}_{\sigma_i}}.E^{ab}\ [[f]]\ \rho^{ab}[\bar{x}/x^{\sigma_i}]\ E^{ab}\ [[e_1]]\ \rho^{ab}[\bar{x}/x^{\sigma_i}]\ \cdots\ E^{ab}\ [[e_{i-1}]]\ \rho^{ab}[\bar{x}/x^{\sigma_i}]$

$E^{ab}\ [[x^{\sigma}]]\ \rho^{ab}[\bar{x}/x^{\sigma_i}]\ E^{ab}\ [[e_{i+1}]]\ \rho^{ab}[\bar{x}/x^{\sigma_i}]\ \cdots\ E^{ab}\ [[e_n]]\ \rho^{ab}[\bar{x}/x^{\sigma_i}])\ \overline{s_i}$

$= (\lambda \bar{x}^{D^{ab}_{\sigma_i}}.E^{ab}\ [[f\ e_1\ \cdots\ e_{i-1}\ x^{\sigma_i}\ e_{i+1}\ \cdots\ e_n]]\ \rho^{ab}[\bar{x}/x^{\sigma_i}])\ \overline{s_i}$

$= (E^{ab}\ [[\lambda x^{\sigma_i}.f\ e_1\ \cdots\ e_{i-1}\ x^{\sigma_i}\ e_{i+1}\ \cdots\ e_n]]\ \rho^{ab})\ \overline{s_i}$

=> for all $s_i \in Conc_{\sigma_i}(\overline{s_i})$,

$(E^{st}\ [[\lambda x^{\sigma_i}.f\ e_1\ \cdots\ e_{i-1}\ x^{\sigma_i}\ e_{i+1}\ \cdots\ e_n]]\ \rho^{st})\ (s_i) \in Conc_{\tau}(\overline{t})$

by a slight adaptation of Theorem 2.8.2. possible since $Conc_{\sigma_i}$ is continuous

i.e. $(\lambda x^{D^{st}_{\sigma_i}}.E^{st}\ [[f\ e_1\ \cdots\ e_{i-1}\ x^{\sigma_i}\ e_{i+1}\ \cdots\ e_n]]\ \rho^{st})\ (s_i) \in Conc_{\tau}(\overline{t})$

i.e. $(\lambda x^{D^{st}_{\sigma_i}}.E^{st}\ [[f]]\ \rho^{st}\ d_1\ \cdots\ d_{i-1}\ x^{\sigma_i}\ d_{i+1}\ \cdots\ d_n)\ (s_i) \in Conc_{\tau}(\overline{t})$

i.e. $E^{st}\ [[f]]\ \rho^{st}\ d_1\ \cdots\ d_{i-1}\ s_i\ d_{i+1}\ \cdots\ d_n \in Conc_{\tau}(\overline{t})$

□

Theorem 2.9.2: (Context-Sensitive Information Theorem)

Given $f : \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \tau$ and an application $f\ e_1\ \cdots\ e_n : \tau$, if

$E^{ab}\ [[f]]\ \rho^{ab}\ E^{ab}\ [[e_1]]\ \rho^{ab}\ \cdots E^{ab}\ [[e_{i-1}]]\ \rho^{ab}\ \overline{s_i}\ E^{ab}\ [[e_{i+1}]]\ \rho^{ab}\ \cdots\ E^{ab}\ [[e_n]]\ \rho^{ab} = \overline{t}$

then for all $s_i \in Conc_{\sigma_i}(\overline{s_i})$

$E^{st}\ [[f]]\ \rho^{st}\ E^{st}\ [[e_1]]\ \rho^{st}\ \cdots E^{st}\ [[e_{i-1}]]\ \rho^{st}\ s_i\ E^{st}\ [[e_{i+1}]]\ \rho^{st}\ \cdots\ E^{st}\ [[e_n]]\ \rho^{st} \in Conc_{\tau}(\overline{t})$

Proof :

The proof of this theorem follows exactly as in the proof of Theorem 2.9.1, except that the first two lines are replaced by the following line :

$$\bar{t} = E^{ab}[[f]]\,\rho^{ab}\,E^{ab}[[e_1]]\,\rho^{ab}\,\cdots E^{ab}[[e_{i-1}]]\,\rho^{ab}\,\overline{s_i}\,E^{ab}[[e_{i+1}]]\,\rho^{ab}\,\cdots\,E^{ab}[[e_n]]\,\rho^{ab}$$

i.e. we have equality instead of the inequality on the second line of the other proof.

□

2.10. The Undual Duality.

Those familiar with category theory may have noticed something most tantalising while reading through this chapter. In section 2.1 we made a fundamental decision about what we meant by the correctness of an abstract interpretation, and said that it was correct if the result in the abstract interpretation represented all of the possible results in the standard interpretation. This meant that a result in the abstract interpretation may represent considerably more values than were really possible in the standard interpretation. A related decision was the choosing of the Hoare powerdomain and the least upper bound operator in defining the abstraction maps for higher types.

The point is that most of these things have a dual; the dual to making sure that the result represents all possible values is to ensure that it represents a subset of the values; the dual to the least upper bound is the greatest lower bound; in some ways the Smyth powerdomain [Smyth 1978] is a dual to the Hoare powerdomain.

Working in a monotone framework, Abramsky [Abramsky 1985b] was able to develop a termination analysis which was a dual of the strictness analysis he presents. Because all of the abstract domains are finite, the interpretation is computable even though the abstraction are not continuous. However, it was not possible to raise the termination analysis to a continuous world, and the sets resulting from the obvious induced concretisation map are not in general elements of the Smyth powerdomain.

We can get some intuition about why this is true by considering the following example. For each base type A, we define D_A^{ab} to be the two point domain $2 = \{0,1\}$ with $0 \leqslant 1$, and define

$$abs_A : D_A^{st} \to D_A^{ab}$$

by

$$abs_A(a) = \begin{cases} 0 \text{ if } a = \bot_{D_A^{st}} \\ 1 \text{ otherwise} \end{cases}$$

The concretisation map we have given for safety interpretations (Definition 2.2.2) gives the following interpretations to the values in the two point domain :

$$Conc_A(0) = \{\perp_{D_A^{st}}\}$$

$$Conc_A(1) = D_A^{st}$$

which says intuitively that 0 represents the idea of definite undefinedness and 1 represents the possibility that something may be defined.

Another way to interpret this domain is to let 1 represent the idea of definite definedness and 0 represent the possibility that something may be undefined, defining :

$$Conc'_A(0) = D_A^{st}$$

$$Conc'_A(1) = D_A^{st} - \{\perp_{D_A^{st}}\}$$

But here is the problem - the set which we have given for the concretisation of 1 is not a member of the Smyth powerdomain! To handle analyses such as these, a special powerdomain has been developed [Mycroft and Nielson 1983], [Nielson 1984], but has only been applied to a first-order framework.

2.11. Relationship to Other Work.

Alan Mycroft [Mycroft 1981] was the first to apply abstract interpretation to functional languages, where he developed a framework for abstract interpretation of first order functional languages over base types with flat domains as standard interpretations of the types.

There are five other higher-order frameworks that we are aware of. We mention firstly the theory presented in [Burn, Hankin and Abramsky 1985a] and [Hankin, Burn and Peyton Jones 1986] to which this work is the closest, being basically a generalisation of the work to allow abstract domains which are more complex than the two point domain.

Two other frameworks [Maurer 1985] and [Mycroft and Jones 1985] define abstract domains which model the standard interpretation of the type

$$D \cong A + D \rightarrow D,$$

although we note that [Maurer 1985] also has products and lifting in the above equation. This is in direct contrast to our philosophy of using the type structure of the language in a strong way to allow us to have a computable and as accurate an abstract interpretation

as possible. It is notable that the framework of [Maurer 1985] is only designed to cope with strictness analysis.

A further point worth noting is that in our framework, we have parameterised only the interpretation of the types and the constants, for this is what we need for our applications. However, in [Mycroft and Jones 1985] the meanings of λ-abstraction and application are also parameterised, so making a more general framework. The work is based on logical relations [Plotkin 1980] rather than domain theory.

In [Hudak and Young 1985] the abstract interpretation really is a non-standard semantics, returning for each function a set of variables in which it is strict. The first-order case is equivalent to the first-order case of our work in that it explicitly calculates the set of variables in which a function is strict rather than calculating its characteristic function. To extend the work to higher-order functions, they introduce *strictness ladders*, where the ith element in the strictness ladder seems to give strictness information about the first i arguments given that the function has been applied to i arguments. Their main problem is that they are dealing in a pseudo-untyped framework. Since it is impossible to get finite answers for functions with non-finite type, the framework could probably be reworked to take into account the type information and thus do away with strictness ladders.

The final higher-order framework is due to Abramsky [Abramsky 1985b] and is based on logical relations rather than domain theoretic ideas. Two dual analyses, *safety* and *liveness*, are developed in a monotone framework. In particular, they are applied to developing respectively a strictness and a termination analysis. Furthermore, conditions are proved which show when the analysis can be raised to the world of continuous functions.

We mention finally the work of Nielson [Nielson 1984, 1986a, 1986b]. Giving an interpretation of a language can be viewed as a two step process, firstly translating the language into some standard "meta-language" (for example, the λ-calculus with constants), and then giving an interpretation of the meta-language. In [Nielson 1984], which is overviewed in [Nielson 1986a], a meta-language is presented which is powerful enough to give the denotational definition of most programming languages. It cannot however express the denotation of languages with storable procedures.

The outstanding feature of the meta-language is that it has a two-level type system, with the intuition that the top-level types represent the type of compile-time objects and run-time objects have bottom-level types. In using the framework, it is the interpretation of the bottom-level types that is changed, while the interpretation of the top-level types is fixed in all interpretations.

To apply the framework to our problem of testing the definedness of functions expressed in the λ-calculus with constants, we note that we wish to change the interpretation of all of the base types in the language, and so these would be made into bottom-level types. Their interpretation would then be allowed to change.

Due to a technical restriction on the constants allowed in the framework of [Nielsen 1984, 1986a], namely that they be *contravariantly pure*, it was not powerful enough to do the definedness interpretation which we present in the next chapter. Basically a type is contravariantly pure if there was a bottom-level type as the first argument to the function space constructor. Thus, all our higher-order constants like the conditional could not be treated in the framework. This situation has been remedied in [Nielson 1986b] where the restriction of contravariant purity has been weakened, and the resulting theory is applied to proving the strictness result of [Burn, Hankin and Abramsky 1985a], as well as the results in [Mycroft and Jones 1985]. When reduced to solving the problem of strictness analysis, the framework looks very much like that presented in [Abramsky 1985b].

2.12. Conclusion

Abstract interpretation is used to provide a computable method of discovering information of interest about a program without actually having to run the program. In this chapter we have introduced a framework for safe abstract interpretations of the typed lambda-calculus with a set of base types and a set of typed constants.

Correctness of abstract interpretation is an important notion, and we began the chapter by arguing what we meant by correctness. Intuitively, we said that an abstract interpretation was correct if the answer to a calculation in the abstract interpretation represented all of the possible answers that could have occurred in the standard interpretation. From this we moved onto motivating the various abstraction and concretisation maps, and their forms, that arise naturally.

A correct framework was developed which required three things from the "user" of the framework. Firstly an abstract domain for each of the base types must be provided which is a finite, complete lattice. The lattice must be complete because we need to be able to take least upper bounds; it must be finite because we need to take fixed points, which requires the testing of functions at all possible argument values, and finiteness implies effectiveness. The abstract domains for higher types are then given because we have said that $D^{ab}_{\sigma\to\tau}$ is just the set of continuous functions $D^{ab}_{\sigma} \to D^{ab}_{\tau}$.

Secondly, a strict, continuous abstraction map must be provided, which maps the

standard interpretation of the base types to the abstract interpretation of the base types. The abstract domain and the abstraction map should be chosen so they make the distinctions on the base types in which the user is interested. In Chapter 3 we will see that the property we are interested in is how much evaluation is safe for arguments of functions, and so we choose an abstract domain which models the way that the various evaluators divide up the elements of the standard domains. Given such abstraction maps, the framework defines an abstraction map abs_σ for each type σ, which is the best possible abstraction map in the sense that it loses as little information as possible while being safe.

Finally, abstract interpretations of the typed constants must be provided which satisfy :

$$E^{ab}\,[[c_\sigma]]\,\rho^{ab} \geqslant abs_\sigma(E^{st}\,[[c_\sigma]]\,\rho^{st})$$

In Chapter 3 we will derive abstract interpretations of our constants as abstractions of the standard interpretation of the constants, and so the above condition will automatically hold.

The framework we have provided is proved correct. It is necessary that we are able to determine context-free information and context-sensitive information about a function, where the former is information true in the context of any application while the latter is true about a particular function application. Our two final theorems give practical tests for both of these types of information.

Chapter 3

A Definedness Interpretation and its Application to Changing Evaluation Strategies

The framework of Chapter 2 is applied in developing and proving correct an abstract interpretation which gives the definedness levels of functions. This is used to show how we can safely change the evaluation strategy for a functional program.

Abstract domains will be developed for the base types by considering the definedness level of elements of the standard interpretation of the base types and the way that different evaluators divide up the domain in that they preserve some elements of the domain and not others.

We determine abstract interpretations for the constants of our language by deriving them as the abstraction of the standard interpretation of the constants. This gives us the best possible approximation to constants while still staying within the limits set by the theory of Chapter 2.

The correctness of the abstract interpretation follows as a corollary of the correctness of the framework of Chapter 2. We discuss with the specific example of this interpretation how context-free and context-sensitive issues arise, and give two mechanical tests to find this information using the abstract interpretation that has been developed. Again these are just corollaries of the corresponding theorems in Chapter 2.

Using the abstract interpretation, we are able to show when it is safe to change evaluation strategies by doing some evaluation of the arguments to a function, either in parallel with the function application in a parallel system or before the function application in a sequential system.

The abstract domains we use in this chapter - the two point domain and the four point domain - are due respectively to Mycroft [Mycroft 1981] and Wadler [Wadler 1985]. The abstract interpretations for list constants are also due to Wadler, however this chapter contains the first proofs of their correctness. Simplification rules for expressions involving the abstract interpretations of these constants are also new to this work. Abstract interpretations for the other constants have appeared elsewhere, for example [Mycroft 1981] and [Burn, Hankin and Abramsky 1985a], with proofs of their correctness appearing in the latter.

3.1. Abstraction of Base Domains and Properties of Abstraction Maps.

Choosing an abstract domain and abstraction functions which capture the properties of interest seems to be one of the hardest things in abstract interpretation. We have two intuitions about the way abstract domains can be defined for the base types in our application. Firstly, since we are looking for information about the definedness of functions, we consider what abstract domains we might develop by looking at the definedness structure of the domains which are the standard interpretation of the base types. However, since we really want to use the abstract interpretation to change evaluation mechanisms by allowing some arguments to functions to be evaluated, and our argument in section 1.3 said that this requires seeing which elements of the standard interpretation a function preserves, we look at which elements the various evaluators preserve. It turns out that both intuitions lead to the same abstract domains, namely the two point domain of [Mycroft 1981] for flat base domains and the four point domain of [Wadler 1985] for list data types.

3.1.1. Defining Abstract Domains and Abstraction Maps From the Definedness Structure of the Standard Interpretation.

Let us first consider the case of a flat domain such as the standard interpretation of integers and booleans. Here there are two levels of definedness - either something is undefined (i.e. bottom), or it is totally defined. Thus, to capture the definedness information for such domains, we can define a simple abstract domain and abstraction map. Denoting the type by A, we have that D_A^{st} is a flat domain, and we can define $D_A^{ab} = 2$, where $2 = \{0,1\}$, with $0 \leqslant 1$. Then we define

$$abs_A : D_A^{st} \rightarrow D_A^{ab}$$

$$abs_A(a) = \begin{cases} 0 \text{ if } a = \perp_{D_A^{st}} \\ 1 \text{ otherwise} \end{cases}$$

We have chosen *Alists*, that is, lists of elements of type A as our prototypical example of a domain which is an infinite sum of products. Initially, we will simplify the discussion by restricting ourselves to the case where the standard interpretation of A is a flat domain so that, for example, we are considering list of integers or lists of booleans. The type of *Alist* is given by the recursive type equation

$$Alist \cong 1 + A \times Alist$$

(where 1 is the one-point domain, + is separated sum and $\times$ is cartesian product)

and the standard interpretation, as we mentioned in section 1.5.2, is obtained by solving this over the category of domains [Smyth and Plotkin 1982] to give

$$D^{st}_{Alist} = D^{st}_{A}{}^{*}.nil \cup D^{st}_{A}{}^{*}.\perp_{D^{st}_{Alist}} \cup D^{st}_{A}{}^{\omega}.$$

How might we go about defining an abstract domain which captures the definedness of elements of D^{st}_{Alist}? At first sight we may think of using the fact, since we are for the moment assuming that the elements of the list are from a flat domain, that for each element of the list, we have two levels of definedness - either the element is bottom or it is not - and so we could represent the strictness information for a list as a list of 0's and 1's. The theoretical development of Chapter 2 required that we have a finite, complete lattice as the abstract domain for any base type, so clearly we cannot have an abstract domain which contains a list of 0's and 1's for every list in the standard domain and anyway, such an abstract domain would be useless because we could never do all of the calculations required! A way to obtain finiteness of the domain would be to choose some n and have only lists of length at most n in the abstract domain. However, this is only ever going to give us information about the behaviours of a function on the first n elements of a list. Furthermore, the abstract domain would have 2^n elements in it, and so it would be computationally very expensive to find out any information about functions using it.

The main problem with the above abstract domain is that it treats the elements of a list in a way which is not typical of the way we use lists in programs; the first n elements of the list are treated differently from the rest of the elements of the list, whereas lists are used most naturally in the case that we wish to treat all of the elements in a uniform way.

If we ignore infinite lists for a moment, then we notice that we can divide up the standard domain for lists into four natural subsets :

(*i*) the set containing only $\perp_{D^{st}_{Alist}}$ i.e. $\{\perp_{D^{st}_{Alist}}\}$;

(*ii*) the set containing all of the partial lists plus $\perp_{D^{st}_{Alist}}$ i.e. $D^{st}_{A}{}^{*}.\perp_{D^{st}_{Alist}}$;

(*iii*) the set containing finite lists with (at least one bottom element), the partial lists and $\perp_{D^{st}_{Alist}}$ i.e. $D^{st}_{A}{}^{*}.\perp_{D^{st}_{Alist}} \cup D^{st}_{A}{}^{*}.\perp_{D^{st}_{A}}.D^{st}_{A}{}^{*}.nil$; and

(*iv*) the set containing all lists i.e. D^{st}_{Alist}.

In going from one subset to the next, we are adding elements which are more defined in some way; in going from (*i*) to (*ii*), we have added lists which have at least one element in them, but still have an undefined tail after a finite number of elments from D^{st}_{A}, and the

extra elements added in (*iii*) represent becoming more defined by replacing the undefined tail by *nil*, and the final addition in (*iv*) adds all of the finite lists which have no bottom elements.

We can define an abstract domain and abstraction map which captures these various levels of definedness. Let us use the four distinct elements $\perp_L$, I, F and $\top_L$, ordered by $\perp_L \leqslant I \leqslant F \leqslant \top_L$, as our abstract domain D^{ab}_{Alist}, and define

$$abs_{Alist} : (D^{st}_{Alist} - D^{st\,\omega}_{A}) \to D^{ab}_{Alist}$$

$$abs_{Alist}(L) = \begin{cases} \perp_L \text{ if } L = \perp_{D^{st}_{Alist}} \\ I \text{ if } L \in D^{st}_{A}.D^{st}_{A}{}^{*}.\perp_{D^{st}_{Alist}} \\ F \text{ if } L \in D^{st}_{A}.\perp_{D^{st}_{A}}.D^{st}_{A}{}^{*}.nil \\ \top_L \text{ if } L \in (D^{st}_{A} - \{\perp_{D^{st}_{A}}\})^{*}.nil \end{cases} \qquad (\dagger)$$

Moving from one level in the abstract interpretation to the next corresponds to being an element in the standard interpretation of *Alist* which is in one of the sets (*i*) to (*iv*) above but not in any proper subset of that set.

We can add infinite lists into the above discussion by noting that we need to have a continuous abstraction map for each base type. Since all of the approximations to an infinite list (i.e. partial lists) are abstracted to I, we must have that the abstraction of any infinite list is also I by the definition of continuity. Thus we finish the definition of the abstraction map for *Alist* :

$$abs_{Alist} : D^{st}_{Alist} \to D^{ab}_{Alist}$$

$$abs_{Alist}(L) = \begin{cases} \perp_L \text{ if } L = \perp_{D^{st}_{Alist}} \\ I \text{ if } L \in D^{st}_{A}.D^{st}_{A}{}^{*}.\perp_{D^{st}_{Alist}} \cup D^{st\,\omega}_{A} \\ F \text{ if } L \in D^{st}_{A}.\perp_{D^{st}_{A}}.D^{st}_{A}{}^{*}.nil \\ \top_L \text{ if } L \in (D^{st}_{A} - \{\perp_{D^{st}_{A}}\})^{*}.nil \end{cases}$$

It is worth noting that with this domain we can see the dual reading of the elements in the abstract domain which was introduced in section 2.1. From the point of view of

(†) The elements $\perp_L$, I, F and $\top_L$ correspond respectively to *BOT*, *INF*, *BOT−MEM* and *TOP−MEM* of [Wadler 1985]. They just form the four point domain {0,1,2,3}, but we have given them different names so that the first two elements are not confused with the elements of the two point domain. T stands for "Infinite" and F for "Finite" (with at least one ⊥ element).

the map abs_{Alist}, an element in the abstract domain represents a single list. However, if one was to take the view of the map $Conc_{Alist}$, then the elements $\perp_L$ to $\top_L$ represent respectively the sets (*i*) to (*iv*) above.

Before discussing the development of abstract domains using intuitions from the various sensible levels of evaluation, we note that the definedness interpretation of the type $\sigma \to \tau$ is induced as in section 1.5.2 by

$$D^{ab}_{\sigma\to\tau} = D^{ab}_{\sigma} \to D^{ab}_{\tau}$$

3.1.2. Defining Abstract Domains and Abstraction Maps From the Sensible Levels of Evaluation.

In section 1.3 we discussed intuitively that we needed to find out when a function was defined so that any evaluation strategy made sure it preserved the arguments to a function where the function was defined. To see whether it is safe to use a particular evaluator for an argument to a function, we must check that the function really is undefined for the elements which the evaluator does not preserve, and so the evaluator preserves all of the elements it is supposed to preserve.

Each of the evaluators divides up the standard interpretation of a type into two subsets, those elements which it does preserve and those which it does not. Definition 1.3.3 defines the evaluators we consider according to the elements they preserve. Since ξ_0 represents doing no evaluation of an expression, and we wish to see if we can change the evaluation strategy to do some evaluation of expressions which are arguments to functions, then we need only consider evaluators which do some evaluation.

For a flat domain we have that ξ_1 is the only sensible evaluator which does any evaluation. This evaluator divides up the domain as shown in Table 3.1.1.

Table 3.1.1
Division of Domain by ξ_1

Evaluator	*Elements Preserved*	*Elements Not Preserved*
ξ_1	$D^{sl}_A - \{\perp_{D^{sl}_A}\}$	$\perp_{D^{sl}_A}$

The evaluator initiates a non-terminating computation only for the bottom element, and so we need to be able to distinguish in our abstract domain the difference between the bottom element and all other elements. Thus we need a two point domain for our abstract interpretation.

For the type *Alist*, there are three sensible evaluators which do some evaluation. These divide up the standard interpretation as shown in Table 3.1.2.

Table 3.1.2
Division of Domain by ξ_1, ξ_2 and ξ_3

Evaluator	*Elements Preserved*	*Elements Not Preserved*
ξ_1	$D^{st}_{Alist} - \{\perp_{D^{st}_{Alist}}\}$	$\perp_{D^{st}_{Alist}}$
ξ_2	$D^{st}_A{}^*.nil$	$D^{st}_A{}^*.\perp_{D^{st}_{Alist}} \cup D^{st\,\omega}_A$
ξ_3	$(D^{st}_A - \{\perp_{D^{st}_A}\})^*.nil$	$D^{st}_A{}^*.\perp_{D^{st}_{Alist}} \cup D^{st\,\omega}_A \cup D^{st}_A{}^*.\perp_{D^{st}_A}.D^{st}_A{}^*.nil$

We thus need to be able to distinguish between the sets $\{\perp_{D^{st}_{Alist}}\}$, $D^{st}_A{}^*.\perp_{D^{st}_{Alist}} \cup D^{st\,\omega}_A$, $D^{st}_A{}^*.\perp_{D^{st}_{Alist}} \cup D^{st\,\omega}_A \cup D^{st}_A{}^*.\perp_{D^{st}_A}.D^{st}_A{}^*.nil$ and D^{st}_{Alist}. We saw in the previous section that we could do this with the four point domain and defining the abstraction map abs_{Alist} appropriately.

We have seen that for functions there is only one sensible evaluator which does any evaluation, namely ξ_1. Thus as was the case for flat domains, when we are asking how much evaluation can be done of an expression representing a function which is an argument to another function, we need only to be able to test for undefinedness using the bottom element of the abstract interpretation of the appropriate function space. All the extra information in the abstract domains for function spaces is to allow us to find out the definedness of various functions.

3.1.3. Definition of the Abstract Domains and Abstraction Functions for Base Types.

In the previous two sections we have attacked the problem of finding abstract domains and abstraction functions for the base types from two natural intuitions and found that they pointed to the same abstract domains and abstraction maps for the various interpretations of base types. We thus formally make these definitions in this section.

If A is a base type whose standard interpretation is a flat domain, then we define D^{ab}_A:

Definition 3.1.3.1:

$D_A^{ab} = 2$, where $2 = \{0,1\}$ with 0 and 1 distinct elements and $0 \leqslant 1$.

□

We define the abstraction map :

Definition 3.1.3.2:

$$abs_A : D_A^{st} \rightarrow D_A^{ab}$$

$$abs_A(a) = \begin{cases} 0 \text{ if } a = \perp_{D_A^{st}} \\ 1 \text{ otherwise} \end{cases}$$

□

For the type *Alist*, we define the abstract domain, where we lift the restriction that D_A^{st} has to be a flat domain, but we will still use the two point domain for the abstract interpretation of elements of the list :

Definition 3.1.3.3:

$D_{Alist}^{ab} = \{\perp_L, I, F, \top_L\}$, where $\perp_L$, I, F and $\top_L$ are all distinct and are ordered by $\perp_L \leqslant I \leqslant F \leqslant \top_L$.

□

We define the abstraction map :

Definition 3.1.3.4:

$$abs_{Alist} : D_{Alist}^{st} \rightarrow D_{Alist}^{ab}$$

$$abs_{Alist}(L) = \begin{cases} \perp_L \text{ if } L = \perp_{D_{Alist}^{st}} \\ I \text{ if } L \in D_A^{st} . D_A^{st *} . \perp_{D_{Alist}^{st}} \cup D_A^{st\,\omega} \\ F \text{ if } L \in D_A^{st *} . \perp_{D_A^{st}} . D_A^{st *} . nil \\ \top_L \text{ if } L \in (D_A^{st} - \{\perp_{D_A^{st}}\})^* . nil \end{cases}$$

□

Note that our abstract domains for the base types are finite, complete lattices and that the abstraction maps for the base types are strict and continuous as required by the theory of Chapter 2.

3.1.4. Some Useful Facts.

The following facts and lemma are useful in the ensuing development.

Fact 3.1.4.1:

For all types σ, abs_σ is continuous (Lemma 2.2.8 (*i*)).

□

Fact 3.1.4.2:

For all types σ, abs_σ is strict (Lemma 2.2.8 (*iii*)).

□

Fact 3.1.4.3:

For all types σ, abs_σ is bottom-reflexive (Lemma 2.4.4).

□

Lemma 3.1.4.4:

For all types σ, abs_σ is onto.

Proof :

From Lemma 2.4.6 we have to provide a continuous function $abs_A^{-1} : D_A^{ab} \to D_A^{st}$ which is a right inverse of abs_A for each base type A, and then the result follows. For $D_A^{ab} = 2$ we can make the following definition :

$$abs_A^{-1}(0) = \perp_{D_A^{st}}$$

$$abs_A^{-1}(1) = a \in D_A^{st}.\ a \neq \perp_{D_A^{st}}$$

For D_{Alist}^{ab}, a suitable $abs_{Alist}^{-1} : D_{Alist}^{ab} \to D_{Alist}^{st}$ is :

$$abs_{Alist}^{-1}(\perp_L) = \perp_{D_{Alist}^{st}}$$

$$abs_{Alist}^{-1}(I) = \perp_{D_A^{st}} . \perp_{D_{Alist}^{st}}$$

$$abs_{Alist}^{-1}(F) = \perp_{D_A^{st}} . nil$$

$$abs_{Alist}^{-1}(\top_L) = a.nil \quad a \in D_A^{st}.\ a \neq \perp_{D_A^{st}}$$

Clearly abs_A^{-1} and abs_{Alist}^{-1} are continuous and have the required inverse property.

□

Fact 3.1.4.5:

For all σ, $Conc_\sigma(\perp_{D_\sigma^{ab}}) = \{\perp_{D_\sigma^{st}}\}$ (Lemma 2.4.7).

□

3.2. Abstract Interpretation of Constants

To be able to prove the main results of this Chapter, we need to have abstract interpretations of constants which satisfy

$$E^{ab}[[c_\sigma]]\rho^{ab} \geqslant abs_\sigma(E^{st}[[c_\sigma]]\rho^{st}).$$

for all constants c_σ. In this section we derive abstract interpretations of the constants in our language as abstractions of their standard interpretation (c.f. [Burn, Hankin and Abramsky 1985a, Abramsky 1985b], [Nielson 1986b]). This means that the above condition will be automatically satisfied.

As the derivations are rather tedious, we will give the abstract interpretations of the constants below. However, both because we feel that the proofs provide valuable examples of the use of this method to determine optimal abstract interpretations of constants, and because we must assure ourselves of their correctness, we have included them in the following subsections. The proof of the abstract interpretation labelled n below is given in section 3.2.n.

To give the abstract interpretation of the conditional, it is useful to define the following function :

Definition 3.2.1:

$$\overline{if_\sigma} : 2 \to D_\sigma^{ab} \to D_\sigma^{ab}$$

$$\overline{if_\sigma}(0,s) = \perp_{D_\sigma^{ab}}$$

$$\overline{if_\sigma}(1,s) = s$$

□

It is clear that $\overline{if_\sigma}$ is continuous. A reduction rule for this function is given in Lemma 3.2.2.2.

We also note that Lemma 3.2.1.2 is more general than just proving the correctness of the abstract interpretation of strict, binary arithmetic and logical operators.

Definition 3.2.2:

The abstract interpretations of the constants in our language are defined below. Throughout the rest of the thesis we will denote the abstract interpretation of a function (including a functional constant) by writing its name with a bar over it.

(1) $E^{ab}\,[[f]]\,\rho^{ab} = \lambda x\,^{2}_{1}.\lambda x\,^{2}_{2}.x_1$ **and** x_2.

if f is a strict, binary arithmetic or logical operator.

(2) $E^{ab}\,[[\mathrm{if}_{bool\to\sigma\to\sigma\to\sigma}]]\,\rho^{ab} = \lambda x\,^{2}.\lambda y\,^{D^{ab}_{\sigma}}.\lambda z\,^{D^{ab}_{\sigma}}.\overline{\mathrm{if}_{\sigma}}(x.\bigsqcup\{y.z\})$

where $\bigsqcup$ for function spaces is calculated pointwise. i.e. $(\bigsqcup\{f.g\})\,x = \bigsqcup\{f(x).g(x)\}$.

(3) $(E^{ab}\,[[hd]]\,\rho^{ab})\,\bar{L} = \overline{hd}\,\bar{L} = \begin{cases} 0 \text{ if } \bar{L} = \bot_L \\ 1 \text{ otherwise} \end{cases}$

(4) $(E^{ab}\,[[tl]]\,\rho^{ab})\,\bar{L} = \overline{tl}\,\bar{L} = \begin{cases} \bot_L & \text{if } \bar{L} = \bot_L \\ I & \text{if } \bar{L} = I \\ \top_L & \text{otherwise} \end{cases}$

(5) $(E^{ab}\,[[cons]]\,\rho^{ab})\,\bar{a}\,\bar{L} = \overline{cons}\,\bar{a}\,\bar{L} = \begin{cases} I \text{ if } \bar{L} = \bot_L \text{ or } \bar{L} = I \\ F \text{ if } \bar{L} = F \text{ or } (\bar{a} = 0 \text{ and } \bar{L} = \top_L) \\ \top_L \text{ if } \bar{a} = 1 \text{ and } \bar{L} = \top_L \end{cases}$

(6) $(E^{ab}\,[[case]]\,\rho^{ab})\,\bar{s}\,\bar{f}\,\bar{L} = \overline{case}\,\bar{s}\,\bar{f}\,\bar{L} = \begin{cases} \bot_{D^{ab}_{\sigma}} & \text{if } \bar{L} = \bot_L \\ \bar{f}\,1\,I & \text{if } \bar{L} = I \\ (\bar{f}\,1\,F)\bigsqcup(\bar{f}\,0\,\top_L) & \text{if } \bar{L} = F \\ \bar{s}\bigsqcup(\bar{f}\,1\,\top_L) & \text{if } \bar{L} = \top_L \end{cases}$

□

There are some simplification rules for expressions involving applications of the abstract interpretations of *hd*, *tl* and *cons*. To give them, we will first state the standard definitions of two terms.

Definition 3.2.3:

A function $f \in D^I_{\sigma_1} \to \cdots \to D^I_{\sigma_{n+1}}$, where each $D^I_{\sigma_i}$ is a complete lattice, is *multiplicative* in its ith argument if

$$f\, s_1 \cdots s_{i-1}\, (s_i \sqcap s'_i)\, s_{i+1} \cdots s_n = (f\, s_1 \cdots s_i \cdots s_n) \sqcap (f\, s_1 \cdots s'_i \cdots s_n)$$

□

Definition 3.2.4:

A function $f \in D^I_{\sigma_1} \to \cdots \to D^I_{\sigma_{n+1}}$, where each $D^I_{\sigma_i}$ is a complete lattice, is *additive* in its ith argument if

$$f\, s_1 \cdots s_{i-1}\, (s_i \sqcup s'_i)\, s_{i+1} \cdots s_n = (f\, s_1 \cdots s_i \cdots s_n) \sqcup (f\, s_1 \cdots s'_i \cdots s_n)$$

□

Proposition 3.2.5:

Both $\overline{hd}$ and $\overline{tl}$ are multiplicative and additive.

Proof :

Our lattice D^{ab}_{Alist} is a chain, and so the greatest lower bound or least upper bound of any pair is one of the pair, and hence the result follows from monotonicity of $\overline{hd}$ and $\overline{tl}$.

□

We have the following proposition for simplification of expressions involving $\overline{cons}$.

Proposition 3.2.6:

$\overline{cons}$ is multiplicative in its argument considered as a pair (and hence in each argument separately) and is additive in each argument separately, but not in its arguments considered as a pair.

Proof :

The proposition follows by tedious calculation, except for the last part which we show by exhibiting a counter-example.

$$\overline{cons}(1.\bot_L) \sqcup \overline{cons}(0.\top_L) = I \sqcup F$$

$$= F$$

whereas

$$\overline{cons}(<1.\bot_L> \sqcup <0.\top_L>) = \overline{cons}(1.\top_L)$$

$$= \top_L$$

and the two are clearly unequal.

□

Those willing to take the abstract interpretations of the constants on trust may skip sections 3.2.1 to 3.2.6.

While reading the proofs of the correctness of the abstract interpretations of functions, recall that abstraction is modelling the idea that a function "is at most as defined as" something, and so we must pick the maximum value that is possible. This is why we have the least upper bound in the definition of the abstraction maps. So in proofs of the correctness of abstract interpretations, we will often just try and find out what the most defined case is. For example, in Lemma 3.2.4.2 about the abstract interpretation of *tl*, we find that if *tl* is given a finite list with bottom elements in it, it may return either a finite list with bottom elements in it or a finite list with no bottom elements in it (if the only bottom element was the first element of the list). Since it is possible for *tl* to return a finite list with no bottom elements in it, then the abstract interpretation must acknowledge this fact, and will thus choose $\top_L$ as the abstract interpretation of the result in this case. Similar considerations apply to all of the abstract interpretations of the other constants.

3.2.1. Abstract Interpretation of Strict Functions.

Let A be a base type with abstract interpretation the two point domain. If we denote A by A^1 and $A \rightarrow A^n$ by A^{n+1}, then if $f : A^{n+1}$ is such that for all i

$$(E^{st}\ [[f]]\ \rho^{st})\ a_1\ \cdots\ a_{i-1}\ \bot_{D_A^{st}}\ a_{i+1}\ \cdots\ a_n\ =\ \bot_{D_A^{st}}$$

i.e. f is strict in each of its parameters, and

$$E^{st}\ [[f\ a_1\ \cdots\ a_n]]\ \rho^{st}\ =\ \bot_{D_A^{st}} \Rightarrow E^{st}\ [[a_i]]\ \rho^{st}\ =\ \bot_{D_A^{st}}$$

for some i, i.e. a generalisation of $\bot$-reflexivity, then define the abstract interpretation of f by :

Definition 3.2.1.1:

$$E^{ab}\ [[f]]\ \rho^{ab}\ =\ \lambda x_1^2 \cdots \lambda x_n^2 . x_1\ \textbf{and}\ \cdots\ \textbf{and}\ x_n .$$

□

Thus functions like '+' and '×' have abstract interpretation $\lambda x^2.\lambda y^2.x$ and y.

Lemma 3.2.1.2:

Given such an f,

$$E^{ab}\,[[f]]\,\rho^{ab} = abs_{A^{n+1}}(E^{st}\,[[f]]\,\rho^{st})$$

Proof :

We prove this by induction with the base case being $n = 2$. In this proof we will denote the standard interpretation of f by f^{st}.

$$\begin{aligned}
(abs_{A\to A}(f^{st}))\,0 &= \bigsqcup\{abs_A(f^{st}\,x) \mid x \in Conc_A(0)\}^\circ && \text{Proposition 2.3.3}\\
&= \bigsqcup\{abs_A(f^{st}\perp_{D_A^{st}})\}^\circ && \text{Fact 3.1.4.5}\\
&= \bigsqcup\{0\}^\circ && \text{since } f^{st} \text{ and } abs_A \text{ are strict}\\
&= 0 && \text{(P10)}
\end{aligned}$$

$$\begin{aligned}
(abs_{A\to A}(f^{st}))\,1 &= \{abs_A(f^{st}\,x) \mid x \in Conc_A(1)\}^\circ && \text{Proposition 2.3.3}\\
&= \bigsqcup\{abs_A(f^{st}\,a) \mid a \in D_A^{st}\}^\circ\\
&= \bigsqcup\{0.1\}^\circ && \text{since } f^{st} \text{ is strict and not } \perp_{D_{A\to A}^{st}}\\
&= 1
\end{aligned}$$

Therefore, by extensionality, $abs_{A\to A}(f^{st}) = \lambda x^2.x$. In the inductive step, we assume the result for all $n \leqslant k$.

$$\begin{aligned}
(abs_{A^{k+1}}(f^{st}))\,0 &= \bigsqcup\{abs_{A^k}(f^{st}\,x) \mid x \in Conc_A(0)\}^\circ && \text{Proposition 2.3.3}\\
&= \bigsqcup\{abs_{A^k}(f^{st}\perp_{D_A^{st}})\}^\circ && \text{Fact 3.1.4.5}\\
&= \bigsqcup\{abs_{A^k}(\lambda x_1^{D_A^{st}}\cdots\lambda x_{k-1}^{D_A^{st}}.\perp_{D_A^{st}})\}^\circ && \text{since } f^{st} \text{ is strict}\\
&= \lambda x_1^2\cdots\lambda x_{k-1}^2.0 && \text{since } abs_{A^k} \text{ is strict}
\end{aligned}$$

$$\begin{aligned}
(abs_{A^{k+1}}(f^{st}))\,1 &= \bigsqcup\{abs_{A^k}(f^{st}\,x) \mid x \in Conc_A(1)\}^\circ && \text{Proposition 2.3.3}\\
&= \bigsqcup\{abs_{A^k}(f^{st}\,a) \mid a \in D_A^{st}\}^\circ
\end{aligned}$$

$= \bigsqcup\{abs_{A^k}(f^{st}\ a) \mid a \neq \bot_{D_A^{st}}, a \in D_A^{st}\}^\circ$ since f^{st} is monotonic

$= \bigsqcup\{\lambda x_1^2 \cdots \lambda x_{k-1}^2 . x_1 \text{ and } \cdots x_{k-1}\}^\circ$

by inductive hypothesis for $(f^{st}\ a)$ satisfies the condition of the Lemma

$= \lambda x_1^2 \cdots \lambda x_{k-1}^2 . x_1 \text{ and } \cdots \text{ and } x_{k-1}$ (P10)

Hence, $abs_{A^{k+1}}(f^{st}) = \lambda x_1^2 \cdots \lambda x_k^2 . x_1 \text{ and } \cdots \text{ and } x_k$.

□

3.2.2. Abstract Interpretation of the Conditional.

For defining the abstract interpretation of the conditional, it is useful to define the following function :

Definition 3.2.2.1:

$\overline{\text{if}_\sigma} : 2 \to D_\sigma^{ab} \to D_\sigma^{ab}$

$\overline{\text{if}_\sigma}(0, s) = \bot_{D_\sigma^{ab}}$

$\overline{\text{if}_\sigma}(1, s) = s$ □

It is clear that $\overline{\text{if}_\sigma}$ is continuous. The rules defining $\overline{\text{if}_\sigma}$ imply the following reduction rule :

Lemma 3.2.2.2:

If $\overline{e_1} \in D_{bool}^{ab}\ (= 2)$, $\overline{e_2} \in D_{\sigma \to \tau}^{ab}$ and $\overline{e_3} \in D_\sigma^{ab}$, then

$(\overline{\text{if}_{\sigma \to \tau}}(\overline{e_1}, \overline{e_2}))\ \overline{e_3} = \overline{\text{if}_\tau}(\overline{e_1}, \overline{e_2}\ \overline{e_3})$

Proof :

We have two cases :

(i) $\overline{e_1} = 0$:

$(\overline{\text{if}_{\sigma \to \tau}}(0, \overline{e_2}))\ \overline{e_3} = (\bot_{D_{\sigma \to \tau}^{ab}})\ \overline{e_3}$

$= \bot_{D_\tau^{ab}}$

$= \overline{\text{if}_\tau}(0, \overline{e_2}\ \overline{e_3})$

(ii) $\overline{e_1} = 1$:

$$(\overline{\mathrm{if}_{\sigma\to\tau}}(1,\overline{e_2}))\,\overline{e_3} = \overline{e_2}\,\overline{e_3}$$

$$= \overline{\mathrm{if}_\tau}(1,\overline{e_2}\,\overline{e_3})$$

□

We can now define the abstract interpretation of the conditional.

Definition 3.2.2.3:

$$E^{ab}[[\mathrm{if}_{bool\to\sigma\to\sigma\to\sigma}]]\,\rho^{ab} = \lambda x^2.\lambda y^{D_\sigma^{ab}}.\lambda z^{D_\sigma^{ab}}.\overline{\mathrm{if}_\sigma}(x,\bigsqcup\{y,z\})$$

where $\bigsqcup$ for function spaces is calculated pointwise. i.e. $(\bigsqcup\{f,g\})\,x = \bigsqcup\{f(x),g(x)\}$.

□

The abstract interpretation of $\mathrm{if}_{bool\to\sigma\to\sigma\to\sigma}$ is clearly continuous. In the case that $\sigma = A$, we have that

$$E^{ab}[[\mathrm{if}_{bool\to A\to A\to A}]]\,\rho^{ab} = \lambda\bar{x}^2.\lambda\bar{y}^2.\lambda\bar{z}^2.\overline{\mathrm{if}_A}(\bar{x},\bigsqcup\{\bar{y},\bar{z}\})$$

$$= \lambda\bar{x}^2.\lambda\bar{y}^2.\lambda\bar{z}^2.\bar{x}\ \textbf{and}\ (\bar{y}\ \textbf{or}\ \bar{z})$$

and so this abstract interpretation of the conditional can be seen as a generalisation of the interpretation given in [Mycroft 1981].

Lemma 3.2.2.4:

$$E^{ab}[[\mathrm{if}_{bool\to\sigma\to\sigma\to\sigma}]]\,\rho^{ab} = abs_{bool\to\sigma\to\sigma\to\sigma}(E^{st}\ [[\mathrm{if}_{bool\to\sigma\to\sigma\to\sigma}]]\,\rho^{st})$$

Proof :

In this proof we will denote the standard semantics of $\mathrm{if}_{bool\to\sigma\to\sigma\to\sigma}$ by if^{st}.

$abs_{bool\to\sigma\to\sigma\to\sigma}(\mathrm{if}^{st})$

$$= \lambda\bar{x}^2.\lambda\bar{y}^{D_\sigma^{ab}}.\lambda\bar{z}^{D_\sigma^{ab}}.\{abs_\sigma(\mathrm{if}^{st}(x,y,z)) \mid abs_A(x)\leqslant\bar{x}, abs_\sigma(y)\leqslant\bar{y}, abs_\sigma(z)\leqslant\bar{z}\}^\circ$$

Proposition 2.3.1

$(abs_{bool\to\sigma\to\sigma\to\sigma}(\mathrm{if}^{st}))\ 0$

$$= \lambda\bar{y}^{D_\sigma^{ab}}.\lambda\bar{z}^{D_\sigma^{ab}}.\bigsqcup\{abs_\sigma(\mathrm{if}^{st}(\bot_{D_{bool}^{st}},y,z)) \mid abs_\sigma(y)\leqslant\bar{y}, abs_\sigma(z)\leqslant\bar{z}\}^\circ$$

since abs_A is $\bot$-reflexive

$= \lambda \bar{y}^{D_\sigma^{ab}} . \lambda \bar{z}^{D_\sigma^{ab}} . \bigsqcup \{abs_\sigma(\bot_{D_\sigma^{st}})\}^\circ$ by the standard semantics of the conditional

$= \lambda \bar{y}^{D_\sigma^{ab}} . \lambda \bar{z}^{D_\sigma^{ab}} . \bigsqcup \{\bot_{D_\sigma^{ab}}\}^\circ$ since abs_σ is strict

$= \lambda \bar{y}^{D_\sigma^{ab}} . \lambda \bar{z}^{D_\sigma^{ab}} . \bot_{D_\sigma^{ab}}$ (P10)

$(abs_{bool \to \sigma \to \sigma \to \sigma}(\text{if}^{st}))\ 1$

$= \lambda \bar{y}^{D_\sigma^{ab}} . \lambda \bar{z}^{D_\sigma^{ab}} . \bigsqcup \{abs_\sigma(\text{if}^{st}(x,y,z)) \mid x \in D_{bool}^{st} . abs_\sigma(y) \leqslant \bar{y} . abs_\sigma(z) \leqslant \bar{z}\}^\circ$

$= \lambda \bar{y}^{D_\sigma^{ab}} . \lambda \bar{z}^{D_\sigma^{ab}} . \bigsqcup \{abs_\sigma(\bot_{D_\sigma^{st}}) . abs_\sigma(y) . abs_\sigma(z) \mid abs_\sigma(y) \leqslant \bar{y} . abs_\sigma(z) \leqslant \bar{z}\}^\circ$

by the standard semantics of the conditional

$= \lambda \bar{y}^{D_\sigma^{ab}} . \lambda \bar{z}^{D_\sigma^{ab}} . \bigsqcup \{\bar{y} . \bar{z}\}^\circ$ (§)

by monotonicity of abs_σ and since abs_σ is onto (Lemma 3.1.4.4)

$= \lambda \bar{y}^{D_\sigma^{ab}} . \lambda \bar{z}^{D_\sigma^{ab}} . \bigsqcup \{\bar{y} . \bar{z}\}$

since $\bigsqcup X^\circ = \bigsqcup X$ for finite, complete lattices [Abramsky 1985b]

and so we have the result by extensionality.

□

Note that we have equality in the step marked by (§), where we would have to replace this by $\leqslant$ if the abstraction map for the type σ was not onto.

3.2.3. Abstract Interpretation of *hd*.

Definition 3.2.3.1:

$$\overline{hd} = E^{ab}\,[[hd]]\,\rho^{ab} : D_{Alist}^{ab} \to 2$$

$$\overline{hd}(\bar{L}) = (E^{ab}\,[[hd]]\,\rho^{ab})\,(\bar{L}) = \begin{cases} 0 \text{ if } \bar{L} = \bot_L \\ 1 \text{ otherwise} \end{cases}$$

□

We note that $\overline{hd}$ is continuous.

Lemma 3.2.3.2:

$$E^{ab}\,[[hd]]\,\rho^{ab} = abs_{Alist\to A}\,(E^{st}\,[[hd]]\,\rho^{st})$$

Proof :

In this proof we will denote the standard semantics of hd by hd^{st}.

$(abs_{Alist\to A}\,(hd^{st}))\;\bar{L} = \bigsqcup\{abs_A\,(hd^{st}(L)) \mid L \in Conc_{Alist}\,(\bar{L})\}^{\circ}$ Proposition 2.3.3

(*i*) If $\bar{L} = \perp_L$, then the only L in $Conc_{Alist}\,(\perp_L)$ is $\perp_{D^{st}_{Alist}}$ (Fact 3.1.4.5) and so we obtain $\perp_{D^{ab}_A}$ for the above as both hd^{st} and abs_A are strict.

(*ii*) If $\bar{L}$ is any other element of D^{ab}_{Alist}, then the concretisation of $\bar{L}$ contains lists, L, with defined heads, and $abs_A\,(hd^{st}(L))$ for these will be 1 and hence the result.

□

If we were to use the two point domain for the abstract interpretation of the type *Alist*, then it can easily be shown that $abs_{Alist\to A}\,(hd^{st}) = \lambda x^2.x$.

3.2.4. Abstract Interpretation of *tl*.

Definition 3.2.4.1:

$$\bar{tl} = E^{ab}\,[[tl]]\,\rho^{ab} : D^{ab}_{Alist} \to D^{ab}_{Alist}$$

$$\bar{tl}(\bar{L}) = (E^{ab}\,[[tl]]\,\rho^{ab})\,(\bar{L}) = \begin{cases} \perp_L & \text{if } \bar{L} = \perp_L \\ I & \text{if } \bar{L} = I \\ \top_L & \text{otherwise} \end{cases}$$

□

We note that $\bar{tl}$ is continuous.

Lemma 3.2.4.2:

$$E^{ab}\,[[tl]]\,\rho^{ab} = abs_{Alist\to Alist}\,(E^{st}\,[[tl]]\,\rho^{st})$$

Proof :

In this proof we will denote the standard semantics of tl by tl^{st}.

$(abs_{Alist \to Alist}(tl^{st}))\ \bar{L} = .\bigsqcup\{abs_{Alist}(tl^{st}(L)) \mid L \in Conc_{Alist}(\bar{L})\}^{\circ}$ Proposition 2.3.3

(*i*) If $\bar{L}$ is $\bot_L$, then as $Conc_{Alist}$ is strict (Fact 3.1.4.5), and tl^{st} and abs_{Alist} are strict, we have the result.

(*ii*) If $\bar{L}$ is I, then L is in the concretisation of $\bar{L}$ if and only if L is the bottom list or partial or infinite. Taking the tail of such a list returns one of the bottom list, a partial list or an infinite list, and the least upper bound of the abstraction of these things is I.

(*iii*) If $\bar{L}$ is F or $\top_L$, then the concretisation of $\bar{L}$ contains, besides other things, all the finite lists. Taking the tail of a list which has only an undefined head returns a list with no bottom elements, and so we can get $\top_L$ for both F (finite lists with bottom elements) and $\top_L$. We obtain the result since we take the least upper bound.

□

If we were to use the two point domain for the abstract interpretation of the type *Alist*, then it can easily be shown that $abs_{Alist \to Alist}(tl^{st}) = \lambda x^2 . x$.

3.2.5. Abstract Interpretation of *cons*.

Definition 3.2.5.1:

$$\overline{cons} = E^{ab}[[cons]]\ \rho^{ab} : 2 \to D^{ab}_{Alist} \to D^{ab}_{Alist}$$

The value of $\overline{cons}(\bar{a}, \bar{L})$ for each $\bar{a} \in D^{ab}_A$ and $\bar{L} \in D^{ab}_{Alist}$ are given in Table 3.2.5.1.

Table 3.2.5.1
Abstract Interpretation of *cons*

$\bar{L}$ \ $\bar{a}$	0	1
$\bot_L$	I	I
I	I	I
F	F	F
$\top_L$	F	$\top_L$

□

We note that $\overline{cons}$ is continuous.

Lemma 3.2.5.2:

$$E^{ab}\,[[cons]]\,\rho^{ab} = abs_{A\to Alist\to Alist}(E^{st}\,[[cons]]\,\rho^{st})$$

Proof :

We denote the standard semantics of *cons* by $cons^{st}$ in the following proof.

$$abs_{A\to Alist\to Alist}(cons^{st})\;\bar{a}\;\bar{L} =$$

$$\bigsqcup\{abs_{Alist}(cons^{st}\;a\;L) \mid a\in Conc_A(\bar{a}).L\in Conc_{Alist}(\bar{L})\}^{\circ} \qquad \text{Proposition 2.3.3}$$

We give two examples of the calculation for two pairs of arguments from the table. The others follow in a similar manner.

$$(i)\;(abs_{A\to Alist\to Alist}(cons^{st}))\;0\;\bot_L = \bigsqcup\{abs_{Alist}(cons^{st}(\bot_{D_A^{st}}.\bot_{D_{Alist}^{st}}))\}^{\circ}$$

Fact 3.1.4.5

$$= \bigsqcup\{I\}^{\circ}$$

$$= I \qquad \text{(P10)}$$

$$(ii)\;(abs_{A\to Alist\to Alist}(cons^{st}))\;1\;F$$

$$= \bigsqcup\{abs_{Alist}(cons^{st}\;a\;L) \mid a\in D_A^{st}.L\in Conc_{Alist}(F)\}^{\circ}$$

The most defined result of the above formula is going to be when we have a total element for the first argument to the $cons^{st}$ and an element from $D_A^{st*}.\bot_{D_A^{st}}.D_A^{st*}.nil$ for the second element, in which case $cons^{st}\;a\;L$ is a finite list with bottom elements and so the formula collapses to F, as in the table.

The other six cases follow in a similar manner.

□

If we wish to use the two point domain for the abstract interpretation of the type *Alist*, then it can easily be shown that $abs_{A\to Alist\to Alist}(cons^{st}) = \lambda x^{2}.\lambda y^{D_{Alist}^{ab}}.1$.

3.2.6. Abstract Interpretation of the Case Statement.

Definition 3.2.6.1:

$$\overline{case} = E^{ab}\,[[case]]\,\rho^{ab} : D_{\sigma}^{ab} \to (2 \to D_{Alist}^{ab} \to D_{\sigma}^{ab}) \to D_{Alist}^{ab} \to D_{\sigma}^{ab}$$

$$\overline{case}\ \bar{s}\ \bar{f}\ \bar{L} = (E^{ab}\,[[case]]\,\rho^{ab})\ \bar{s}\ \bar{f}\ \bar{L} = \begin{cases} \bot_{D_{\sigma}^{ab}} & \text{if } \bar{L} = \bot_L \\ \bar{f}\ 1\ I & \text{if } \bar{L} = I \\ (\bar{f}\ 1\ F) \sqcup (\bar{f}\ 0\ \top_L) & \text{if } \bar{L} = F \\ \bar{s} \sqcup (\bar{f}\ 1\ \top_L) & \text{if } \bar{L} = \top_L \end{cases}$$

□

We note that $\overline{case}$ is continuous.

Lemma 3.2.6.2:

$$E^{ab}\,[[case]]\,\rho^{ab} \geqslant abs_{\sigma \to (A \to Alist \to \sigma) \to Alist \to \sigma}(E^{st}\ [[case]]\,\rho^{st})$$

Proof :

We denote the standard interpretation of the case statement by $case^{st}$ in the following proof.

$$(abs_{\sigma \to (A \to Alist \to \sigma) \to Alist \to \sigma}(case^{st}))\ \bar{s}\ \bar{f}\ \bar{L}$$

$$= \bigsqcup \{abs_{\sigma}(case^{st}\ s\ f\ L) \mid abs_{\sigma}(s) \leqslant \bar{s}, abs_{A \to Alist \to \sigma}(f) \leqslant \bar{f}, abs_{Alist}(L) \leqslant \bar{L}\}^{\circ} \qquad (\S)$$

Proposition 2.3.1

We will give two examples for $\bar{L}$.

(i) If $\bar{L} = \bot_L$, then $L = \bot_{D_{Alist}^{st}}$ in (§) by Fact 3.1.4.5, and $case^{st}\ s\ f \bot_{D_{Alist}^{st}} = \bot_{D_{\sigma}^{st}}$. The result then follows from the strictness of abs_{σ}.

(ii) If $\bar{L} = I$, then we have that $L \in D_A^{st\,*}.\bot_{D_{Alist}^{st}} \cup D_A^{st\,\omega}$ and so $case^{st}\ s\ f\ L = f(hd(L), tl(L))$. We have that $a = hd(L)$ can be any element in D_A^{st}, and that $L' = tl(L)$ is in the same set as L. Hence (§) becomes

$$\bigsqcup \{abs_{\sigma}(f\ a\ L') \mid abs_{A \to Alist \to \sigma}(f) \leqslant \bar{f}, abs_{A}(a) \leqslant 1, abs_{Alist}(L') \leqslant I\}^{\circ}$$

There is no other way to simplify this other than by expanding out $abs_{\sigma}(f\ a\ L')$ and replacing $=$ by $\leqslant$ by the semi-homomorphic property of abstraction (Proposition 2.6.1) to obtain

$$\bigsqcup\{abs_{\sigma}(f\ a\ L') \mid abs_{A\to Alist\to\sigma}(f) \leqslant \bar{f}.abs_A(a) \leqslant 1.abs_{Alist}(L') \leqslant I\}^{\circ}$$

$$\leqslant \bigsqcup\{(abs_{A\to Alist\to\sigma}(f))\ (abs_A(a))\ (abs_{Alist}(L'))$$

$$\mid abs_{A\to Alist\to\sigma}(f) \leqslant \bar{f}.abs_A(a) \leqslant 1.abs_{Alist}(L') \leqslant I\}^{\circ}$$

$$= \bigsqcup\{\bar{f}\ 1\ I\}^{\circ}$$

$$= \bar{f}\ 1\ I \qquad \text{(P10)}$$

The other two cases follow in a similar manner. We note that the inequality is produced in both of the other two cases for the same reason as the above. Finally, $\bar{s}$ only appears in the case that $\bar{L}$ is $\top_L$ as *nil* is a total list and so only appears in the concretisation of $\top_L$.

□

This is the only constant we consider for which we will not be able to obtain equality between the abstract interpretation and the abstraction of the standard interpretation (except for the conditional if the abstraction maps of the base domains are not onto). This is because *case* is not only an higher-order function, but the functional argument is applied to (parts of) one of the other arguments, and so our requirement about safety means we end up with the inequality.

If we were to use the two point domain for the abstract interpretation of *Alist*, then it can easily be shown that

$$(abs_{\sigma\to(A\to Alist\to\sigma)\to Alist\to\sigma}(E^{st}\ [[case]]\ \rho^{st})) \leqslant g$$

where

$$g\ \bar{s}\ \bar{f}\ \bar{L} = \begin{cases} \bot_{D^{st}_{\sigma}} & \text{if } \bar{L} = 0 \\ \bar{s} \bigsqcup (\bar{f}\ 1\ 1) & \text{otherwise} \end{cases}$$

3.3. Some Examples of the Abstract Interpretation of Functions.

Having determined the abstract interpretation of constants, we are now able to give some examples of the abstract interpretation of user-defined functions. How we interpret these abstract interpretations is the topic of the next two sections.

We first give a couple of examples of non-recursive functions.

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

$$h = g\ (\lambda z^{int}.z)\ 5$$

where we have written the '+' in prefix form to make explicit where the function applications are. We note that from Definition 3.2.1.1 that

$$E^{ab}\ [[+]]\ \rho^{ab} = \lambda u^{2}.\lambda v^{2}.u \text{ and } v$$

$$\bar{g} = E^{ab}\ [[g]]\ \rho^{ab}$$

$$= E^{ab}\ [[\lambda f^{int \to int}.\lambda x^{int}.\lambda y^{int}.+\ x\ (f\ y)]]\ \rho^{ab}$$

$$= \lambda \bar{f}^{2 \to 2}.E^{ab}\ [[\lambda x^{int}.\lambda y^{int}.+\ x\ (f\ y)]]\ \rho^{ab}[\bar{f}/f]$$

$$= \lambda \bar{f}^{2 \to 2}.\lambda \bar{x}^{2}.E^{ab}\ [[\lambda y^{int}.+\ x\ (f\ y)]]\ \rho^{ab}[\bar{f}/f, \bar{x}/x]$$

$$= \lambda \bar{f}^{2 \to 2}.\lambda \bar{x}^{2}.\lambda \bar{y}^{2}.E^{ab}\ [[+\ x\ (f\ y)]]\ \rho^{ab}[\bar{f}/f, \bar{x}/x, \bar{y}/y]$$

$$= \lambda \bar{f}^{2 \to 2}.\lambda \bar{x}^{2}.\lambda \bar{y}^{2}.(E^{ab}\ [[+\ x]]\ \rho^{ab'})(E^{ab}\ [[f\ y]]\ \rho^{ab'})$$

where $\rho^{ab'} = \rho^{ab}[\bar{f}/f, \bar{x}/x, \bar{y}/y]$

$$= \lambda \bar{f}^{2 \to 2}.\lambda \bar{x}^{2}.\lambda \bar{y}^{2}.((E^{ab}\ [[+]]\ \rho^{ab'})(E^{ab}\ [[x]]\ \rho^{ab'}))((E^{ab}\ [[f]]\ \rho^{ab'})(E^{ab}\ [[y]]\ \rho^{ab'}))$$

$$= \lambda \bar{f}^{2 \to 2}.\lambda \bar{x}^{2}.\lambda \bar{y}^{2}.((\lambda u^{2}.\lambda v^{2}.u \text{ and } v)\ (\rho^{ab'}(x)))\ ((\rho^{ab'}(f))\ (\rho^{ab'}(y)))$$

$$= \lambda \bar{f}^{2 \to 2}.\lambda \bar{x}^{2}.\lambda \bar{y}^{2}.((\lambda u^{2}.\lambda v^{2}.u \text{ and } v)\ \bar{x})\ (\bar{f}\ \bar{y})$$

$$= \lambda \bar{f}^{2 \to 2}.\lambda \bar{x}^{2}.\lambda \bar{y}^{2}.\bar{x} \text{ and } (\bar{f}\ \bar{y})$$

Similarly,

$$E^{ab}\ [[h]]\ \rho^{ab} = ((E^{ab}\ [[g]]\ \rho^{ab})\ (E^{ab}\ [[\lambda z^{int}.z]]\rho^{ab}))\ (E^{ab}\ [[5]]\ \rho^{ab})$$

$$= ((\lambda \bar{f}^{2 \to 2}.\lambda \bar{x}^{2}.\lambda \bar{y}^{2}.\bar{x} \text{ and } (\bar{f}\ \bar{y}))\ (\lambda \bar{z}^{2}.\bar{z}))\ 1$$

$$= \lambda \bar{y}^{2}.\bar{y}$$

We give an example of a recursive function.

$$fac = fix(\lambda f^{int \to int}.\lambda n^{int}.\text{if}(=\ n\ 0, 1, \times\ n\ (f\ (-\ n\ 1))))$$

$$\overline{fac} = E^{ab}\ [[fac]]\ \rho^{ab}$$

$$= E^{ab}\ [[fix(\lambda f^{int \to int}.\lambda n^{int}.\text{if}(=\ n\ 0, 1, \times\ n\ (f\ (-\ n\ 1))))]]\ \rho^{ab}$$

$$= fix(E^{ab}\,[[\lambda f^{int\to int}.\lambda n^{int}.\mathrm{if}(=\ n\ 0.1.\times\ n\ (f\ (-\ n\ 1)))]]\ \rho^{ab})$$

The abstract interpretation of $\lambda f^{int\to int}.\lambda n^{int}.\mathrm{if}(=\ n\ 0.1.\times\ n\ (f\ (-\ n\ 1)))$ follows as in the previous example, and we end up with

$$E^{ab}\,[[fac]]\ \rho^{ab} = fix(\lambda \bar{f}^{2\to 2}.\lambda \bar{n}^{2}.\bar{n}\ \mathbf{and}\ (1\ \mathbf{or}\ \bar{f}(\bar{n})))$$

To finish off the abstract interpretation we must now calculate the least fixed point. The bottom of the domain $2\to 2$ is $\lambda x^{2}.0$, and so we have the following sequence, $\{\overline{fac_n}\}$, of approximations to $\overline{fac}$:

$$\overline{fac}_0 = \lambda x^{2}.0$$

$$\overline{fac}_1 = \lambda \bar{n}^{2}.\bar{n}\ \mathbf{and}\ (1\ \mathbf{or}\ (\lambda x^{2}.0)\ \bar{n})$$

$$= \lambda \bar{n}^{2}.\bar{n}\ \mathbf{and}\ (1\ \mathbf{or}\ 0)$$

$$= \lambda \bar{n}^{2}.\bar{n}$$

$$\overline{fac}_2 = \lambda \bar{n}^{2}.\bar{n}$$

and thus

$$E^{ab}\,[[fac]]\ \rho^{ab} = \lambda \bar{n}^{2}.\bar{n}.$$

Although we could have simplified the above expression straight away to $\lambda \bar{n}^{2}.\bar{n}$, noting that $1\ \mathbf{or}\ e = 1$ and $\bar{n}\ \mathbf{and}\ 1 = \bar{n}$, we chose not to because to have a fixed pointing algorithm based on equivalence of expressions means we have to deal with the problem of finding canonical forms, and we are after all interested in functional equality.

It is useful to gather together the abstract interpretations of some functions. Besides the constant functions and functions we have already discussed, we present the abstract interpretation of the following functions, where we allow ourselves the luxury of using the name *append* in the definition of *reverse*.

$$sumlist = fix(\lambda f^{Alist\to int}.\lambda L_1^{Alist}.case(0,\lambda x^{A}.\lambda L_2^{Alist}.x + f(L_2).L_1))$$

$$length = fix(\lambda f^{Alist\to int}.\lambda L_1^{Alist}.case(0,\lambda x^{A}.\lambda L_2^{Alist}.1 + f(L_2).L_1))$$

$$append = fix(\lambda f^{Alist\to Alist\to Alist}.\lambda L_1^{Alist}.L_2^{Alist}.case(L_2,\lambda x^{A}.\lambda L_3^{Alist}.cons(x,f(L_3,L_2)).L_1))$$

$$reverse = fix(\lambda f^{Alist\to Alist}.\lambda L_1^{Alist}.case(nil,\lambda x^{A}.\lambda L_2^{Alist}.append(f\ L_2,cons(x,nil)).L_1))$$

$$map = fix(\lambda f^{(A\to B)\to Alist\to Blist}.\lambda g^{A\to B}.\lambda L_1^{Alist}.case(nil,\lambda x^{A}.\lambda L_2^{Alist}.cons(g\ x,f\ g\ L_2).L_1))$$

(Notice how we have restricted the type of the first parameter to *map*, the function g, to being $A \rightarrow B$ and not the more general $\sigma \rightarrow \tau$ because we are working with a monotyped framework, and we have chosen the simplest type for this example.)

Written in a more familiar notation [Turner 1985], these definitions are :

sumlist [] = 0
sumlist x:xs = x + *sumlist* xs

length [] = 0
length x:xs = 1 + *length* xs

append [] L = L
append x:xs L = x:*append* xs L

reverse [] = []
reverse x:xs = *append* (*reverse* xs) x:*nil*

map f [] = []
map f x:xs = (f x):*map* f xs

The abstract interpretations of these functions are given in the following five tables. In each case, the abstract interpretation of a function is denoted by putting a bar over the name of the function. For example, $\overline{sumlist}$ is the abstract interpretation of *sumlist*.

Table 3.3.1 gives $\bar{f}\,\overline{x_1}\,\overline{x_2}$ where f is any binary function satisfying the conditions of Definition 3.2.1.1, for example, +, × and **or**.

Table 3.3.1
Abstract Interpretation of Binary Strict Functions

$\overline{x_2}$ \ $\overline{x_1}$	0	1
0	0	0
1	0	1

Table 3.3.2 gives the abstract interpretation of functions of type $Alist \rightarrow A$ where the abstract interpretation of the type A is 2. Abstract interpretations of functions of type $Alist \rightarrow Alist$ are given in Table 3.3.3. Table 3.3.4 gives the values of $\overline{append}\,\overline{L_1}\,\overline{L_2}$. The values of $\overline{map}\,\bar{f}\,\bar{L}$ are given in Table 3.3.5.

Table 3.3.2.
Abstract Interpretation of *sumlist*, *length*, and *hd*

$\bar{L}$	$\overline{sumlist}(\bar{L})$	$\overline{length}(\bar{L})$	$\overline{hd}(\bar{L})$
$\bot_L$	0	0	0
I	0	0	1
F	0	1	1
$\top_L$	1	1	1

Table 3.3.3.
Abstract Interpretation of *reverse* and *tl*

$\bar{L}$	$\overline{reverse}(\bar{L})$	$\overline{tl}(\bar{L})$
$\bot_L$	$\bot_L$	$\bot_L$
I	$\bot_L$	I
F	F	$\top_L$
$\top_L$	$\top_L$	$\top_L$

Table 3.3.4.
Abstract Interpretation of *append*

$\bar{L}_2 \backslash \bar{L}_1$	$\bot_L$	I	F	$\top_L$
$\bot_L$	$\bot_L$	I	I	I
I	$\bot_L$	I	I	I
F	$\bot_L$	I	F	F
$\top_L$	$\bot_L$	I	F	$\top_L$

3.4. Correctness of the Definedness Interpretation and Context-free and Context-sensitive Issues.

We are now in the position where we are able to prove the correctness of our definedness interpretation and provide theorems for determining context-free and context-sensitive definedness information. Some examples will show why we need both types of information.

Table 3.3.5.
Abstract Interpretation of *map*

$\bar{L} \backslash \bar{f}$	$\lambda\bar{x}^2.0$	$\lambda\bar{x}^2.x$	$\lambda\bar{x}^2.1$
$\perp_L$	$\perp_L$	$\perp_L$	$\perp_L$
I	I	I	I
F	F	F	$\top_L$
$\top_L$	$\top_L$	$\top_L$	$\top_L$

The context-free and context-sensitive theorems are applied in the next section to prove two theorems which allow us to safely change the evaluation strategy by doing some evaluation of some of the arguments to a function.

Theorem 3.4.1: (Correctness Theorem for the Definedness Interpretation)

Suppose $f : \sigma \rightarrow \tau$ and $\rho^{ab} \geqslant abs \circ \rho^{st}$, and $(E^{ab}\,[[f]]\,\rho^{ab})\,\bar{s} = \bar{t}$, then for all $s \in Conc_\sigma(\bar{s})$, $(E^{st}\,[[f]]\,\rho^{st})\,s \in Conc_\tau(\bar{t})$.

Proof :

We have provided abstract interpretations and abstraction maps for the base types as required for the theory of Chapter 2. Furthermore, we have given for each constant c_σ an abstract interpretation which satisfies :

$$E^{ab}\,[[c_\sigma]]\,\rho^{ab} \geqslant abs_\sigma(E^{st}\,[[c_\sigma]]\,\rho^{st})$$

as required by Theorem 2.7.1. Hence this theorem follows as does Theorem 2.8.2.

□

Before we give the context-free and context-sensitive theorems, we will show how in this interpretation each type of information gives something which the other lacks. For first-order functions over flat domains, the context-free definedness information is sufficient to ensure maximum parallel evaluation. Considering a function like

$$f = \lambda x^{int}.\lambda^{int}.\mathit{if}\ x = 0\ \mathit{then}\ y\ \mathit{else}\ f(x-1,y)$$

we see that f will be strict in both of its parameters in all contexts.

However, if we have more complex domains for the abstract interpretation of the base types than the two point domain, or if we have higher-order functions, then this is no longer the case. For example, if we have a higher-order function, say

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

we see that the strictness of g in y depends on the strictness of the parameter f. Thus in the context of an application of g to a strict function, say

$$g\ (\lambda z^{int}.z)\ e$$

the application $g\ (\lambda z^{int}.z)$ is strict in its parameter, while in an application of g to a non-strict function, say

$$g\ (\lambda z^{int}.5)\ e$$

we can see that the application $g\ (\lambda z^{int}.5)\ e$ does not need the value of e.

Thus we see that the maximum potential for parallel evaluation is only captured if we take into account the contextual information in applications.

Seeing as such *context-sensitive* information is always stronger than context-free information, the question arises as to whether we can dispense with context-free information. Unfortunately the answer is no, and the reason is that the evaluation of the program can dynamically create application nodes which do not appear in the original program. This is a simple consequence of the fact that functions are curried, and so a partially applied function can be passed as an argument to a function and later applied to more arguments. An example of this occurs with the higher-order conditional

$$(\textit{if condition then } f_1 \textit{ else } f_2)\ e$$

where f_1 is a strict function and f_2 is a non-strict function. In this case we will not be able to label the apply node with e as an argument because it is not known until run-time whether the *condition* will be true or not. Thus we would like functions to carry around information regarding how much evaluation can be done on arguments so that it may be possible to initiate the evaluation of the argument expression in dynamically created applications. Clearly this information must be true in any context, and so we must use the *context-free* information.

Having motivated the need for both types of information, we give two theorems which can be applied to determine firstly context-free information and secondly context-sensitive information. These are just the context-free and context-sensitive information theorems (Theorems 2.9.1 and 2.9.2). They follow directly from the correctness of this abstract interpretation (Theorem 3.4.1), just as Theorems 2.9.1 and 2.9.2 follow directly from the correctness of the framework for abstract interpretation (Theorem 2.8.2). Thus they will be stated without proof.

Theorem 3.4.2: (Context-Free Definedness Theorem)

If $f : \sigma_1 \rightarrow \cdots \sigma_n \rightarrow \tau$ and

$$(E^{ab}\,[[f]]\,\rho^{ab})\ \top_{D^{ab}_{\sigma_1}} \cdots \top_{D^{ab}_{\sigma_{i-1}}}\ \overline{s_i}\ \top_{D^{ab}_{\sigma_{i+1}}} \cdots \top_{D^{ab}_{\sigma_n}} = \overline{t}$$

then for all $e_j : \sigma_j$, $j \neq i$, for all $s_i \in Conc_{\sigma_i}(\overline{s_i})$, we have

$$E^{st}\,[[f]]\,\rho^{st}\,E^{st}\,[[e_1]]\,\rho^{st} \cdots E^{st}\,[[e_{i-1}]]\,\rho^{st}\,s_i\,E^{st}\,[[e_{i+1}]]\,\rho^{st} \cdots E^{st}\,[[e_n]]\,\rho^{st} \in Conc_\tau(\overline{t})$$

□

We note that the concept of the information being true in any context is captured by putting $\top_{D^{ab}_{\sigma_j}}$ for the jth argument, $j \neq i$ where we are testing the ith argument. This is because all the elements of the standard domain abstract to something which is less than or equal to the top of the abstract domain.

Theorem 3.4.3: (Context-Sensitive Definedness Theorem)

Given $f : \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \tau$ and an application $f\,e_1 \cdots e_n : \tau$, if

$$E^{ab}\,[[f]]\,\rho^{ab}\,E^{ab}\,[[e_1]]\,\rho^{ab} \cdots E^{ab}\,[[e_{i-1}]]\,\rho^{ab}\,\overline{s_i}\,E^{ab}\,[[e_{i+1}]]\,\rho^{ab} \cdots E^{ab}\,[[e_n]]\,\rho^{ab} = \overline{t}$$

then for all $s_i \in Conc_{\sigma_i}(\overline{s_i})$

$$E^{st}\,[[f]]\,\rho^{st}\,E^{st}\,[[e_1]]\,\rho^{st} \cdots E^{st}\,[[e_{i-1}]]\,\rho^{st}\,s_i\,E^{st}\,[[e_{i+1}]]\,\rho^{st} \cdots E^{st}\,[[e_n]]\,\rho^{st} \in Conc_\tau(\overline{t})$$

□

3.5. Using the Definedness Information to Safely Change the Evaluation Strategy.

The context-free and context-sensitive definedness information can be used to change the evaluation strategy for a functional program. Our intuition is that if a function application is undefined for a certain definedness level of one of its arguments, no matter how defined the other arguments are, then the function must have to evaluate that argument a certain amount at some time in the evaluation of that function application. We capture this formally in the following two theorems, which allow us to change the evaluation strategy to do some evaluation of the arguments to a function in parallel with the function application (or before it on a sequential machine) when it is safe to do so.

Theorem 3.5.1: (Safe Context-Free Change of Evaluation Strategy)

Suppose that $f : \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \tau$, that it is safe to do some evaluation of an application of f, and that

$$(E^{ab}\,[[f]]\,\rho^{ab})\ \top_{D^{ab}_{\sigma_1}}\ \cdots\ \top_{D^{ab}_{\sigma_{i-1}}}\ \overline{s_i}\ \top_{D^{ab}_{\sigma_{i+1}}}\ \cdots\ \top_{D^{ab}_{\sigma_n}} = \bot_{D^{ab}_{\tau}}.$$

Then in any application of f, it is safe to use an evaluation strategy which evaluates the ith argument of f so as to preserve all of the values in $D^{st}_{\sigma_i} - Conc_{\sigma_i}(\overline{s_i})$.

Proof :

Given any function application $f\,e_1\ \cdots\ e_n$ there are two cases :

(*i*) Suppose $E^{st}\,[[e_i]]\,\rho^{st} \in Conc_{\sigma_i}(\overline{s_i})$. Then the evaluation of e_i may initiate a non-terminating computation because we have only guaranteed to preserve elements in the complement of this set. However, in this case, Theorem 3.4.2 ensures us that

$$E^{st}\,[[f\,e_1\ \cdots\ e_n]]\,\rho^{st} \in Conc_{\tau}(\bot_{D^{ab}_{\tau}}).$$

Fact 3.4.5 says that in this interpretation, $Conc_{\tau}(\bot_{D^{ab}_{\tau}})$ is just $\{\bot_{D^{st}_{\tau}}\}$ and so the above collapses to

$$E^{st}\,[[f\,e_1\ \cdots\ e_n]]\,\rho^{st} = \bot_{D^{st}_{\tau}}.$$

The evaluation of the application $f\,e_1\ \cdots\ e_n$ will not terminate (as no amount of evaluation will preserve $\bot_{D^{st}_{\tau}}$, except doing no evaluation!), but it was safe to evaluate the function application, and so this means that the semantics of the original expression was bottom (by the fact that it was safe to evaluate the function application), and so it was safe to initiate a non-terminating computation in the evaluation of e_i.

(*ii*) If $E^{st}\,[[e_i]]\,\rho^{st} \notin Conc_{\sigma_i}(\overline{s_i})$ then, since we have chosen an evaluator which preserves these elements, we have that no non-terminating computation will be initiated in evaluating the ith argument, and so doing this is safe.

□

If there is no such $\overline{s_i}$, then we can do no evaluation of the ith argument, that is, the only safe evaluator is ξ_0.

Theorem 3.5.2:(Safe Context-Sensitive Change of Evaluation Theorem)

Suppose that $f : \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \tau$, that it is safe to do some evaluation of the function application $f\, e_1 \cdots e_n$ and that

$$E^{ab}[[f]]\,\rho^{ab}\,E^{ab}[[e_1]]\,\rho^{ab} \cdots E^{ab}[[e_{i-1}]]\,\rho^{ab}\,\overline{s_i}\,E^{ab}[[e_{i+1}]]\,\rho^{ab} \cdots E^{ab}[[e_n]]\,\rho^{ab} = \perp_{D_\tau^{st}}$$

Then it is safe to use an evaluation strategy which evaluates e_i so as to preserve all values in $D_{\sigma_i}^{st} - Conc_{\sigma_i}(\overline{s_i})$.

Proof :

The proof of this theorem follows exactly as the proof of Theorem 3.5.1 except that we appeal to Theorem 3.4.3 rather than Theorem 3.4.2.

□

If there is no such $\overline{s_i}$, then it is not safe to do any evaluation of the ith argument, that is, the only safe evaluator is ξ_0.

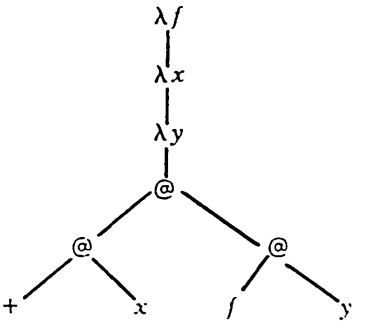
The previous two theorems allowed us to choose an evaluation strategy which evaluated the ith argument to a function application as long as all elements in $D_{\sigma_i}^{st} - Conc_{\sigma_i}(\overline{s_i})$ were preserved. If we are to find out the maximum possible amount of evaluation, then we will find the maximum $\overline{s_i}$ for which the theorem holds.

One way we can encode the information about changing the evaluation strategies is to label the arguments to a function and the apply nodes in any function application with the amount of evaluation it is safe to do if ever an application needs to be evaluated (c.f. [Burn, Hankin and Abramsky 1985a], [Hankin, Burn and Peyton Jones 1986]). We will now illustrate with a couple of examples how to determine and use the information available from the previous four theorems.

As our first example of the application of these theorems, we will work out the context-free and context-sensitive information for the function

$$g = \lambda f^{int \rightarrow int}.\lambda x^{int}.\lambda y^{int}.x + f(y)$$

which has graph [Wadsworth 1971] (omitting type information) :



and has abstract interpretation

$$\bar{g} = \lambda \bar{f}^{2\to 2}.\lambda \bar{x}^{2}.\lambda \bar{y}^{2}.\bar{x} \text{ and } \bar{f}(\bar{y})$$

as we saw in section 3.3. We test for context-free strictness of g in each parameter using Theorem 3.4.2.

$$\bar{g}\,(\lambda x^{2}.0)\ 1\ 1 = 1 \text{ and } (\lambda x^{2}.0)\ 1$$

$$= 0$$

$$\bar{g}\,(\lambda x^{2}.1)\ 0\ 1 = 0 \text{ and } (\lambda x^{2}.1)\ 1$$

$$= 0$$

$$\bar{g}\,(\lambda x^{2}.1)\ 1\ 0 = 1 \text{ and } (\lambda x^{2}.1)\ 0$$

$$= 1$$

So we see that g is context-freely strict in its first and second parameters, but not in its third.

For the context-sensitive strictness information we have three apply nodes in the body of f :

$+ \ @\ x\ (f\ y)$
$(+\ x)\ @\ (f\ y)$
$f\ @\ y$

where we have denoted the apply node of interest using the "@" symbol as in the graphical representation of g. We note that in these expressions we have free variables, and this will be the general case. The semantic function for variables is :

$$E^{ab}\,[[x^{\sigma}]]\,\rho^{ab} = \rho^{ab}(x^{\sigma})$$

and we have to guarantee that

$$\rho^{ab}\,[[x^{\sigma}]] \geqslant abs_{\sigma}(\rho^{st}\,[[x^{\sigma}]])$$

for the conditions of the theorems which ensure correctness to hold. Since we are not sure what values will be taken by the free variables in the standard semantics, for they may take on any value, then we must set the abstract interpretation of all free variables to be the top of the relevant domain to ensure that the above condition is true.

From Definition 3.2.1.1 we have that

$$E^{ab}\,[[+]]\,\rho^{ab} = \lambda u^{2}.\lambda v^{2}.u \text{ and } v$$

We use Theorem 3.4.3 to test the apply nodes. Firstly we test the apply node between the + and x. In the expression we test both f and y are free, and so we must set their abstract interpretation to $\lambda\bar{x}^{2}.1$ and 1 respectively.

$$(E^{ab}\,[[+]]\rho^{ab})\;0\;(E^{ab}\,[[(f\;y)]]\rho^{ab}) = (\lambda u^{2}.\lambda v^{2}.u \text{ and } v)\,@\,((\lambda\bar{x}^{2}.1)\;1)$$

$$= (\lambda u^{2}.\lambda v^{2}.u \text{ and } v)\;0\;1$$

$$= 0$$

We next test the top application node in the graph of the function. In this case x is free and so we must set its abstract interpretation to 1.

$$E^{ab}\,[[+]]\,\rho^{ab}\;x\;0 = (\lambda u^{2}.\lambda v^{2}.u \text{ and } v)\;1\;0$$

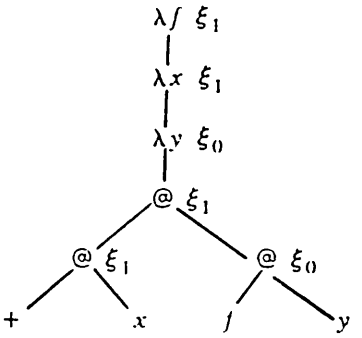
$$= 0$$

For the final apply node, we have that f is free, and so we must set its abstract interpretation to $\lambda x^{2}.1$ to obtain

$$(E^{ab}\,[[f]]\,\rho^{ab})\;0 = (\lambda x^{2}.1)\;0$$

$$= 1$$

Using the theorems on changing the evaluation strategy, we can label the function graph as follows :

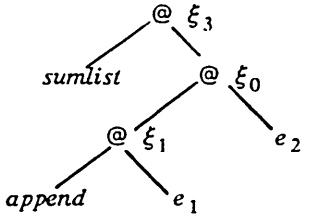


This says that in any application of the function it is safe to evaluate using ξ_1 the first and second arguments to g, but it is not safe to do any evaluation of the third argument. The annotations in the function body say that it is safe to evaluate the expression $(f\ y)$ and the expression x using ξ_1 when evaluating the function body, but it is not safe to evaluate the expression y.

In a similar manner, using the abstract interpretations of *sumlist* and *append* given in section 3.3, the context-sensitive definedness theorem (Theorem 3.4.3), and the context-sensitive change of evaluation strategy theorem (Theorem 3.5.2), we are able to label the application

$$sumlist(append(e_1, e_2))$$

as in the diagram :



This says that when we have to evaluate the application, then its is safe to initiate a computation of $append(e_1, e_2)$ using ξ_3, and evaluation of e_1 using ξ_1, but it is not safe to do any evaluation of e_2.

3.6. More Abstract Domains for Base Types.

By taking *Alist* as the example of a type which has as its interpretation an infinite sum of products, and choosing the four evaluators for lists and the abstract domain to model the way the evaluators behave, we have left some questions unanswered. Specifically, what is to be done about other types which have as a standard interpretation a domain which is an infinite sum of products, and, can we find out extra information

about nested structures which will allow us to do more evaluation?

Another example of a data type with a domain which is an infinite sum of products as its standard interpretation is *Atree* :

$$Atree \cong 1 + A \times Atree \times Atree$$

which are trees of elements of A. It may have as constructors for the type :

$$nil\_tree : \rightarrow Atree$$

and

$$node : A \rightarrow Atree \rightarrow Atree \rightarrow Atree$$

In section 3.1.2 we saw that one way of developing abstract domains was to look at the sensible evaluators for elements of a type. Just as there are four sensible evaluators for *Alist*, for *Atree* there are also four sensible evaluators :

(*i*) no evaluation;

(*ii*) evaluate to head normal form, that is, as far as a *nil_tree* or a *node* ;

(*iii*) evaluate the *shape* of the tree i.e. recursively unfold the second and third arguments to *node* until a *nil_tree* is reached in each one; and

(*iv*) evaluate the shape of the tree, and evaluate each of the elements from D_A^{st} in the nodes to head normal form.

These evaluators are sensible because they treat each node of a tree in the same way.

Again we can see how these evaluators divide up the standard interpretation of *Atree* - the evaluator in (*i*) preserves all elements of the standard interpretation; the evaluator in (*ii*) preserves all non-bottom elements of the domain; the evaluator in (*iii*) preserves all trees where each branch of the tree ends in a *nil_tree* after a finite number of nodes; and the evaluator in (*iv*) preserves only those elements which are preserved by (*iii*) which also have no $\perp_{D_A^{st}}$ elements in the nodes. The way the evaluators have divided the standard interpretation is exactly analogous to the way the evaluators for lists broke up the domain, and so the four point domain is also a suitable abstract domain for the type *Alist*. To use this domain for the abstract interpretation of programs which have functions over *Atree*, we need to define the constants that are used and their standard interpretation. We can then develop abstract interpretations of the constants by calculating the abstraction of their standard interpretation as we did for the constants in section 3.2. Correctness of the definedness interpretation for programs using this type would then follow immediately, and we could use Theorems 3.5.1 and 3.5.2 to label a

program to indicate safe changes of evaluation strategy.

Although our discussion so far has been of the type *Atree*, we hope this has motivated the claim that for any similar type which has as its standard interpretation a domain which is an infinite sum of products, there are only four sensible evaluators, where (*ii*) to (*iv*) above are modified to recursively unfold each of the recursive parts of the type definition. Thus the four point domain is a sensible abstract domain for such types. To finish the abstract interpretation one has to provide the information about the constants as has been done for *Alist* and mentioned above for *Atree*.

Let us now examine more closely the four point domain we have used for the abstract interpretation of lists. The first two points in the abstract domain, $\bot_L$ and I give us information about the definedness of the top-level structure of a list only, namely whether it is totally undefined or whether it is partial or infinite; the other two points also give us information about the definedness of elements of the list, namely whether there are any bottom elements in the list or not. These last two points allowed us to find that sometimes we could use the evaluator ξ_3 which evaluated the elements of the list to head normal form.

If the elements of the list were themselves lists (or some other data type which was an infinite sum of products), then there are some extra evaluators that we could use :

(*v*) evaluate the spine of the list and evaluate each element the list using ξ_2; and

(*vi*) evaluate the spine of the list and evaluate each element of the list using ξ_3.

Our intuition by now should tell us that in order to be able to capture how these evaluators treat lists we need to add some points to the abstract domain to test the definedness of functions on elements which these evaluators do not preserve. For example, if the type A was *Blist*, where

$$Blist \cong 1 + B \times Blist,$$

then we may wish to additionally find out how defined functions were when applied to an element of type *Alist* which was finite but for which the minimally defined element (i.e. a *Blist*) was partial or infinite, or if the minimally defined element was finite but had bottom elements (i.e. from D_B^{st}). In an analogous manner to the way we replaced the two point domain by the four point domain for lists so that we could talk of the definedness of elements of a list, we can replace the top two points of our four point domain for lists with another four point domain which gives information about the definedness of the elements of an *Alist* which are themselves lists. We could thus define the abstract domain $D_{Alist}^{ab} = \{\bot_L, I, FB, FI, FF, \top_L\}$, where $\bot_L \leqslant I \leqslant FB \leqslant FI \leqslant FF \leqslant \top_L$, and the

abstraction map :

$$abs_{Alist} : D^{st}_{Alist} \rightarrow D^{ab}_{Alist}$$

in the obvious way.

Clearly the process could be continued according to the depth of nesting of (type) structures in the program. We however note that extra points in the abstract domain mean that the process of finding fixed points in calculating the abstract interpretation will take longer. Furthermore, the more evaluators we have, the more complex the hardware which supports the evaluation of the functional language has to be. The choice to how much information is sought is thus a pragmatic one; the theory presented in this chapter is sufficiently general to support any of the abstract domains we may choose.

We note that the abstract domains mentioned above for nested structures are also due to [Wadler 1985]. Their motivation using sensible amounts of evaluation is new to this work.

3.7. Relationship to Other Work.

Much of the work done in this area has been to do with strictness analysis, or the detection of when a function is strict in an argument. The first work was presented in [Mycroft 1981], where his framework was applied to strictness analysis of first-order functions. In [Burn, Hankin and Abramsky 1985a] and [Hankin, Burn and Peyton Jones 1986] theory was developed to allow for strictness analysis of higher-order functions. Similarly, the framework of [Maurer 1985] was developed for strictness analysis, and the framework of [Abramsky 1985a] was applied to strictness analysis.

A big breakthrough was made by Wadler [Wadler 1985] in using the four point domain, also used in this chapter, as the abstract domain for lists. He was able to give abstract interpretations for the constants, and interpreted the results of the abstract interpretation, but no formal justification for his assertions was given. We have provided the justification in this chapter.

Finally, the work in [Hankin, Burn and Peyton Jones 1986] provided a way of encoding the strictness information (which arguments could be safely evaluated to head normal form given that we had to evaluate an application to head normal form) using the **P** combinator(†). An extension of this needs to be made to cover the extra evaluators, which is probably just having three "flavours" of the **P** combinator.

(†) The **P** combinator has the same reduction rule as **I** but sets off a parallel process to evaluate its second argument.

3.8. Conclusion.

We have applied the abstract interpretation framework of Chapter 2 to see how much evaluation of arguments in a function application is safe given that we have to evaluate the function application.

Abstract domains and abstraction maps were defined which captured the distinctions between the elements of the standard domain made by the sensible evaluation strategies. The abstract domains that we discussed, the two point domain and the four point domain, are due to [Mycroft 1981] and [Wadler 1985] respectively.

The abstract interpretations of constants were derived by taking the abstraction of their standard interpretation. For all of the constants except *case*, we were able to define abstract interpretations of the constants which were equal to the abstraction of the standard interpretation. Because *case* is an higher-order function in which the functional argument is applied to one of the other arguments (actually the head and the tail of one of the other arguments), we find that our requirement of safety means that the equality does not hold. However, since we only needed that

$$E^{ab}\,[[c_\sigma]]\,\rho^{ab} \geqslant abs_\sigma(E^{st}\,[[c_\sigma]]\,\rho^{st}),$$

this is no problem.

We proved the correctness of this interpretation as a simple corollary of the Correctness Theorem for Abstract Interpretation from Chapter 2. The theorems concerning context-sensitive and context-free definedness information follow in a similar manner.

The definedness information was then used to show when it was safe to change evaluation strategies from a left-most outer-most strategy to one where some of the inner redices were also reduced.

Finally we note that the pragmatics given in this chapter asked how much evaluation was safe of arguments to a function given that we had to "evaluate" a function application. Given a function application

$$f\,e_1\,\cdots\,e_n\,,$$

putting $\perp$ on the right-hand side of the test effectively meant we were finding out how much evaluation of e_i was safe given that we knew only that it was safe to evaluate the application to head normal form. If e_i was a function application, and it was possible to evaluate e_i using ξ_2 or ξ_3, then we would like to utilise this information, because it may allow more evaluation of its arguments to be done. This is the subject of the next

chapter; in fact it leads to a much more natural understanding of what is going on.

Chapter 4

Evaluation Transformers

In the last chapter we developed a definedness interpretation and used it to see when we could change the evaluation strategy by evaluating some of the arguments to a function in parallel with evaluating the function application. The two theorems for changing the evaluation strategy were developed assuming that we had to do some evaluation of the function application, but ignored the amount of evaluation of the application of the function that was safe. A simple example in the next section shows intuitively that if we take into account the amount of evaluation it is safe to do of a function application, then we can sometimes allow stronger evaluators to be applied to the arguments of a function. Moreover, because a function application can appear in several differing contexts, it is no longer sufficient to just label functions with information about how much evaluation it is safe to do of the arguments because at various times different amounts of evaluation may be safe for an application of a function. We thus introduce *evaluation transformers* for each argument of a function and application node of a program which, given a safe evaluator for an application, will transform it into safe evaluators for the arguments in the function application.

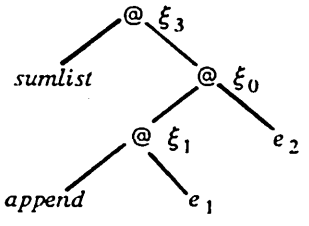
Evaluation transformers can be determined statically from the program text using the definedness abstract interpretation of Chapter 3. Two theorems which are generalisations of the Change of Evaluation Strategy theorems of Chapter 3 give us a method for determining evaluation transformers.

4.1. Motivation for Evaluation Transformers.

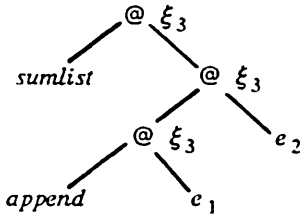
In the section 3.5 we saw that that it was possible to label the applications in the expression :

$$sumlist(append(e_1, e_2))$$

as in the diagram :



This says that when evaluating the expression, we can initiate the computation of $append(e_1.e_2)$ using ξ_3, the evaluation of e_1 by ξ_1 and that we could do no evaluation of e_2. However, we know that the reason why we could evaluate $append(e_1.e_2)$ using ξ_3 was that *sumlist* needed to have a finite list with no bottom elements in it or else an expression $sumlist(e)$ would be undefined. So really we need that $append(e_1.e_2)$ should have as semantics a finite list with no bottom elements in it or else the above expression will be undefined. The only way *append* can return such a list is if both of its arguments are lists which are finite and have no bottom elements in them. Thus, in this case it is safe to label the applications in the expression $append(e_1.e_2)$ with ξ_3 because this preserves all of the elements for which $append(e_1.e_2)$ is a finite list with bottom elements, and these are all of the elements that need to be preserved for an application of *sumlist*. We could then label the application nodes of the expression as in



Taking into account the amount of evaluation it is safe to do of a function application, we are able to use a stronger evaluator in the evaluation of e_1 and e_2 than we were able to conclude in the last chapter where we did not specify the amount of evaluation that it was safe to do of a function application.

A function application may occur in several different contexts in a program, which may not be statically detectable. For example, we may have the definition :

$$e = append(e_1.e_2)$$

where e_1 and e_2 are some expressions, and then find that the expressions $sumlist(e)$ and $hd(e)$ appear somewhere else in the program. From the above discussion we can see that in the first case both e_1 and e_2 can be evaluated safely using ξ_3, while in the latter case it is only safe to evaluate e to head normal form and so the original annotations on the applications are the maximally safe annotations. It is thus no longer good enough to use labels which give a fixed amount of evaluation, but we have to introduce evaluation transformers.

Definition 4.1.1:

Given a safe evaluator for a function application, an *evaluation transformer* for the ith argument in the function application specifies a safe evaluator for the ith argument.

□

Note that we are now interpreting the abstract interpretation in a "backwards" manner. We are asking, given that an amount of evaluation is safe for a function application, what amount of evaluation is safe for each of the arguments in the application. The question being asked in the last chapter then was how much evaluation of arguments to a function in an application is safe given that some unspecified amount of evaluation had to be done.

We can obtain evaluation transformers from the abstract interpretation developed in the last chapter.

4.2. Determining Evaluation Transformers.

As in Chapter 3, where we had two theorems for the safe changing of evaluation strategy, we give here two theorems to determine respectively the context-free and context-sensitive evaluation transformers. After giving each theorem we will give an example of its use in determining an evaluation transformer.

4.2.1. Context-free Evaluation Transformers.

We have the following theorem to determine context-free evaluation transformers :

Theorem 4.2.1.1: (Context-Free Evaluation Transformer Theorem)

Suppose that $f : \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \tau$ and that ξ is a safe evaluator for an application of f. Furthermore, suppose that ξ does not preserve any of the elements in $Conc_\tau(\bar{t})$. If

$$E^{ab}[[f]]\rho^{ab} \top_{D^{ab}_{\sigma_1}} \cdots \top_{D^{ab}_{\sigma_{i-1}}} \overline{s_i} \top_{D^{ab}_{\sigma_{i+1}}} \cdots \top_{D^{ab}_{\sigma_n}} \leqslant \bar{t}$$

then in any application of f where ξ is a safe amount of evaluation for the application, any evaluation strategy which evaluates the ith argument to f so as to preserve all of the values in $D^{s'}_{\sigma_i} - Conc_{\sigma_i}(\overline{s_i})$ is a safe evaluation strategy.

Proof :

Given any application $f\ e_1\ \cdots\ e_n$ there are two cases.

(i) Suppose $E^{st}\ [[e_i]]\ \rho^{st} \in Conc_{\sigma_i}(\overline{s_i})$. Then the evaluation of e_i may initiate a non-terminating computation. However, in this case, we are assured by Theorem 3.4.2 that

$$E^{st}\ [[f\ e_1\ \cdots\ e_n]]\ \rho^{st} \in Conc_{\tau}(\overline{t}).$$

Since ξ is was a safe amount of evaluation for the application, we have that the original expression must have had bottom as its semantics because ξ does not preserve any elements in $Conc_{\tau}(\overline{t})$. Thus it is safe to initiate a non-terminating computation when evaluating the ith argument.

(ii) If $E^{st}\ [[e_i]]\ \rho^{st} \notin Conc_{\sigma_i}(\overline{s_i})$ then, since we have chosen an evaluator which preserves non-$\overline{s_i}$ values, no divergent computation will be initiated in evaluating e_i and so the evaluation strategy is safe.

□

If there is no such $\overline{s_i}$ then it is not safe to do any evaluation of the ith argument. By choosing the most defined $\overline{s_i}$ for which it is true, then we will be able to find out the maximum permissible amount of evaluation.

We will now give a detailed example of how to use this theorem, and then give the context-free evaluation transformers for the functions for which we gave abstract interpretations in Chapter 3.

As the example, we will use the *append* function, where the values of $\overline{append}\ \overline{L_1}\ \overline{L_2}$ are given in Table 3.3.4, reproduced below as Table 4.2.1.1.

Table 4.2.1.1
Abstract Interpretation of *append*

$\overline{L_2}$ \ $\overline{L_1}$	$\bot_L$	I	F	$\top_L$
$\bot_L$	$\bot_L$	I	I	I
I	$\bot_L$	I	I	I
F	$\bot_L$	I	F	F
$\top_L$	$\bot_L$	I	F	$\top_L$

We note that in this case $\top_{D^{sl}_{Alist}}$ is $\top_L$. The elements of type *Alist* which each evaluator preserves are listed in Table 3.1.2.2.

Suppose that we have that ξ_1 is a safe evaluator, where $\perp_{D^{sl}_{Alist}}$ is the only element which is not preserved by ξ_1. For the first argument we must therefore find the maximum $\bar{L}$ such that

$$\overline{append}\ \bar{L}\ \top_L \leqslant \perp_L$$

From Table 4.2.1.1 we find that the maximum element is $\perp_L$, and thus we must choose an evaluator which preserves all elements in $D^{sl}_{Alist} - Conc_{Alist}(\perp_L)$. The maximum evaluator which satisfies this restriction is ξ_1, that is, evaluating a list to head normal form.

For the second argument we must find the maximum $\bar{L}$ such that

$$\overline{append}\ \top_L\ \bar{L} \leqslant \perp_L$$

We see that there is no such $\bar{L}$, and so we conclude that it is not safe to do any evaluation of the second argument to append in this case.

If we denote the evaluation transformer for the i th argument to a function f by F_i, so far we have that:

$$APPEND_1(\xi_1) = \xi_1$$

$$APPEND_2(\xi_1) = \xi_0$$

Similarly, ξ_2 does not preserve any elements in $Conc_{Alist}(I)$. The maximum $\bar{L}$ such that

$$\overline{append}\ \bar{L}\ \top_L \leqslant I$$

is I and so the evaluation of the first argument to *append* must preserve all values in $D^{sl}_{Alist} - Conc_{Alist}(I)$. The evaluator which allows maximum evaluation is ξ_2.

Likewise, the maximum $\bar{L}$ such that

$$\overline{append}\ \top_L\ \bar{L} \leqslant I$$

is I. Thus we have

$$APPEND_1(\xi_2) = \xi_2$$

$$APPEND_2(\xi_2) = \xi_2$$

The final case for which we have to do any work is for ξ_3 which does not preserve any elements in $Conc_{Alist}(F)$. The maximum $\bar{L}$ such that

$$\overline{append}\ \bar{L} \top_L \leqslant F$$

is F and so the evaluation of the first argument to *append* must preserve all elements in $D^{sl}_{Alist} - Conc_{Alist}(F)$. The mechanism which allows maximum evaluation is ξ_3. Similarly, the maximum $\bar{L}$ such that

$$\overline{append} \top_L \bar{L} \leqslant F$$

is F. Thus we have

$$APPEND_1(\xi_3) = \xi_3$$

$$APPEND_2(\xi_3) = \xi_3$$

Because the evaluator ξ_0 has to preserve all elements of D^{sl}_{Alist}, we can do no evaluation of either argument of *append* in this case.

These results are gathered together in Table 4.2.1.5.

In a similar manner, we can determine the evaluation transformers for the functions whose abstract interpretations we gave in Chapter 3. The following five tables give the context-free evaluation transformers for these functions, where Table 4.2.1.n+1 gives the evaluation transformers for the functions whose abstract interpretation appears in Table 3.3.n. Recall that we denote the evaluation transformer for the ith argument to a function f by F_i.

Table 4.2.1.2 gives the evaluation transformers for functions f satisfying the conditions of Definition 3.2.1.1, for example +, × and or.

Table 4.2.1.2
Context-free Evaluation Transformers for Binary Strict Functions

E	$F_1(E)$	$F_2(E)$
ξ_0	ξ_0	ξ_0
ξ_1	ξ_1	ξ_1

Evaluation transformers for functions of type $Alist \rightarrow A$ where the abstract interpretation of the type A is 2 are given in Table 4.2.1.3.

Table 4.2.1.3
Context-free Evaluation Transformers for *sumlist*, *length* and *hd*

E	$SUMLIST_1(E)$	$LENGTH_1(E)$	$HD_1(E)$
ξ_0	ξ_0	ξ_0	ξ_0
ξ_1	ξ_3	ξ_2	ξ_1

Table 4.2.1.4 gives the evaluation transformers for functions of type $Alist \rightarrow Alist$.

Table 4.2.1.4
Context-free Evaluation Transformers for *reverse* and *tl*

E	$REVERSE_1(E)$	$TL_1(E)$
ξ_0	ξ_0	ξ_0
ξ_1	ξ_2	ξ_1
ξ_2	ξ_2	ξ_2
ξ_3	ξ_3	ξ_2

The evaluation transformers for *append* that we obtained earlier in this section are given in Table 4.2.1.5.

Table 4.2.1.5.
Context-free Evaluation Transformers for *append*

E	$APPEND_1(E)$	$APPEND_2(E)$
ξ_0	ξ_0	ξ_0
ξ_1	ξ_1	ξ_0
ξ_2	ξ_2	ξ_2
ξ_3	ξ_3	ξ_3

Finally, the evaluation transformers for *map* are given in Table 4.2.1.6.

Table 4.2.1.6
Context-free Evaluation Transformers for *map*

E	$MAP_1(E)$	$MAP_2(E)$
ξ_0	ξ_0	ξ_0
ξ_1	ξ_0	ξ_1
ξ_2	ξ_0	ξ_2
ξ_3	ξ_0	ξ_2

4.2.2. Context-sensitive Evaluation Transformers.

Theorem 4.2.2.1: (Context-Sensitive Evaluation Transformer Theorem)

Suppose that $f : \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \tau$ and that ξ is a safe evaluator for an application $f\, e_1 \cdots e_n$. Furthermore, suppose that ξ does not preserve any of the elements in $Conc_\tau(\bar{\iota})$. If

$$E^{ab}[[f]]\,\rho^{ab}\,E^{ab}[[e_1]]\,\rho^{ab}\cdots E^{ab}[[e_{i-1}]]\,\rho^{ab}\,\overline{s_i}\,E^{ab}[[e_{i+1}]]\,\rho^{ab}\cdots E^{ab}[[e_n]]\,\rho^{ab} \leqslant \bar{\iota}$$

then it is safe to use an evaluation strategy which evaluates e_i so as to preserve all values in $D^{st}_{\sigma_i} - Conc_{\sigma_i}(\overline{s_i})$.

Proof :

The proof follows exactly as in the proof of Theorem 4.2.1.1 except that we appeal to Theorem 3.4.3 instead of Theorem 3.4.2.

□

If there is no such $\overline{s_i}$ then it is not safe to do any evaluation of the expression e_i.

The reason why we have context-sensitive theorems is that they give us more information than context-free ones in the case that we use more complex domains than the two point domain for the base types or, if we use higher-order functions where a functional argument is applied to an expression in the body of the function. As an example of the use of this theorem, we will use an application of the function *map* :

$map\ f\ [\,] = [\,]$
$map\ f\ x{:}xs = (f\ x) : (map\ f\ xs)$

The abstract interpretation, $\overline{map}$, of *map* is given in Table 4.2.2.1, which is just a copy of Table 3.3.5.

Table 4.2.2.1
Abstract Interpretation of *map*

$\bar{L} \backslash \bar{f}$	$\lambda\bar{x}^2.0$	$\lambda\bar{x}^2.x$	$\lambda\bar{x}^2.1$
$\perp_L$	$\perp_L$	$\perp_L$	$\perp_L$
I	I	I	I
F	F	F	$\top_L$
$\top_L$	$\top_L$	$\top_L$	$\top_L$

We will determine the context-sensitive evaluation transformer for the second argument to *map* in the application

*map plus*1 *e*

where *plus*1 is the strict function

$$plus1\ n = n+1$$

and so

$$E^{ab}\,[[plus1]]\,\rho^{ab} = \lambda x^2.x.$$

If the evaluator for the application is ξ_0, then it is not safe to do any evaluation of the second argument, and so the evaluation transformer is ξ_0 at ξ_0.

The evaluator ξ_1 does not preserve any elements in $Conc_{Alist}(\perp_L)$, and so from the theorem we must find an $\bar{L}$ such that

$$E^{ab}\,[[map]]\,\rho^{ab}\,E^{ab}\,[[plus1]]\,\rho^{ab}\,\bar{L} \leqslant \perp_L$$

that is,

$$\overline{map}\,(\lambda x^2.x)\,\bar{L} \leqslant \perp_L$$

From Table 4.2.2.1 we find that the maximum such $\bar{L}$ is $\perp_L$ and so we must choose an evaluator which preserves all elements in $D^{St}_{Alist} - Conc_{Alist}(\perp_L)$. The strongest evaluator given in Table 3.1.2.2 satisfying this property is ξ_1.

No elements in $Conc_{Alist}(I)$ are preserved by ξ_2 and so we must find the maximum $\bar{L}$ such that

$$\overline{map}\,(\lambda x^2.x)\,\bar{L} \leqslant I$$

to find the value of the evaluation transformer at ξ_2. The most defined $\bar{L}$ for which this is the case is I and so any evaluator must preserve all elements in $D^{st}_{Alist} - Conc_{Alist}(I)$. From Table 3.1.2.2 the strongest evaluator with this property is ξ_2.

Finally, ξ_3 preserves no elements in $Conc_{Alist}(F)$ and so we must find the most defined $\bar{L}$ such that

$$\overline{map}\,(\lambda x^2.x)\;\bar{L} \leqslant F.$$

Here F is the maximum element, and so as ξ_3 is the strongest evaluator which preserves all elements in $D^{st}_{Alist} - Conc_{Alist}(F)$, we may use it.

Thus the evaluation transformer for the second argument to *map* in this application is :

Table 4.2.2.2
A Context-sensitive Evaluation Transformer for *map*

E	$MAP_2(E)$
ξ_0	ξ_0
ξ_1	ξ_1
ξ_2	ξ_2
ξ_3	ξ_3

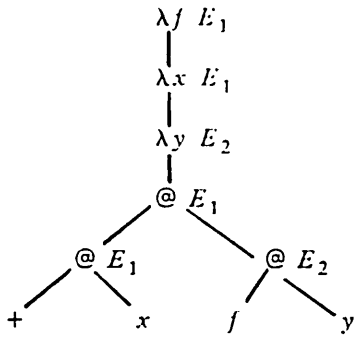
If one compares this with the context-free evaluation transformer for the second argument of *map* given in Table 4.2.1.6, then we can see that a stronger evaluator is allowed (ξ_3 instead of ξ_2) when ξ_3 is a safe evaluator for the application. This is because the test for the context-free evaluation transformer had to use $\lambda x^2.1$ in the test of the second argument, and a function which is defined everywhere and a strict function only differ in the way they behave in terms of the definedness of the result on finite lists with bottom elements in them.

4.2.3. Using Evaluation Transformers.

Evaluation transformers are used to label arguments to functions and application nodes in the same way that we labelled them with evaluators in Chapter 3. It is important to note that labelling with evaluation transformers replaces labelling with evaluators, that is, we label with evaluation transformers and not evaluators. In section 3.5 we labelled the function

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

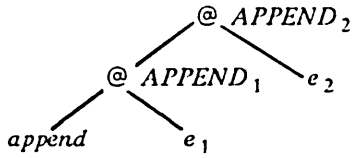
If, for the purposes of this example, we denote by E_1 the evaluation transformer which sends ξ_1 to ξ_1 and ξ_0 to ξ_0, and by E_2 the evaluation transformer which sends both ξ_1 and ξ_0 to ξ_0, then we can label the graph of g as in the diagram :



This says that if ever the evaluator ξ_1 is used to evaluate an application of g, then the first and second parameters to f can be evaluated using ξ_1, since $E_1(\xi_1) = \xi_1$, while the third parameter may not be evaluated for $E_2(\xi_1) = \xi_0$. The body of the function is then being evaluated by ξ_1, which means that the subexpressions x and $(f\ y)$ can be evaluated using ξ_1.

In this example we are able to see one of the pragmatic problems which still has to be tackled with parallel machines. The evaluation strategy described above will try and initiate the evaluation of the expression which is substituted for the variable x twice, which at best is a communication overhead on the machine. It would be valuable to see if we could detect cases like the above where the evaluation of the expression which is the second parameter to g has always been initiated and so we can change the label E_1 to E_2 on the application node which has x as its operand to prevent the evaluation strategy trying to initiate its computation again.

As another example of using evaluation transformers, suppose we have labelled the application $append(e_1.e_2)$ as shown in the diagram :



so that in this application the context-sensitive evaluation transformers are the same as the context-free evaluation transformers, given in Table 4.2.1.5. If, for example, the application was to be evaluated using ξ_2, then e_1 could be evaluated using $APPEND_1(\xi_2)$

$= \xi_2$, and e_2 could be evaluated using $APPEND_2(\xi_2) = \xi_2$. At another time, ξ_1 may be a safe evaluator for the application and so $\xi_1 = APPEND_1(\xi_1)$ is a safe evaluator for e_1, while $\xi_0 = APPEND_2(\xi_1)$ is a safe evaluator for e_2.

4.3. Values Used for Determining Evaluation Transformers.

If $f : \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \tau$, then we have to determine an evaluation transformer for each of f's n arguments. These evaluation transformers must give an evaluator for each of the evaluators for the type τ.

The evaluation transformers always give ξ_0 at ξ_0 because if it is not safe to do any evaluation of a function application, then it is not safe to do any evaluation of any of the arguments to the function(†).

If ξ is any other evaluator for the type τ, then Theorem 4.2.2.1 says that we must choose a $\bar{t}$ such that ξ does not preserve any elements in $Conc_\tau(\bar{t})$. We choose the most defined such $\bar{t}$ because that will allow us to detect the maximum amount of evaluation that is safe. For example, for types which have the evaluator ξ_1 as the only evaluator which does any evaluation, we can see from Table 3.1.2.1 that the appropriate $\bar{t}$ is $\perp_{D_\tau^{ab}}$, while for lists, from Table 3.1.2.2 we see that the values for $\bar{t}$ are $\perp_L$, I and F for ξ_1, ξ_2 and ξ_3 respectively.

For any particular evaluator, where we have fixed a $\bar{t}$, we then have to find out which evaluator is safe for the ith argument. Both Theorems 4.2.1.1 and Theorem 4.2.2.2 say that we must find an $\bar{s_i}$ which satisfies the test and then choose some evaluator preserves all of the elements of $D^{st}_{Alist} - Conc_{\sigma_i}(\bar{s_i})$. Thus for the type *Alist* there are three values that can be tried, namely $\perp_L$, I and F, while for types where ξ_1 is the only evaluator besides ξ_0, then we have only to try $\perp_{D_\tau^{ab}}$. (See Tables 3.1.2.2 and 3.1.2.1 respectively.) This is why in strictness analysis ([Mycroft 1981], [Burn, Hankin and Abramsky 1985a]), the description of using strictness analysis speaks of putting the bottom of the appropriate type for the argument being tested and seeing if bottom is returned by the abstract interpretation, for this is the only case to test when we have a two point domain as the abstract interpretation for all base types. We will of course choose the maximum $\bar{s_i}$ for which the appropriate test is true because this will allow

(†) If the argument is totally defined, then this is not true; it would be safe to evaluate the argument. However, the fact that it is not safe to do any evaluation of the function application indicates that the value of the function application may not be needed and so the value of the argument may not be needed. Thus any computation of the argument may be wasted. Furthermore, we have not developed an abstract interpretation which allows us to infer definite termination. However, see [Mycroft and Nielson 1983] and [Abramsky 1985b].

maximum possible evaluation of an argument.

Because all of the functions involved are monotonic, the quickest way to do the tests is to start off with the minimum value from the set of values we have to test for $\bar{t}$ and start testing the for the maximum $\bar{s_i}$ for which the condition holds with the minimum value from the set of values for the type σ_i. When we have found the minimum value, we can begin testing for the next minimal $\bar{t}$, beginning with the value for $\bar{s_i}$ which we had for the last one.

4.4. Evaluation Transformers, "Need" Labels and the P Combinator.

The methodology of strictness analysis as given in [Mycroft 1981] and [Burn, Hankin and Abramsky 1985a] tested the strictness of a function by putting in the bottom of the appropriate type and seeing if the result in the abstract interpretation was bottom; if it was, then that argument to the function could be evaluated when the function application was evaluated.

We can now see that this is equivalent to asking how much evaluation can be done to arguments of a function given that a function application could be evaluated, and given that there was only one evaluator, ξ_1, which did any non-trivial amount of evaluation. Thus the "need" labels of [Burn, Hankin and Abramsky 1985a], and their encapsulation using the P combinator of [Hankin, Burn and Peyton Jones 1986] are just evaluation transformers which send ξ_1 to ξ_1 and, of course, ξ_0 to ξ_0. The unlabelled applications then are the evaluation transformers which send both ξ_1 and ξ_0 to ξ_0.

4.5. Relationship to Other Work.

The idea that we should be looking at how much evaluation it is safe to do of the arguments to a function given that it is safe to evaluate a function application is also tackled by Hughes in [Hughes 1985]. In his paper he introduces several things that are similar to those which we have introduced. As a guide to the change in terminology, we note that what Hughes calls "contexts" roughly correspond to what we call evaluators, and what he terms "strictness functions" we have called context-free evaluation transformers.

There are several things to note about the work he presents. Firstly, one might expect that he can derive more exact information; it looks like his method should be able to determine evaluation strategies which for instance evaluated every second element of a list argument in parallel with the function application. We have restricted ourselves to evaluators which treat every element of the list in a uniform manner.

Sometimes recursive "contexts" are defined, for example

$$spine = nil\ OR\ cons\ ABSENT\ spine$$

which says roughly to evaluate the argument down to a *nil* or a *cons* and then evaluate the tail of the list in the same manner, doing no evaluation of the head of the list (i.e. our evaluator ξ_2). If $APPEND_1$ denotes the evaluation transformer associated with the first argument of *append*, then he is able to deduce that

$$APPEND_1\ spine = spine.$$

This can be translated into our terminology to say that if it is safe to use ξ_2 for the evaluation of an application of *append*, then it is safe to evaluate the first argument to *append* using ξ_2.

Surprisingly, in some cases the results he obtains lose more information than our method does. The example which shows this is due to [Hughes 1985]. For *reverse* in the context *cons c ABSENT*, which says that we evaluate the head of the list using the context c and do no evaluation of the tail, and so is stronger than our evaluator ξ_1 (i.e. evaluate the list to head normal form), the following context is obtained :

$$REVERSE_1\ (cons\ c\ ABSENT) = g\ where\ g = ABSENT\ OR\ nil\ OR\ cons\ h\ g$$
$$h = c\ OR\ ABSENT\ OR\ nil\ OR\ cons\ h\ g$$

Ignoring the finer details, the fact that we have *ABSENT "OR"*'d in at the top level means that it is not safe to do any evaluation of the argument to reverse when in this context. However, with our analysis, we are able to show that with the weaker safe evaluator ξ_1 for an application of reverse, it is still safe to evaluate the spine of the list which is the argument to reverse.

A second thing we note is that [Hughes 1985] is a framework for first order functions only, whereas we have presented a framework which handles higher-order functions. As well, we note that the distinction between context-sensitive and context-free issues is not made.

Finally, our work rests on firm semantic foundations, whereas that of [Hughes 1985] has not had such foundations fully worked out. If the ideas of [Hughes 1985] were formalised then a proper comparison of the two pieces of work could be made.

4.6. Conclusion.

A very natural question to ask of a function application is how much evaluation is safe for each of the arguments to the function given that a particular amount of evaluation is safe for the function application. If this is more than no evaluation, then we may change the evaluation strategy to do some evaluation of the arguments to the function in parallel with (or before) the evaluation of the function application. In general, a function application may used be in many different contexts which have different safe evaluators for that function application. Thus we need the concept of an evaluation transformer for each argument of a function, which transforms an evaluator which is safe for the application into one which is safe for the evaluation of the argument.

As with previous chapters, we have given theorems which allow us to determine both context-free and context-sensitive evaluation transformers. The evaluation transformers are determined using the definedness interpretation of Chapter 3.

The P combinator of [Hankin, Burn and Peyton Jones 1986] can be seen as being an evaluation transformer when we only use the two point domain for the abstract interpretation of all base types and thus are only distinguishing between the evaluation strategy which does no evaluation of an argument to a function and the evaluation strategy which evaluates an argument to head normal form using a left-most outer-most strategy in parallel with, or before he evaluation of the function application.

Chapter 5

Abstract Interpretation and New Type Constructors

So far we have been developing a framework where we have assumed we were given standard and abstract interpretations of the base types, and then we added the function space as the only type constructor. We were able to deal with complex data types such as infinite lists (infinite sums of products) by putting them in a black box and treating them as a base type by giving them a standard and abstract interpretation. In this chapter, we are slightly more imaginative, allowing types to be constructed from finite combinations of base types using sums, products and lifting. Abstract interpretations and abstractions of the standard interpretation of the structured types are defined in a natural way using the interpretations and abstractions of their components. We show that the vital relationship between the standard and abstract interpretation, namely that for all $e{:}\sigma$ we have

$$E^{ab}\,[[e]]\,\rho^{ab} \geqslant abs_{\sigma}(E^{st}\,[[e]]\,\rho^{st})$$

still holds, and thus the correctness of the abstract interpretation follows as in Chapter 2.

We must begin by giving the syntax for our new constructs and their interpretations.

5.1. Syntax of Type Constructs.

Here we introduce the syntax for the rest of our language which will include constructs to express elements of new structured data types formed from finite combinations of products, sums and lifting.

We note that products come in two flavours, the cartesian product, which we will denote as $\times$, and the smash product, which we will denote by $\otimes$. Sums also come in two flavours, the separated sum, denoted by $+$, and the coalesced sum, denoted by $\oplus$. However, as we have that

$$D^I_{\sigma} + D^I_{\tau} \cong (D^I_{\sigma})_{\perp} \oplus (D^I_{\tau})_{\perp}$$

for the interpretations I that we will consider, we will from now on only concern ourselves with the coalesced sum.

The following syntactic formation rules extend the rules (1) to (5) given in section 1.5.1 for our language *Exp* :

(6) $\dfrac{s : \sigma \quad t : \tau}{<s,t> : \sigma \times \tau}$ cartesian product

(7) $\dfrac{s : \sigma \quad t : \tau}{\langle\!\langle s,t \rangle\!\rangle : \sigma \otimes \tau}$ smash product

(8) $\dfrac{s : \sigma}{i(s) : \sigma \oplus \tau} \qquad \dfrac{t : \tau}{j(t) : \sigma \oplus \tau}$ coalesced sum

(9) $\dfrac{s : \sigma}{<0,s> : \sigma_\perp}$ lifting

5.2. Interpretation of New Constructs.

Interpretations for the type constructors and for the above syntactic constructs must be given.

We begin by giving the interpretation, I, of the type constructs. We interpret the product and lifting constructors in the usual way :

$$(D_{\sigma \times \tau})^I = D_\sigma^I \times D_\tau^I$$

$$(D_{\sigma \otimes \tau})^I = D_\sigma^I \otimes D_\tau^I$$

$$(D_{\sigma_\perp})^I = (D_\sigma^I)_\perp$$

where $\times$, $\otimes$ and $\perp$ on the right-hand side are the usual domain operations [Scott 1981, 1982].

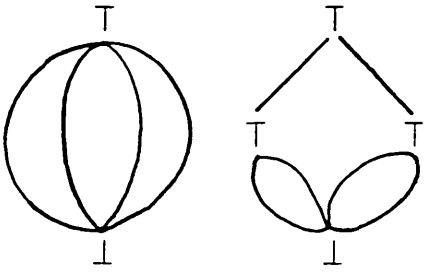
We note that $D_\sigma^I \times D_\tau^I$ and $D_\sigma^I \otimes D_\tau^I$ are finite, complete lattices if and only if D_σ^I and D_τ^I are, and that $(D_\sigma^I)_\perp$ is a finite, complete lattice if and only if D_σ^I is. Thus these constructions preserve the requirements for our abstract domains that they be finite, complete lattices.

The coalesced sum is slightly problematical. For the standard interpretation, we can have

$$(D_{\sigma \oplus \tau})^{st} = D_\sigma^{st} \oplus D_\tau^{st}$$

where the $\oplus$ on the right-hand side is the usual domain theoretic $\oplus$. This interpretation of $\oplus$ however does not have the property that $D_\sigma^I \oplus D_\tau^I$ is a complete lattice when D_σ^I and D_τ^I are complete lattices. Therefore, we must instead use a complete lattice theoretic $\oplus$ in the abstract interpretation, which only differs from the domain theoretic interpretation in terms of what it does with the top elements of D_σ^I and D_τ^I. With the lattice theoretic $\oplus$ we have a choice between coalescing the tops of the two complete

lattices, or adding a separated top to the lattice on top of the result of using the domain theoretic $\oplus$, as depicted in the following diagram :



coalesced top *separated top*

If we choose a separated top, then the abstraction maps we will define can never be onto for types including a sum. While this is no problem for the theory we have developed, it runs against the philosophy of only having elements in the abstract domain which represent some object in the standard domain. For this reason we choose the $\oplus$ in the abstract domain which coalesces the tops of the two component complete lattices.

We note that the isomorphism

$$D^I_{\sigma+\tau} \cong D^I_{\sigma_\perp \oplus \tau_\perp}$$

still holds provided we are consistent in what we do with the tops on both sides of the isomorphism.

Having given interpretations to our type constructors, we can now add the following semantic equations to those in section 1.5.2 :

$$E^I\,[[<s,t>]]\,\rho^I = <E^I\,[[s]]\,\rho^I, E^I\,[[t]]\,\rho^I>$$

$$E^I\,[[\lfloor s,t \rfloor]]\,\rho^I = \lfloor E^I\,[[s]]\,\rho^I, E^I\,[[t]]\,\rho^I \rfloor$$

$$E^I\,[[i(s)]]\,\rho^I = i(E^I\,[[s]]\,\rho^I)$$

$$E^I\,[[j(t)]]\,\rho^I = j(E^I\,[[t]]\,\rho^I)$$

$$E^I\,[[<0,s>]]\,\rho^I = <0, E^I\,[[s]]\,\rho^I>$$

where we have used the same symbols on the right-hand side to denote operations on the appropriate domains as we used for the syntactic constructs; which we are using will be obvious from the context.

5.3. Abstraction Maps For These Constructions.

All the constructs we have been discussing in this chapter are functors on the category of domains, which means that as well as acting on objects (domains), they work on morphisms between domains (continuous maps). In particular, we have that they work on our continuous abstraction maps. It thus trivial to define the abstraction maps for these constructed domains :

Definition 5.3.1:

$$abs_{\sigma\times\tau} : D^{st}_{\sigma\times\tau} \to D^{ab}_{\sigma\times\tau}$$

$$abs_{\sigma\times\tau} = abs_\sigma \times abs_\tau$$

□

Definition 5.3.2:

$$abs_{\sigma\otimes\tau} : D^{st}_{\sigma\otimes\tau} \to D^{ab}_{\sigma\otimes\tau}$$

$$abs_{\sigma\otimes\tau} = abs_\sigma \otimes abs_\tau$$

□

Definition 5.3.3:

$$abs_{\sigma\oplus\tau} : D^{st}_{\sigma\oplus\tau} \to D^{ab}_{\sigma\oplus\tau}$$

$$abs_{\sigma\oplus\tau} = abs_\sigma \oplus abs_\tau$$

□

Definition 5.3.4:

$$abs_{\sigma_\perp} : D^{st}_{\sigma_\perp} \to D^{ab}_{\sigma_\perp}$$

$$abs_{\sigma_\perp} = (abs_\sigma)_\perp$$

(where if $f: S \to T$ then $f_\perp : S_\perp \to T_\perp$). □

The definitions of Abs_σ and $Conc_\sigma$ for each type σ are as in Definitions 2.2.4 and 2.2.5, that is :

Definition 5.3.5:

$$Abs_\sigma : \mathbf{P}\, D^{st}_\sigma \to D^{ab}_\sigma$$

$$Abs_\sigma = \bigsqcup \circ \mathbf{P}\, abs_\sigma$$

□

Definition 5.3.6:

$$Conc_\sigma : D_\sigma^{ab} \rightarrow \mathbf{P}\, D_\sigma^{st}$$

$$Conc_\sigma(\vec{s}) = \bigcup \{T \mid Abs_\sigma(T) \leqslant \vec{s}\}$$

□

In Chapter 2 we required that the abstraction maps be strict and continuous. The following lemma about the continuity and strictness of the abstraction and concretisation maps is just Lemma 2.2.8 generalised to include all of the new type constructs.

Lemma 5.3.7:

If for each base type A, we are given a strict, continuous abstraction map $abs_A : D_A^{st} \rightarrow D_A^{ab}$, then for all types σ,

(*i*) abs_σ is continuous.

(*ii*) Abs_σ is continuous.

(*iii*) abs_σ and Abs_σ are strict.

(*iv*) $Conc_\sigma$ is well-defined and continuous.

Proof :

We have to add the following into the inductive steps of the proof of Lemma 2.2.8 :

(*i*) $abs_{\sigma\times\tau}$, $abs_{\sigma\otimes\tau}$, $abs_{\sigma\oplus\tau}$ and $abs_{\sigma_\perp}$ are continuous because they are each constructed by a (different) functor acting on continuous maps.

(*ii*) As in the proof of Lemma 2.2.8 (*ii*).

(*iii*) $abs_{\sigma\times\tau}$, $abs_{\sigma\otimes\tau}$, $abs_{\sigma\oplus\tau}$ and $abs_{\sigma_\perp}$ are strict because the appropriate functors preserve strictness.

(*iv*) As in the proof of Lemma 2.2.8 (*iv*).

□

5.4. A Relationship Between $abs_{\sigma\to\tau\to\mu}$ and $abs_{\sigma\times\tau\to\mu}$.

There is a well known relationship between $D^I_{\sigma\to\tau\to\mu}$ and $D^I_{\sigma\times\tau\to\mu}$, namely that

$$D^I_{\sigma\to\tau\to\mu} \cong D^I_{\sigma\times\tau\to\mu}$$

The isomorphism from left to right is often called *uncurry*, and its inverse *curry*.

There is also a relationship between $abs_{\sigma\to\tau\to\mu}$ and $abs_{\sigma\times\tau\to\mu}$ which is stated in the following Proposition :

Proposition 5.4.1:

$$abs_{\sigma\to\tau\to\mu} = curry \circ abs_{\sigma\times\tau\to\mu} \circ uncurry$$

Proof :

Let $f \in D^{St}_{\sigma\to\tau\to\mu}$. Then, by Proposition 2.3.1 we have that

$(curry \circ abs_{\sigma\times\tau\to\mu} \circ uncurry)\,(f)$

$= curry(\lambda <\overline{x_1}.\overline{x_2}>^{D^{ab}_{\sigma\times\tau}} .\bigsqcup\{abs_\mu(uncurry(f)\ <x_1.x_2>) \mid abs_{\sigma\times\tau}(<x_1.x_2>) \leqslant <\overline{x_1}.\overline{x_2}>\}^\circ)$

$= curry(\lambda <\overline{x_1}.\overline{x_2}>^{D^{ab}_{\sigma\times\tau}} .\bigsqcup\{abs_\mu(f\ x_1\ x_2) \mid abs_{\sigma\times\tau}(<x_1.x_2>) \leqslant <\overline{x_1}.\overline{x_2}>\}^\circ)$

by the definition of *uncurry*

$= curry(\lambda <\overline{x_1}.\overline{x_2}>^{D^{ab}_{\sigma\times\tau}} .\bigsqcup\{abs_\mu(f\ x_1\ x_2) \mid <abs_\sigma(x_1).abs_\tau(x_2)> \leqslant <\overline{x_1}.\overline{x_2}>\}^\circ)$

by the definition of $abs_{\sigma\times\tau}$

$= curry(\lambda <\overline{x_1}.\overline{x_2}>^{D^{ab}_{\sigma\times\tau}} .\bigsqcup\{abs_\mu(f\ x_1\ x_2) \mid abs_\sigma(x_1) \leqslant \overline{x_1}.\ abs_\tau(x_2) \leqslant \overline{x_2}\}^\circ)$

by the definition of $\leqslant$ on products

$= \lambda \overline{y_1}^{D^{ab}_\sigma} .\lambda \overline{y_2}^{D^{ab}_\tau} .$

$[(\lambda <\overline{x_1}.\overline{x_2}>^{D^{ab}_{\sigma\times\tau}} .\bigsqcup\{abs_\mu(f\ x_1\ x_2) \mid abs_\sigma(x_1) \leqslant \overline{x_1}.\ abs_\tau(x_2) \leqslant \overline{x_2}\}^\circ)\ <\overline{y_1}.\overline{y_2}>]$

by definition of *curry*

$= \lambda \overline{y_1}^{D^{ab}_\sigma} .\lambda \overline{y_2}^{D^{ab}_\tau} .\bigsqcup\{abs_\mu(f\ x_1\ x_2) \mid abs_\sigma(x_1) \leqslant \overline{y_1}.\ abs_\tau(x_2) \leqslant \overline{y_2}\}^\circ$

$= abs_{\sigma\to\tau\to\mu}(f)$ by Proposition 2.3.1

□

5.5. Properties of Abstraction Maps.

As in section 2.4, some properties of abstraction maps on lower types are carried over to the abstraction maps on constructed types. We give two examples here. Also, concretisation is strict.

Lemma 5.5.1:

If for each base type A abs_A is $\perp$-reflecting, then abs_σ and Abs_σ are $\perp$-reflecting for each type σ.

Proof:

We must add an inductive step for each of the new type constructs into the proof of Lemma 2.4.4. Since abs_A is strict for each base type A, we have from Lemma 5.3.7 (i) that abs_σ is strict for all types σ. We will thus prove for each type σ that if $s \neq \perp_{D_\sigma^{st}}$ then $abs_\sigma(s) \neq \perp_{D_\sigma^{ab}}$ which, together with strictness, implies $\perp$-reflexivity.

$(i) <s,t> \neq \perp_{D_{\sigma\times\tau}^{st}}$ $\Rightarrow$ $not(s = \perp_{D_\sigma^{st}}$ **and** $t = \perp_{D_\tau^{st}})$

$\Rightarrow$ $not(abs_\sigma(s) = \perp_{D_\sigma^{ab}}$ **and** $abs_\tau(t) = \perp_{D_\tau^{ab}})$

since abs_σ, abs_τ are $\perp$-reflexive by the inductive hypothesis

$\Rightarrow$ $not(<abs_\sigma(s), abs_\tau(t)> = <\perp_{D_\sigma^{ab}}, \perp_{D_\tau^{ab}}>)$

$\Rightarrow$ $not(abs_{\sigma\times\tau}(<s,t>) = \perp_{D_{\sigma\times\tau}^{ab}})$

(ii) $<s,t> \neq \perp_{D_{\sigma\otimes\tau}^{st}}$ $\Rightarrow$ $not(s = \perp_{D_\sigma^{st}}$ **or** $t = \perp_{D_\tau^{st}})$

$\Rightarrow$ $not(abs_\sigma(s) = \perp_{D_\sigma^{ab}}$ **or** $abs_\tau(t) = \perp_{D_\tau^{ab}})$

since abs_σ, abs_τ are $\perp$-reflexive by the inductive hypothesis

$\Rightarrow$ $not(\{abs_\sigma(s), abs_\tau(t)\} = \perp_{D_{\sigma\otimes\tau}^{ab}})$

$\Rightarrow$ $not(abs_{\sigma\otimes\tau}(<s,t>) = \perp_{D_{\sigma\otimes\tau}^{ab}})$

(iii) If $u \in D^{st}_{\sigma\oplus\tau}$, $u \neq \perp_{D^{st}_{\sigma\oplus\tau}}$, then either $u = i(s)$ where $s \in D^{st}_{\sigma}$ and $s \neq \perp_{D^{st}_{\sigma}}$ or $u = j(t)$ where $t \in D^{st}_{\tau}$ and $t \neq \perp_{D^{st}_{\tau}}$. Supposing the former, then

$$abs_{\sigma\oplus\tau}(i(s)) = i(abs_{\sigma}(s))$$

$$\neq i(\perp_{D^{ab}_{\sigma}})$$

since abs_{σ} is $\perp$-reflexive by the inductive hypothesis

and so is not equal to $\perp_{D^{ab}_{\sigma\oplus\tau}}$. Similarly for the latter case.

(iv) $u \in D^{st}_{\sigma_\perp}$, $u \neq \perp_{D^{st}_{\sigma_\perp}}$ implies $u = <0,s>$ where $s \in D^{st}_{\sigma}$. Thus

$$abs_{\sigma_\perp}(u) = abs_{\sigma_\perp}(<0,s>)$$

$$= <0,abs_{\sigma}(s)>$$

$$\neq \perp_{D^{ab}_{\sigma_\perp}}$$

Abs_{σ} is $\perp$-reflexive by Lemma 2.4.3.

□

The following Lemma is a generalisation of Lemma 2.4.6 to include the type constructors we have added in this chapter.

Lemma 5.5.2:

If abs_A is onto for each base type, and for each base type A we can define a continuous function $abs_A^{-1} : D_A^{ab} \rightarrow D_A^{st}$ which is a right inverse of abs_A, that is, $abs_A \circ abs_A^{-1} = id_{D_A^{ab}}$, then for all types σ

(i) If σ is a function type, then there is a continuous function $abs_{\sigma}^{-1} : D^{ab}_{\sigma} \rightarrow D^{st}_{\sigma}$ which is a right inverse of abs_{σ}.

(ii) abs_{σ} and Abs_{σ} are onto.

(iii) $Abs_{\sigma} \circ Conc_{\sigma} = id_{D^{ab}_{\sigma}}$

Proof :

The proofs of parts (*i*) and (*iii*) follow exactly as in the proof of Lemma 2.4.6. We need to add the steps (*a*) to (*d*) below to the inductive step of part (*ii*) to show the ontoness of the abstraction maps for each of the constructs we have added in this chapter.

(*a*) Suppose $<\bar{s},\bar{t}> \in D^{ab}_{\sigma\times\tau}$. Then since abs_σ and abs_τ are onto by the inductive hypothesis, there exists $s \in D^{st}_\sigma$ and $t \in D^{st}_\tau$ such that $abs_\sigma(s) = \bar{s}$ and $abs_\tau(t) = \bar{t}$. So, by the definition of $abs_{\sigma\times\tau}$, we have that $abs_{\sigma\times\tau}(<s,t>) = <\bar{s},\bar{t}>$, and hence $abs_{\sigma\times\tau}$ is onto.

(*b*) abs_σ and abs_τ are monotonic and onto, and so this means that both of them are strict. Since the $\otimes$ of two strict functions is strict, we have that

$$abs_\sigma \otimes abs_\tau(\perp_{D^{st}_{\sigma\otimes\tau}}) = \perp_{D^{ab}_{\sigma\otimes\tau}}.$$

If $\{\!|\bar{s},\bar{t}|\!\} \in D^{ab}_{\sigma\otimes\tau}$, and $\{\!|\bar{s},\bar{t}|\!\} \neq \perp_{D^{st}_{\sigma\otimes\tau}}$, then since abs_σ and abs_τ are onto by the inductive hypothesis, there exists $s \in D^{st}_\sigma$ and $t \in D^{st}_\tau$ such that $abs_\sigma(s) = \bar{s}$ and $abs_\tau(t) = \bar{t}$. So, by the definition of $abs_{\sigma\otimes\tau}$, we have that $abs_{\sigma\otimes\tau}(\{\!|s,t|\!\}) = \{\!|\bar{s},\bar{t}|\!\}$. Hence $abs_{\sigma\otimes\tau}$ is onto because we have provided for every element in $D^{ab}_\sigma \otimes D^{ab}_\tau$ an element in $D^{st}_\sigma \otimes D^{st}_\tau$ which abstracts to it.

(*c*) abs_σ and abs_τ are monotonic and onto, and so this means that both of them are strict. Since the $\oplus$ of two strict functions is strict, we have that

$$abs_\sigma \oplus abs_\tau(\perp_{D^{st}_{\sigma\oplus\tau}}) = \perp_{D^{ab}_{\sigma\oplus\tau}}.$$

If $\bar{u} \in D^{ab}_{\sigma\oplus\tau}$, $\bar{u} \neq \perp_{D^{ab}_{\sigma\oplus\tau}}$, we have that either $\bar{u} = i(\bar{s})$, $\bar{s} \in D^{ab}_\sigma$ and $\bar{s} \neq \perp_{D^{ab}_\sigma}$, or $\bar{u} = j(\bar{t})$, $\bar{t} \in D^{ab}_\tau$ and $\bar{t} \neq \perp_{D^{ab}_\tau}$. Since abs_σ is onto, there exists an $s \in D^{st}_\sigma$ such that $abs_\sigma(s) = \bar{s}$. Then $abs_{\sigma\oplus\tau}(i(s)) = i(abs_\sigma(s)) = i(\bar{s})$. The case where $u = j(t)$ is treated in a similar manner. Thus $abs_{\sigma\oplus\tau}$ is onto because we have exhibited for every element of $D^{ab}_{\sigma\oplus\tau}$ an element in $D^{st}_{\sigma\oplus\tau}$ which abstracts to it.

(*d*) Suppose that $<0,\bar{s}> \in D^{ab}_{\sigma_\perp}$. Then, since abs_σ is onto, there exists an $s \in D^{st}_\sigma$ such that $abs_\sigma(s) = \bar{s}$. Thus $abs_{\sigma_\perp}(<0,s>) = <0,abs_\sigma(s)> = <0,\bar{s}>$ as required. Also, $abs_{\sigma_\perp}(\perp_{D^{st}_{\sigma_\perp}}) = \perp_{D^{ab}_{\sigma_\perp}}$ by the definition of the lifting of a function. Hence $abs_{\sigma_\perp}$ is onto.

The ontoness of Abs_σ follows from Lemma 2.4.5.

□

Fact 5.5.3:

For all types σ, $Conc_\sigma(\perp_{D_\sigma^{ab}}) = \{\perp_{D_\sigma^{st}}\}$ (Lemma 2.4.7).

□

5.6. Correctness Results.

We are now able to show that the relationship that was shown to hold in section 2.7 between the standard and abstract interpretation holds in this case as well.

Theorem 5.6.1:

Suppose that we have that $E^{ab}[[c_\sigma]]\rho^{ab} \geqslant abs_\sigma(E^{st}[[c_\sigma]]\rho^{st})$ for all constants c_σ. Then for all $\rho^{st} \in Env^{st}$, $\rho^{ab} \in Env^{ab}$ such that for all x^τ, $\rho^{ab}(x^\tau) \geqslant abs_\tau(\rho^{st}(x^\tau))$, we have for all $e : \sigma$:

$$E^{ab}[[e]]\rho^{ab} \geqslant abs_\sigma(E^{st}[[e]]\rho^{st})$$

Proof :

We insert the following steps for the constructs introduced in this chapter into the structural induction proof of Theorem 2.7.1, where the numbers are those given to the syntactic constructs in section 5.1.

(6) $E^{ab}[[<s,t>]]\rho^{ab} = <E^{ab}[[s]]\rho^{ab}, E^{ab}[[t]]\rho^{ab}>$

$\geqslant <abs_\sigma(E^{st}[[s]]\rho^{st}), abs_\tau(E^{st}[[t]]\rho^{st})>$ induction hypothesis

$= abs_{\sigma\times\tau}(<E^{st}[[s]]\rho^{st}, E^{st}[[t]]\rho^{st}>)$

$= abs_{\sigma\times\tau}(E^{st}[[<s,t>]]\rho^{st})$

(7) $E^{ab}[[\lambda s,t\rho]]\rho^{ab} = \lambda E^{ab}[[s]]\rho^{ab}, E^{ab}[[t]]\rho^{ab}\rho$

$\geqslant \lambda abs_\sigma(E^{st}[[s]]\rho^{st}), abs_\tau(E^{st}[[t]]\rho^{st})\rho$ induction hypothesis

$= abs_{\sigma\otimes\tau}(\lambda E^{st}[[s]]\rho^{st}, E^{st}[[t]]\rho^{st}\rho)$

$= abs_{\sigma\otimes\tau}(E^{st}[[\lambda s,t\rho]]\rho^{st})$

(8) $E^{ab}[[i(s)]]\rho^{ab} = i(E^{ab}[[s]]\rho^{ab})$

$$\geqslant i(abs_{\sigma}(E^{st}\ [[s]]\ \rho^{st})) \qquad \text{induction hypothesis}$$

$$= abs_{\sigma \oplus \tau}(i(E^{st}\ [[s]]\ \rho^{st}))$$

$$= abs_{\sigma \oplus \tau}(E^{st}\ [[i(s)]]\ \rho^{st})$$

and similarly for $j(t)$.

$$(9)\ E^{ab}\ [[<0,s>]]\ \rho^{ab} = <0, E^{ab}\ [[s]]\ \rho^{ab}>$$

$$\geqslant <0, abs_{\sigma}(E^{st}\ [[s]]\ \rho^{st})>$$

$$= abs_{\sigma_{\perp}}(<0, E^{st}\ [[s]]\ \rho^{st}>)$$

$$= abs_{\sigma_{\perp}}(E^{st}\ [[<0,s>]]\ \rho^{st})$$

□

The correctness of the abstract interpretation and the theorems regarding context-free and context-sensitive information follow exactly as in Chapter 2. We state Theorems 5.6.2 to 5.6.4 without proof because the proofs are the same as the proofs of Theorems 2.8.2, 2.9.1 and 2.9.2 respectively.

Theorem 5.6.2: (Correctness Theorem for Abstract Interpretation)

The abstract interpretation we have developed is correct. That is, given $f : \sigma \to \tau$ and interpretations of constants satisfying the conditions of Theorem 5.6.1, we have that if $\bar{s} \in D_{\sigma}^{ab}$ and $(E^{ab}\ [[f]]\ \rho^{ab})\ (\bar{s}) = \bar{t}$ then for all $s \in Conc_{\sigma}(\bar{s})$, $(E^{st}\ [[f]]\ \rho^{st})\ (s) \in Conc_{\tau}(\bar{t})$.

□

Theorem 5.6.3: (Context-Free Information Theorem)

If $f : \sigma_1 \to \cdots \sigma_n \to \tau$ and

$$(E^{ab}\ [[f]]\ \rho^{ab})\ \top_{D^{ab}_{\sigma_1}} \cdots \top_{D^{ab}_{\sigma_{i-1}}} \overline{s_i}\ \top_{D^{ab}_{\sigma_{i+1}}} \cdots \top_{D^{ab}_{\sigma_n}} = \bar{t}$$

then for all $e_j : \sigma_j$, $j \neq i$, for all $s_i \in Conc_{\sigma_i}(\overline{s_i})$, we have

$$E^{st}\ [[f]]\ \rho^{st}\ E^{st}\ [[e_1]]\ \rho^{st} \cdots E^{st}\ [[e_{i-1}]]\ \rho^{st}\ s_i\ E^{st}\ [[e_{i+1}]]\ \rho^{st} \cdots E^{st}\ [[e_n]]\ \rho^{st} \in Conc_{\tau}(\bar{t})$$

□

Theorem 5.6.4: (Context-Sensitive Information Theorem)

Given $f : \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \tau$ and an application $f\, e_1 \cdots e_n : \tau$, if

$$E^{ab}\,[[f]]\,\rho^{ab}\,E^{ab}\,[[e_1]]\,\rho^{ab} \cdots E^{ab}\,[[e_{i-1}]]\,\rho^{ab}\,\overline{s_i}\,E^{ab}\,[[e_{i+1}]]\,\rho^{ab} \cdots E^{ab}\,[[e_n]]\,\rho^{ab} = \overline{t}$$

then for all $s_i \in Conc_{\sigma_i}(\overline{s_i})$

$$E^{st}\,[[f]]\,\rho^{st}\,E^{st}\,[[e_1]]\,\rho^{st} \cdots E^{st}\,[[e_{i-1}]]\,\rho^{st}\,s_i\,E^{st}\,[[e_{i+1}]]\,\rho^{st} \cdots E^{st}\,[[e_n]]\,\rho^{st} \in Conc_{\tau}(\overline{t})$$

□

5.7. Abstract Domains, Abstraction Maps and Evaluators.

In section 3.1 we chose abstract domains which modelled the way that the evaluators broke up the standard interpretation of a type. We were then able to give a definedness interpretation and use this to find out when we could change evaluation strategies to do some evaluation of arguments to functions. While the abstract interpretations of domains constructed using the smash product, coalesced sum and lifting seem to preserve this property, domains constructed using the cartesian product do not.

The problem with the cartesian product is that there is no concept of head normal form, for the bottom element of the domain is just a pair of bottom elements. Thus the bottom element of the abstract domain, which has been used to show when we can evaluate expressions to head normal form, is redundant. If we were to change the interpretation of the cartesian product so that we stopped evaluation when we discovered we had a pair of elements, then we would be modelling a lifted cartesian product, where an extra bottom element has been added to the bottom of the domain.

A similar distinction is made by evaluating functions only as far as weak head normal form instead of as far as head normal form. According to the usual semantics of the function space [Scott 1981, 1982], the function which returns the value $\perp$ for all arguments is the bottom of the function space. However, evaluation of such an expression would stop at $\lambda x.e$, where e has $\perp$ as its semantics if only evaluating expressions to weak head normal form. This appears to add an extra bottom in the formation of the function space. Further investigation is necessary to find out the relationship between head normal form and weak head normal form.

5.8. Relationship to Other Work.

We have taken a very simple approach to the abstract interpretation of the extra type constructs that we have added to our language in this chapter where, for instance, products still stay products, albeit in a more restricted category. This is considerably less general than the work of [Nielson 1984], where the interpretation of these type constructors is not so constrained.

5.9. Conclusion.

The work contained in this thesis has up to this chapter been assuming that the function space creator was the only data type constructor available in our language; all other more complex types were relegated to being "base" types. In this chapter we have extended the allowable types to include finite combinations of base types using sums, products and lifting. Because sums, products and lifting are functors, we can define in a natural way abstract domains and abstraction maps for types involving these constructors.

The correctness of the abstract interpretation and the context-free and context-sensitive information theorems follow exactly as in Chapter 2.

The methods used in this chapter cannot be extended in the natural way to recursively defined types because they result in infinite abstract domains, c.f. [Nielson 1984]. Therefore, recursive types must be given special abstract interpretations as has been done with *Alist* in this thesis.

Chapter 6

Further Work and Applications

We began this thesis by illustrating that the obvious way of obtaining parallelism in the evaluation of functional programs was not sensible because it created too many tasks, many of which were to evaluate expressions whose values were not needed. Our work has been to develop and apply a framework for abstract interpretation, which can be implemented in a compiler to annotate programs to show where it is safe to initiate parallel evaluation. The abstract interpretation can also be used to change the evaluation strategy for a sequential machine, moving from a lazily evaluated program to one which has a mixture of call-by-value and lazy evaluation.

In this chapter we summarise some of the theoretical issues which have still to be resolved, the most outstanding of which is a satisfactory treatment of polymorphic functions.

Already this work is being applied in projects on parallel architectures for running functional languages. Some of this work is briefly discussed in the final section.

6.1. Further Work.

6.1.1. Polymorphism.

In this thesis we have been using a mono-typed language, whereas we know that the function which calculates the length of a list :

$$length\ [] = 0$$
$$length\ x{:}xs = 1 + length\ xs$$

does not need to know what the type of the elements of the list is, but behaves in the same way on all types with the same top-level structure, that is, lists. One can then assign the *polymorphic type* [Milner 1978]

$$\alpha\text{-}list \rightarrow int$$

to the function *length*, which says that *length* is a function which will take a list of elements of any type α and return an integer. Anyone who has had to write programs in a strongly, monomorphically typed language will recognise what an advantage it is to be allowed to write polymorphically typed functions.

Since the framework we developed depended on functions having mono-types, we must find some way of handling polymorphically typed programs.

Our first solution is to notice that a polymorphically typed program can be uniquely expanded to a mono-typed program [Holmstrom 1983]. However, while this gives the advantages of polymorphism to the programmer, it gives none of the advantages to the compiler when doing the abstract interpretation - it must give an abstract interpretation for every type instantiation of a function.

It would be good if we could find a single expression for the abstract interpretation of a function which could be used for any type instantiation. However, as was shown in [Burn 1985], this is not in general possible for recursive functions, for the number of iterations of the fixed pointing algorithm goes up with the complexity of the type of the expression being fix pointed. A possible solution would be to try and find some sort of "most general type occurrence" of a function in a program and work out an expression for the abstract interpretation of this.

In [Abramsky 1985a] a framework for proving properties of functions which are *polymorphically invariant* is given. Basically, a property is polymorphically invariant if its truth at one particular instantiation of a type implies its truth at all possible instantiations of the type. He shows that strictness is such a property, which means we can test the strictness of a polymorphic function in its arguments at the simplest instantiation of the type and know the results hold for all higher types.

6.1.2. A Junction Between Operational and Denotational Semantics.

Throughout this thesis we have been using the idea of neededness in an intuitive manner. However, the idea of neededness has been formalised in some systems, for example [Huet and Levy 1979] and [Barendregt and Sleep 1986]. One could formalise the idea of neededness for the system we have been using, and see how the expressions which turn out to be needed correspond with those we have said are safely evaluable.

We believe that, if the semantics of an expression is not bottom, then our method for changing evaluation strategies will only allow at most as much evaluation of an expression as would have eventually been done by a lazy evaluator. However, this is only a conjecture and should be received as such. To prove its correctness or otherwise, an operational model must be constructed for the way annotated programs are evaluated and compared with an operational model for lazy evaluation. If the conjecture is true, then the final expressions that are obtained in evaluating an expression using the altered evaluation strategy and using the lazy strategy should be exactly the same; if they are not then one could try and find out in what way they differed and try to formulate some syntactic/operational notion of equivalence (because they will be semantically equivalent by our insistence on correct evaluation strategies).

6.1.3. Pragmatic Issues.

It was shown in section 3.6 that the framework we have developed will support finding out more information about nested structures than is available using the four point domain. Furthermore, in Chapter 5, we have added interpretations for base types constructed from finite combinations of sums, products and liftings. Adding extra evaluators means that they must be supported in the hardware of the machine. This means that more space will have to be allocated for evaluation transformers (because there are more of them), more complex labels must be placed on data objects (to show how much evaluation has been done), and many other similar issues. As well, the compiler must be adjusted to have all of these elements taken into account when working out the abstract interpretation. We note that having more points in the abstract domains means that taking fixed points takes much longer.

The evaluators ξ_2 and ξ_3 that we have used calculate the whole spine of a list. If the list is very long, then this may not be a space or processor efficient thing to do. Ideally, one would like to use some sort of bounded evaluator which only evaluated a list n elements ahead of the process which was consuming the list. Unfortunately, we cannot guarantee that we will only ever have to keep at most n elements of the list at a time because the list may be being used by more than one function, and we can quite easily write two functions which consume a list where the second function is k elements behind the first function in consuming the list for any k.

6.2. Applications.

As well as finding application in the work in ESPRIT Project 415 (Parallel Architectures and Languages for AIP - A VLSI-Directed Approach), the work of this thesis is being incorporated into several other research projects to do with parallel architectures for evaluating functional programs.

A simple form of strictness detection is being done for programs which are then compiled and run on the G-machine [Johnsson 1983], where arguments to strict operators and variables appearing in the condition of the conditional are detected as being evaluable. The exciting thing is that for this sequential machine, taking note of this strictness information is responsible for an improvement of an order of magnitude in the speed of running some programs [Johnsson 1986]. Only experiments can show how much improvement can be made with a full definedness analysis.

The GRIP (Graph Reduction in Parallel) Project [Peyton Jones, Clack and Salkild 1985] is a project to build a multi-processor machine to do graph reduction. Strictness

analysis [Burn, Hankin and Abramsky 1985a], [Hankin, Burn and Peyton Jones 1985] and the work of this thesis has influenced the way that the machine has been designed to run programs.

In general, the finding of fixed points is *n-exponentially complete* [Damm 1986], [Meyer 1985], but it is quite hard to construct functions for which the algorithm takes a significant number of steps. A SERC-funded project run by Chris Hankin at Imperial College is, amongst other things, implementing the abstract interpretation of this thesis. This will give us a better idea of the practicality of using this work in a production compiler. Furthermore, through a chain of other projects, code to run functional programs with lazy semantics in parallel on Alice [Darlington and Reeve 1981] will be produced. It will be the first parallel implementation of a lazy functional language on Alice.

The COBWEB machine [Shute 1983], [Hankin, Osmon and Shute 1985], [Karia 1986] is a parallel reduction machine designed specifically to be implemented on a wafer. Our work will influence the way programs are evaluated and the design of processing elements.

References

[Abramsky 1985a]

Abramsky, S., Strictness Analysis and Polymorphic Invariance, *Workshop on Programs as Data Objects*, DIKU, Denmark, 17-19 October, 1985, Ganzinger, H., and Jones, N.D., (eds.) Springer-Verlag LNCS 217, pp. 1-23.

[Abramsky 1985b]

Abramsky, S., *Abstract Interpretation, Logical Relations and Kan Extensions*, Draft Manuscript, Imperial College, University of London, October, 1985.

[Arbib and Manes 1975]

Arbib, M.A., and Manes, E.G., *Arrows, Structures and Functors : The Categorical Imperative*, Academic Press, New York and London, 1975.

[Barendregt 1984]

Barendregt, H.P., *The Lambda Calculus - Its Syntax and Semantics*, Studies in Logic and the Foundations of Mathematics, vol. 103, North-Holland, The Netherlands, 1984.

[Barendregt and Sleep 1986]

Barendregt, H.P., and Sleep, M.R., *Private Communication*, May, 1986.

[Burn 1985]

Burn, G.L., *Why the Problem of Polymorphism and Strictness Analysis Has Not Been Solved*, Note distributed on the FP mailboard, 20th November, 1985.

[Burn, Hankin and Abramsky 1985a]

Burn, G.L., Hankin, C.L., and Abramsky, S., Strictness Analysis for Higher-Order Functions, To appear in *Science of Computer Programming* Also : *Imperial College of Science and Technology, Department of Computing, Research Report DoC 85/6, April 1985*.

[Burn, Hankin and Abramsky 1985b]

Burn, G.L., Hankin, C.L., and Abramsky, S., The Theory of Strictness Analysis for Higher-Order Functions, *Workshop on Programs as Data Objects*, Copenhagen, Denmark, October 17-19, 1985, Ganzinger, H., and Jones, N.D., (eds.) Springer-Verlag LNCS 217, pp. 42-62.

[Church 1941]

Church, A., *The Calculi of Lambda Conversion*, Princeton University Press, Princeton, 1941.

[Cousot and Cousot 1979]

Cousot, P., and Cousot, R., Systematic Design of Program Analysis Frameworks, *Conference Record of the 6th ACM Symposium on Principles of Programming Languages*, pp. 269-282, 1979.

[Darlington and Reeve 1981]

Darlington, D., and Reeve, M., ALICE - A Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages, *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, New Hampshire, USA, October, 1981, pp. 65-75.

[Damm 1986]

Damm, W., *Personal Communication*, Semantics Working Group Meeting, ESPRIT Project 415, April 1986.

[Darlington and Reeve 1981]

Darlington, D., and Reeve, M., ALICE - A Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages, *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, New Hampshire, USA, October, 1981, pp. 65-75.

[Downey and Sethi 1976]

Downey, P.J., and Sethi, R., Correct Computation Rules for Recursive Languages, *SIAM Journal on Computing 5*, 3 (September), 1976, 378-401.

[Friedman and Wise 1976]

Friedman, D.P., and Wise, D.S., CONS Should Not Evaluate Its Arguments, *Automata, Languages and Programming*, Michaelson, S., and Milner, R., (eds.), Edinburgh University Press, Edinburgh, 1976.

[Gierz et. al. 1980]

Gierz, G., Hofmann, K.H., Keimel, K., Lawson, J.D., Mislove, M. and Scott, D.S., *A Compendium of Continuous Lattices*, Springer-Verlag, 1980.

[Guessarian 1981]

Guessarian, I., *Algebraic Semantics*, Springer Verlag LNCS 99, 1981.

[Hankin 1986]

Hankin, C.L., *Personal Communication*

[Hankin, Burn and Peyton Jones, 1986]

Hankin, C.L., Burn, G.L., and Peyton Jones, S.L., A Safe Approach to Parallel Combinator Reduction (Extended Abstract), *Proceedings ESOP 86 (European Symposium on Programming)*, Saarbrucken, Federal Republic of Germany, March 1986, Robinet, B., and Wilhelm, R. (eds.), Springer-Verlag LNCS 213, pp. 99-110.

[Hankin, Osmon and Shute 1985]

Hankin, C.L., Osmon, P.E., and Shute, M.J., COBWEB: A Combinator Reduction Architecture, in : *Proceedings of IFIP International Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, 16-19 September, 1985, Jouannaud, J.-P. (ed.), Springer-Verlag LNCS 201, pp. 99-112.

[Henderson and Morris 1976]

Henderson, P., and Morris, J.H., A Lazy Evaluator, *3rd ACM Conference on the Principles of Programming Languages*, Atlanta, Georgia, USA, 1976, pp.95-103.

[Hennessy and Plotkin 1979]

Hennessy, M., and Plotkin, G.D., Full Abstraction for a Simple Parallel Programming Language, Proceedings MFCS '79, Becvar, J. (ed.), *Springer Verlag LNCS 74*, 1979.

[Holmstrom 1983]

Holmstrom, S., *Polymorphic Type Systems and Concurrent Computation in Functional Langauges*, PhD Thesis, Department of Computer Sciences, Chalmers University of Technology, Goteborg, Sweden, 1983.

[Hudak and Young 1985]

Hudak, P., and Young, J., *A Set-Theoretic Characterisation of Function Strictness in the Lambda Calculus*, Research Report YALEU/DCS/RR-391, Department of Computer Science, Yale University, 1985, Also presented at the *Workshop on Abstract Interpretation*, University of Kent at Canterbury, August, 1985.

[Huet and Levy 1979]

Huet, G., Levy, J.-J., *Call by Need Computations in Non-Ambiguous Linear Term Rewriting Systems*, Rapport de Recherche No. 359, IRIA, Le Chesnay, France, 1979.

[Hughes 1984]

Hughes, R.J.M., *Why Functional Programming Matters*, Programming Methodology Group Memo PMG-40, Chalmers University of Tchnology, Goteburg, Sweden, 1984.

[Hughes 1985]

Hughes, R.J.M., Strictness Detection in Non-Flat Domains *Workshop on Programs as Data Objects*, DIKU, Denmark, 17-19 October, 1985, Ganzinger, H., and Jones, N.D., (eds.) Springer-Verlag LNCS 217, pp. 112-135.

[Johnsson 1983]

Johnsson, T., The G Machine : An Abstract Machine for Graph Reduction, *Declarative Programming Workshop*, University College London, 11-13 April, 1983.

[Johnsson 1986]

Johnsson, T., *Efficient Compilation of Lazy Evaluation*, Presentation given to the General Meeting of ESPRIT Project 415, April 1986.

[Karia 1986]

Karia, R.J., *An Investigation of Combintator Reduction on Multiprocessor Architectures*, PhD Thesis (In Preparation), University of London, 1986.

[Klop 1985]

Klop, J.W., *Term Rewriting Systems*, Notes prepared for the seminar on reduction machines, Ustica, September, 1985. (Klop works at CWI, Amsterdam.)

[Maurer 1985]

Maurer, D., Strictness Computation Using Special λ-Expressions, *Workshop on Programs as Data Objects*, DIKU, Denmark, 17-19 October, 1985, Ganzinger, H., and Jones, N.D., (eds.) Springer-Verlag LNCS 217, pp. 136-155.

[Meyer 1985]

Meyer, A.R., *Complexity of Program Flow-Analysis for Strictness : Application of a Fundamental Theorem of Denotational Semantics*, Draft Manuscript, MIT Lab for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, September, 1985.

[Milne and Strachey 1976]

Milne, R.E., and Strachey, C., *A Theory of Programming Language Semantics*, Chapman and Hall (UK), John Wiley (USA), 1976.

[Milner 1978]

Milner, R., A Theory of Type Polymorphism in Programming, *Journal of Computer and System Sciences 17*, 348-375, 1978.

[Mycroft 1981]

Mycroft, A., *Abstract Interpretation and Optimising Transformations for Applicative Programs*, PhD. Thesis, University of Edinburgh, 1981.

[Mycroft and Jones 1985]

Mycroft, A., and Jones, N.D., A new framework for abstract interpretation, *Workshop on Programs as Data Objects*, Copenhagen, Denmark, October 17-19, 1985, Ganzinger, H., and Jones, N.D., (eds.) Springer-Verlag LNCS 217, pp. 156-171.

[Mycroft and Nielson 1983]

Mycroft, A., and Nielson, F., Strong Abstract Interpretation Using Power Domains (Extended Abstract) *Proc. 10th International Colloquium on Automata, Languages and Programming : Springer Verlag LNCS 154*, Diaz, J. (ed.), Barcelona, Spain, 18th-22nd July, 1983, 536-547.

[Naur 1963]

Naur, P. (ed.), Revised report on the algorithmic language Algol 60, *Comm. ACM 6* 1 (Jan. 1963), 1-17.

[Nielson 1984]

Nielson, F., *Abstract Interpretation Using Domain Theory*, PhD Thesis, University of Edinburgh, 1984.

[Nielson 1986a]

Nielson, F., Abstract Interpretation of Denotational Definitions, *Proceedings STACS 1986*, Springer Verlag LNCS vol. 210.

[Nielson 1986b]

Nielson, F., *Strictness Analysis and Abstract Interpretation of Denotational Definitions*, Draft Manuscript, Institute of Electronic Systems, Aalborg University Centre, Aaborg, Denmark, June 1986.

[Peyton Jones 1984]

Peyton Jones, S.L., Comment made at the *Lispkit Open Forum*, University of Stirling, November 1983.

[Peyton Jones 1986]

Peyton Jones, S.L., *Implementing Functional Languages Using Graph Reduction*, To be published in the Prentice-Hall International Series in Computer Science, 1986.

[Peyton Jones, Clack and Salkild 1985]

Peyton Jones, S.L., Clack, C. and Salkild, J., *GRIP - A Parallel Graph Reduction Machine*, Dept of Computer Science, University College, London, November 1985.

[Plotkin 1976]

Plotkin, G.D., A Powerdomain Construction, *SIAM J. Comput.* 5 3 (Sept 1976) 452-487.

[Plotkin 1977]

Plotkin, G.D., LCF Considered as a Programming Language, *Theoretical Computer Science 5*, (1977), p. 223-255.

[Plotkin 1978]

Plotkin, G.D., *Lecture Notes on the Theory of Computation.*

[Plotkin 1980]

Plotkin, G.D., Lambda definability in the full type hierarchy, in : Seldin, J.P., Hindley, J.R., *To H.B. Curry: Essays on combinatory logic, lambda-calculus and formalism*, Academic Press, 1980.

[Scott 1981]

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

[Scott 1982]

Scott, D., Domains for Denotational Semantics, *Automata, Languages and Programming*, Proceedings of the 10th International Colloquium, Nielsen M, and Schmidt, E.M., (eds.), Springer-Verlag Lecture Notes in Computer Science, vol. 140, 1982, 577-613.

[Shute 1983]

Shute, M.J., *The Role of Simulation in the Study of Multiprocessor, Control Flow and Data Flow Systems*, PhD Thesis (2 volumes), University of London, 1983.

[Smyth 1978]

Smyth, M.B., Power Domains, *Journal of Computer and System Sciences 16*, 1978, 23-36.

[Smyth and Plotkin 1982]

Smyth, M.B., and Plotkin, G.D., The category-theoretic solution of recursive domain equations, *SIAM J. Comput. 11* 4 (1982), pp.761-783.

[Stoy 1977]

Stoy, J.E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge Massachusetts, 1977.

[Turner 1985]

Turner, D.A., Miranda(†): A non-strict functional language with polymorphic types, *Functional Programming Languages and Computer Architecture*, September 1985, Nancy, Jouannaud, J.-P., (ed.), Springer-Verlag LNCS 201, pp. 1-16.

[Vuillemin 1974]

Vuillemin, J., Correct and Optimal Implementations of Recursion in a Simple programming Language, *Journal of Computer and System Sciences 9*, 1974, 32-354.

[Wadsworth 1971]

Wadsworth, C.P., *Semantics and Pragmatics of the Lambda Calculus (Chapter 4)*, PhD Thesis, University of Oxford, 1971.

(†) Miranda is a trade mark of Research Software Ltd.