Abstract Interpretation of Polymorphic Higher-Order Functions

Gebreselassie Baraki

A Thesis submitted for the degree of Doctor of Philosophy
at the Department of Computing Science, University of Glasgow
February 1993

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Abstract

This thesis describes several abstract interpretations of polymorphic functions. In all the interpretations, information about any instance of a polymorphic function is obtained from that of the smallest, thus avoiding the computation of the instance directly. This is useful in the case of recursive functions, because it avoids the expensive computation of finding fixed points of functionals corresponding to complex instances.

We define an explicitly typed polymorphic language with the Hindley-Milner type system to illustrate our ideas, and provide two semantics of polymorphism that relate separate instances of any polymorphic function. The choice of which semantics to use depends on the particular program analysis we want to study.

For studying strictness analysis and binding-time analysis, we introduce a semantics based on embedding-closure pairs. We see how the abstract function of the smallest instance of a polymorphic function is used in building an approximation to that of any instance. Furthermore, we extend the language to include lists, and describe both strictness analysis and binding-time analysis of lists. Thus, this work extends previous work by others, on analyses of polymorphic first-order functions and also of monomorphic higher-order functions, to polymorphic higher-order functions.

In relating distinct instances of a polymorphic function, the approximate abstract function is expressed as the greatest lower bound of a set of functions. This may not be very cheap to compute. However, there are often ways of obtaining the same result by considering a smaller set of functions. Another issue concerns how close the approximations are to the exact values. In the first-order case, it is shown that the approximate values coincide with the exact values. In general this is not the case, but experimental results on strictness analysis indicate that good approximations are obtained.

Embedding-projection pairs are used to provide a semantics that is convenient for termination analysis of polymorphic functions. We show that the abstract interpretation of an instance can be approximated by the least upper bound of a set of functions that are built from that of the smallest.

Preface

The main results of the thesis deal with obtaining information about any instance of a polymorphic function from that of the smallest . In Chapter 1 we describe the problem and discuss related work. The mathematical terminology and concepts used in the thesis are given in Chapter 2, where the definitions and properties of some fairly well known structures are given.

In Chapter 3 we focus on those mathematical structures which we use to describe the semantics of polymorphic functions. In particular, we define the category of domains and embedding-closure pairs. This category and its subcategory of finite lattices are studied in some detail.

In Chapter 4 we introduce an explicitly typed polymorphic language, and define the semantic functions. Since our aim is to relate separate instances of any polymorphic function, embedding-closure pairs are used in Chapter 5 to establish such relationships.

In Chapter 6 we use a polymorphic language of abstract terms. Terms in the original language are translated into terms in this language. The values of the semantic functions on the abstract terms are the values of the abstract interpretation used in strictness analysis. Many of the results of the earlier chapter apply, and hence it is shown that the abstract interpretation of the smallest instance of a polymorphic function is used in building an approximation to that of any other instance. Therefore, computing complex instances directly may be avoided.

Strictness analysis of lists is discussed in Chapter 7. We show that the methods of the previous chapter also apply for polymorphic functions defined on lists. In Chapter 8, we show that the same results apply to the abstract interpretation used in binding-time analysis. Finally, we discuss termination analysis, where embedding-projection pairs are used instead.

In the last chapter, we give a summary of results and also discuss some problems. We also give the proofs of some propositions stated in the main body in the appendices.

Acknowledgements

I would like to thank my supervisor John Hughes for his help and encouragement. It was through his lectures that I was first introduced to abstract interpretation. Since his move to Sweden in the summer of 1992, John Launchbury took over as supervisor and I have had many enjoyable discussions with him. I am very grateful for his help during this period.

I am also grateful to many people in the department who helped me in many ways. In particular, I would like to thank Kieran Clenaghan, Phil Trinder, Keith van Rijsbergen, and Phil Wadler. Phil Trinder also read some parts of the thesis and made useful comments. My examiners, Geoffrey Burn and Phil Wadler, pointed out errors and suggested improvements for which I am grateful.

I would also like to thank Julian Seward of Manchester University for the discussions we have had in connection with strictness analysis of polymorphic functions. His implementation of the method proposed in this thesis enabled me to evaluate it, and also to debug some of the theory.

Many friends have been very supportive over the years. In particular, I would like to express my gratitude to Ruben Leon, Elizabeth Baker, Mary and Tesfay Waldemichael, and Anna and Tesfu Gessesse.

Finally, the financial support of Addis Ababa University and the UNDP is gratefully acknowledged.

Contents

Chapter 1

Background

The use of lazy functional programming languages as a convenient tool for software development has been strongly advocated [19]. The presence of laziness and higher-order functions allows the use of infinite data structures and greater modularity at the programming level. At the logical level, the absence of side-effects makes it easier to reason about programs. However, there are problems regarding the efficiency of their implementations. Efficient implementations of these languages rely on static program analysis. One such, which is considered to be important, is *strictness analysis.*

A function is said to be *strict* if it returns a non-terminating value for a non-terminating argument. If a function is strict, arguments may be passed by value instead of the less efficient evaluation mechanism call-by-need. This enables more efficient code to be generated. Moreover, parallel implementations may make use of strictness information of functions of several variables: if a function is known to be strict with respect to several arguments then its application can be computed by evaluating the expressions at such argument positions in parallel [13].

The two most important aspects of strictness analysis are: first, the development of algorithms which detect the strictness of functions, and secondly, a proof of correctness that such algorithms indeed give correct results. When one evaluation mechanism is to be used instead of another for efficiency reasons, it is necessary to ensure that both evaluation mechanisms give the same result. It is for this reason

that we need proofs of correctness.

Over the last decade several techniques for strictness analysis of functional programs have been developed. There are essentially three approaches :

- abstract interpretation
- projection-based analysis
- type inference

In the following sections, an introductory survey of work on abstract interpretation is presented. Emphasis is given to this approach because the thesis is mainly about abstract interpretation. Issues concerning higher-order functions, data structures and polymorphism are covered in some detail. The main contribution of this thesis is with regard to polymorphic functions; the central problem addressed is discussed in the section on polymorphism. A summary of the discussion and an introduction to the proposed solution are given in the third section of this chapter. In the last section of this chapter, a very short introduction to the other approaches is also given.

Most of the mathematical structures and their properties used in this chapter are fairly standard and simple. In any case, the mathematical background required in the entire thesis is given in the next two chapters.

1.1 Abstract Interpretation

There are numerous everyday examples of when one is interested in partial information about the result of a computation. To obtain partial information it may not be necessary to perform the whole computation. For example, to determine the sign of the product $(-694) * 453$, one could first do the multiplication and then inspect the sign of the result. But since it is the product of a negative and a positive number, the result will always be negative, so the precise computation is superfluous.

In practice, these kinds of things are done without bothering about any formal justification of the process. However, it is desirable to view it as a special case of a very general procedure of deriving certain properties of computations without doing the actual computations. A formal development of the process is therefore necessary.

Returning to the example of multiplication, which is also one of the examples given in [3, 12], an operation on the set of possible sign of numbers is defined. Integers have three possible signs—positive, negative and neutral (or sign of zero), and they are respectively denoted by (+), (-) and (0) as in [37]. The operation on {(+),(-),(0)} is given by the table below.

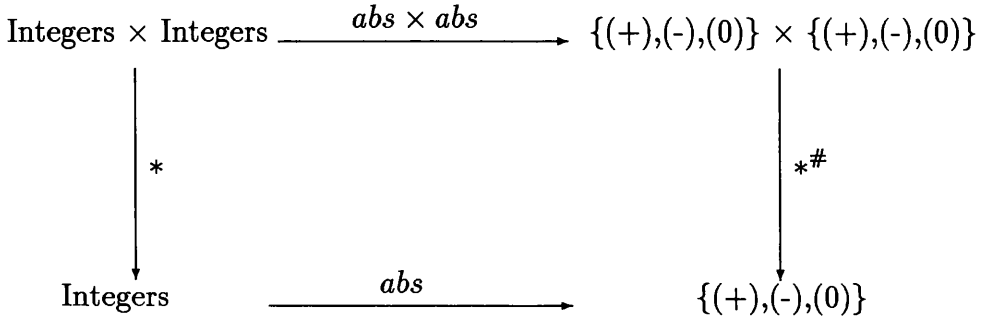
$*^{\#}$	(+)	(-)	(0)
(+)	(+)	(-)	(0)
(-)	(-)	(+)	(0)
(0)	(0)	(0)	(0)

This operation may be regarded as an "abstract multiplication". As far as signs are concerned $*^{\#}$ models $*$, where the latter denotes multiplication.

A formal description is then given by first defining an abstraction map

$$abs : Integers \rightarrow \{(+), (-), (0)\}$$

This map is the function that returns the sign of its argument. It is now easy to show that the diagram below commutes.



This means that for any integers x and y :

$$abs(x * y) \;=\; (abs\ x) *^{\#} (abs\ y)$$

Therefore, the sign of a product of numbers is obtained by applying $*^{\#}$ to the signs of the numbers.

If addition were to be considered instead, the set of signs would not be closed under the corresponding abstract function. For example, the sign of the sum of a negative and a positive number depends on the magnitudes of the numbers. Therefore, a new element $(\pm)$ is added to the set. In this context, $(\pm)$ denotes the property of being an integer (including 0).

In defining $+^{\#}$, (0) is its identity, and $x +^{\#} y = (\pm)$ whenever x is (-) and y is (+) or *vice-versa*, or when one of them is $(\pm)$. Completing the definition is fairly straightforward. Thus a table for $+^{\#}$, similar to that of $*^{\#}$, is obtained. However, the diagram corresponding to $+^{\#}$ does not commute because the sign of of the sum of two numbers is not entirely determined by the signs of the numbers.

To regain some aspects of a commuting diagram it is convenient to introduce an ordering on {(+),(-),(0),$(\pm)$}, where $x \sqsubseteq (\pm)$ for every x. An intuitive meaning of the statement $x \sqsubseteq y$ is that if a number has the sign x then it is safe to assume that it has sign y. It is now easy to see that the following statement holds.

$$abs(x + y) \;\sqsubseteq\; (abs\ x) +^{\#} (abs\ y)$$

Unlike the previous case, some of the information obtained from computing on the finite set is not exact. However, it is always safe, *i.e*, it does not give wrong information. Whenever it is impossible to obtain exact information, the ordering on the finite set makes it possible to obtain approximate values.

Our aim is to study languages in which it is possible to express a variety of operations. Now, depending on the property to be investigated, abstract versions of operations are defined. Interpreting these abstract operations to determine certain properties of programs is called *abstract interpretation.* An essential feature of many program analyses by abstract interpretation is that the abstract versions (or abstract functions) are interpreted over finite domains and therefore there are no infinite loops during their evaluation.

1.1.1 First-Order Functions

Mycroft was the first to use abstract interpretation for strictness analysis of a language of first-order recursion equations on flat domains [37]. Flat domains are simple to deal with because a value in a flat domain denotes either a non-terminating computation or a total one.

If `f` is a function of one variable then its standard interpretation (or semantics) is given by some continuous function

$$f : D \to D$$

where D is some flat domain. In [37] `f` is also given an abstract (non-standard) interpretation which is a continuous function on the two-element lattice $\mathbf{2}(= \{0,1\}$ with $0 \sqsubseteq 1)$

$$f^{\#} : \mathbf{2} \to \mathbf{2}$$

The two interpretations are related by an important *safety condition*; whenever $f^{\#}0 = 0$ it is always the case that $f\bot = \bot$, *i.e*, f is strict. Therefore, it is sufficient to check for $f^{\#}0 = 0$ to conclude that the original function is strict. It is important, however, to note that there are examples of strict functions where $f^{\#}0 \neq 0$.

The non-standard interpretation of a term `e`, that is denoted by $\mathtt{e}^{\#}$, is defined by induction on the structure of `e` as follows :

(i) if `e` is an integer or a boolean constant then $\mathtt{e}^{\#} = 1$

(ii) if `e` is of the form `e1 op e2`, where `op` is one of the usual binary operators `+`, `-`, `*` and `/`, then $\mathtt{e}^{\#} = \mathtt{e}_1^{\#} \wedge \mathtt{e}_2^{\#}$

(*iii*) if `e` is `if b e2 e3` then $\mathtt{e}^{\#} = \mathtt{b}^{\#} \wedge (\mathtt{e2}^{\#} \vee \mathtt{e3}^{\#})$

The operations $\wedge$ and $\vee$ have the usual boolean algebra interpretations, *i.e*, *conjunction* and *disjunction* respectively, where 0 is treated as *False* and 1 is treated as *True*.

Consider the example

```
f x = x + 6
```

It can be shown that $f^{\#}x = x$. Therefore, $f^{\#}0 = 0$ and hence f is strict.

Suppose `f` is a recursive function defined by something of the form

```
f x = . . . f ( . . . ) . . .
```

To obtain the abstract interpretation of `f`, a sequence of monotonic functions $\{f_i^\#\}$ is defined where $f_0^\#$ is the constant function that always returns 0, and each $f_{i+1}^\#$ is defined to be the function obtained using $f_i^\#$ in the right hand side of the definition of the function. Since the functions are interpreted on a finite domain, there is an i beyond which no new functions are obtained. Then, $f_i^\#$ is taken to be the abstract interpretation of `f`. Obtaining this $f_i^\#$ requires an iteration with a test for equality at each step. It must be noted that checking the equality of functions is an expensive operation.

Normally, a function is represented by the set of all its argument-result pairs. To check the equality of functions, their corresponding representations are compared. To make this operation more efficient, Clack and Peyton Jones [12] introduce a more compact representation of monotonic functions. For example, if a monotonic function $g : \mathbf{2} \to \mathbf{2}$ maps 0 to 1, it clearly also maps 1 to 1. Hence, $\{(0,1)\}$, or simply $\{0\}$, may be taken as its representation. This is in some sense the "minimal" set of points mapped to 1 by g. The other points can be determined from the monotonicity of the function. On the other hand, it is possible to represent the function in terms of points mapped to 0. Such sets, which are called *frontiers*, were introduced and used in algorithms developed by Clack and Peyton Jones. The correspondence of frontiers with open sets, and also with closed sets, in the topology which arises from the lattice structure was explored by Hunt [24]. Using these new representations, Hunt and Hankin [25] present algorithms for computing fixed points of functionals; their method applies to higher-order functions as well.

If `f` is a function of n variables, Mycroft defines the abstract interpretation to be some function $f^\# : \mathbf{2}^n \to \mathbf{2}$. He showed that if $f^\#(1,...,1,0,1,...,1) = 0$, where the 0 in the n-tuple is in the k-th position, then the function is strict in its k-th argument.

Consider the recursive function `f` given by

```
f(x,y,z) = if x = 0 then y-1 else f((x-1),z,y)
```

Now, it is easy to show that we need to interpret the function $\mathtt{f}^\# : \mathbf{2}^3 \to \mathbf{2}$, where

$\mathtt{f}^{\#}(\mathtt{x},\mathtt{y},\mathtt{z}) = \mathtt{x} \wedge (\mathtt{y} \vee \mathtt{f}^{\#}(\mathtt{x},\mathtt{z},\mathtt{y}))$. The necessary iterates are tabulated below.

	$f_0^{\#}$	$f_1^{\#}$	$f_2^{\#}$	$f_3^{\#}$
$(0,0,0)$	0	0	0	0
$(0,0,1)$	0	0	0	0
$(0,1,0)$	0	0	0	0
$(0,1,1)$	0	0	0	0
$(1,0,0)$	0	0	0	0
$(1,0,1)$	0	0	1	1
$(1,1,0)$	0	1	1	1
$(1,1,1)$	0	1	1	1

Since $f_2^{\#} = f_3^{\#}$, $f_i^{\#} = f_2^{\#}$ for every $i \geq 2$. Therefore, $f_2^{\#}$ is taken to be $f^{\#}$. Since $f^{\#}(0,1,1) = 0$ $\mathtt{f}$ is strict in its first argument. Both $f^{\#}(1,0,1)$ and $f^{\#}(1,1,0)$ are different from 0, hence it is not known whether or not the function is strict in its second or third argument.

Normally, a program has a set of function definitions $\{\mathtt{f}_j\}$. Strictness analysis is done by starting with a family $\{f_{j,0}^{\#}\}$, where for each j, $f_{j,0}^{\#}$ is the constant function that returns 0. The collection $\{f_{j,i+1}^{\#}\}$ is then defined in terms of $\{f_{j,i}^{\#}\}$ in a way similar to the example above. For each iteration, tests for equality have to be done. If the program contains n function definitions, performing n tests, that is checking if $f_{j,i+1}^{\#} = f_{j,i}^{\#}$ for each j, is expensive. But if one test shows the inequality of a pair, then one more iteration is performed. A more efficient way of doing this analysis would be to divide the call graph into strongly-connected components. A worrying problem is that, even in the case of a single function this process could be expensive.

There is an obvious parallel between this treatment and the rule of signs. In the case of addition and the rule of signs $(+)$ denotes all positive integers, and $(\pm)$ denotes all integers. Here, 0 stands for non-terminating terms and 1 denotes all terms. However, the abstraction map from D to $\mathbf{2}$ is not yet explicitly defined. In the next section, the definition and properties of the map in the context of a language with higher-order functions is provided.

1.1.2 Higher-Order Functions

A language with higher-order functions that has been studied extensively is the simply typed λ-calculus. In the case of the pure simply typed λ-calculus, it would not be necessary to perform strictness analysis. It may be reasonable to talk about strictness of a function in the sense that a function is strict if it needs to evaluate its argument. But no non-terminating computation can be expressed in this language—it is strongly normalisable. However, when the language is extended to allow recursion and some constants are introduced, this property no longer holds.

Mycroft's work was extended to such a language by Burn et al [11]. Below, a summary of their work is presented; from now on, we shall refer to this as BHA-style abstract interpretation. A detailed account of abstract interpretation of such languages may be found in a book by Burn [10].

Type expressions are given by the syntax

$$\sigma ::= A \mid \sigma \to \sigma$$

where A is a base type.

Terms are given by the grammar :

$$\begin{aligned} \mathtt{e} ::= \quad & \mathtt{c}^{\alpha} \\ \mid \; & \mathtt{x}^{\alpha} \\ \mid \; & \lambda \mathtt{x}^{\alpha}.\mathtt{e} \\ \mid \; & \mathtt{e}_1 \; \mathtt{e}_2 \end{aligned}$$

An *interpretation* of a language is a set-up where there is a family of domains $\{D_\sigma\}$, indexed by the type expressions, and an assignment of values c in D_σ for constants $\mathtt{c}^\sigma$. This is normally done by starting with a domain D_A corresponding to the base type A, and then $D_{\sigma\to\tau}$ is defined by induction, to be the set of all continuous functions from D_σ to D_τ. This is sometimes written as $D_\sigma \to D_\tau$. Interpretations of this kind are called domain-theoretic interpretations. Domains are preferred to sets so that the fixed-point combinator $\mathtt{Y}^\alpha$ is given a meaning.

As described in [4], any interpretation induces a semantic function. To see how

this function is defined, first D is let to be the union of the family of domains in the interpretation. Suppose $\mathbf{K} : \mathit{constants} \rightarrow D$ is the function (given in the interpretation) that assigns values to the constants of the language. Then, the semantic function

$$\mathit{sem} : \mathit{Exp} \rightarrow \mathit{Env} \rightarrow D$$

where, Exp is the set of expressions, and Env is the set of all partial functions from the set of variables to values, is defined as

$$\begin{aligned}
\mathit{sem}(\mathtt{c}^\alpha)\ \rho &= \mathbf{K}(\mathtt{c}^\alpha) \\
\mathit{sem}(\mathtt{x}^\alpha)\ \rho &= \rho(\mathtt{x}^\alpha) \\
\mathit{sem}(\lambda\mathtt{x}^\alpha.\mathtt{e})\ \rho &= \lambda y^\alpha.(\mathit{sem}(\mathtt{e})\ \rho[y^\alpha/\mathtt{x}^\alpha]) \\
\mathit{sem}(\mathtt{e}_1\mathtt{e}_2)\ \rho &= (\mathit{sem}(\mathtt{e}_1)\ \rho)(\mathit{sem}(\mathtt{e}_2)\ \rho)
\end{aligned}$$

Depending on the applications in mind, several interpretations may be defined for a language. Here, however, strictness analysis is the application under consideration. Thus, only two interpretations are considered. The first interpretation is the one that provides the standard semantics of the language, and the second is an abstract interpretation. If $\mathbf{K}$ assigns the constants their standard interpretations, then the resulting semantic function gives the usual denotational semantics of the language. It is important to note that $\mathtt{if}^\alpha$ and $\mathtt{Y}^\alpha$ are two of the constants, and their standard interpretations are given by

$$\mathbf{K}(\mathtt{if}^\alpha)\ x\ \ y\ \ z = \begin{cases} \bot & \text{if } x = \bot \\ y & \text{if } x = \mathit{True} \\ z & \text{if } x = \mathit{False} \end{cases}$$

and

$$\mathbf{K}(\mathtt{Y}^\alpha) \ = \ \lambda f^{\alpha\rightarrow\alpha}.\bigsqcup f^n(\bot)$$

In the case of the abstract interpretation, the domains are denoted by B_σ, where B_A is the two-element domain **2**. Since B_A is a lattice, B_σ is also a lattice for every type σ. Since all these lattices are finite they are complete, and hence Tarski's theorem may be applied to them [49]; that is, monotonic functions have fixed points.

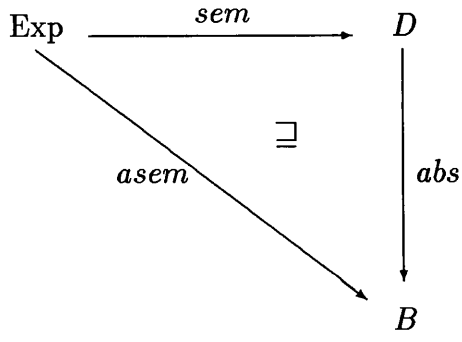
To complete the definition of the abstract interpretation, it is necessary to assign values to the constants in the language. For example, the conditional, $\mathtt{if}^A$, is assigned the value $if^\#$ defined by :

$$if^\# x\ y\ z = \begin{cases} 0 & \text{if } x = 0 \\ y \sqcup z & \text{if } x = 1 \end{cases}$$

The values corresponding to the usual arithmetic operators and constants are the same as in the previous section. Now, the abstract interpretation induces a semantic function in exactly the same way as the standard interpretation. To distinguish this semantics from the previous one, it is sometimes called a non-standard semantics or simply abstract interpretation. For any function `f` in a program, its standard and non-standard semantics are again denoted by f and $f^\#$ respectively.

Obviously, it is necessary to establish a connection between the two semantics. Now, if `f` is a function of type $\sigma_1 \to \ldots \to \sigma_n \to A$, then whenever $f^\# 1_{\sigma_1} \ldots 0_{\sigma_k} \ldots 1_{\sigma_n} = 0$, it is also the case that $f x_1 \ldots \perp_k \ldots x_n = \perp$ for all x_i in D_{σ_i}. Thus the strictness of the abstract interpretation in the k-th argument implies that of the standard interpretation. The points 0_{σ_i} and 1_{σ_i} stand for the least and greatest elements of the lattice B_{σ_i} respectively.

One of the most important contributions of the work reported in [11] is the mathematical machinery developed to show that the analysis is sound. To prove soundness, a relationship between the two interpretations is first established. This is is done by defining abstraction functions abs_σ, for each type σ, from D_σ to B_σ. Then these functions are used to relate *sem* and *asem*, where *asem* is the abstract interpretation. The following diagram may be helpful in trying to see what is going on.



The abstraction functions are defined by induction on the structure of types. In the case of the base type A, it has the same definition as the function $HALT$ that was given by Mycroft in [37]. That is,

$$abs_A(d) = \begin{cases} 0 & \text{if } d = \bot \\ 1 & \text{otherwise} \end{cases}$$

Abstracting Functions

One way of dealing with the higher-order case is to use Hoare powerdomains in the definition

$$abs_{\alpha\to\beta}(f)\ b = \bigsqcup(\mathbf{P}(abs_\beta)(\mathbf{P}(f)(Conc_\alpha(b^\downarrow)))),$$

where $\mathbf{P}$ is the functor that maps domains D to their corresponding Hoare powerdomains $\mathbf{P}(D)$, and any function f to $\mathbf{P}(f)$. The function $\mathbf{P}(f)$ takes any closed subset X of D and returns $f(X)^\downarrow$—the closure of the set of images of elements of X under f, and $b^\downarrow$ is the closure of $\{b\}$. The function $Conc_\alpha$ is also defined on sets and it is the inverse image abs_α^{-1}, *i.e*, it takes any set Y into $\{x \in D \mid abs_\alpha(x) \in Y\}$. Since abs_α is continuous, $Conc_\alpha(Y)$ is closed for every closed set Y. An equivalent definition, which does not explicitly use powerdomains, was given by Abramsky [4].

$$abs_{\alpha\to\beta}(f)\ b = \bigsqcup\{abs_\beta(f(d)) \mid abs_\alpha(d) \sqsubseteq b\}$$

In the abstract interpretation being described, elements of lattices represent information about values. Moreover, if a and b are in a lattice and $a \sqsubseteq b$ then a is more

informative than b. The motivation underlying the definition of $abs_{\alpha\to\beta}(f)$ above, is that if b is some information that is known about (or a property of) an argument of f, then to find some information about the result of f, first the results of f at all arguments that we know more than b about are considered. Finally, we approximate the information across all the results.

Two important properties of the abstraction functions are the fact that they are both strict and $\perp$-reflecting (only $\perp$ is mapped to $\perp$). Moreover, when f and d are of the appropriate types, the following semi-homomorphism property holds

$$abs_\tau(fd) \sqsubseteq abs_{\sigma\to\tau}(f)abs_\sigma(d).$$

A simplified version of a lemma in [11] is that if e is a closed term of type α then $abs_\alpha(sem(\mathsf{e}))\ \rho \sqsubseteq tabs(\mathsf{e})\ \rho'$. In particular, if the functions f and $f^\#$ denote the corresponding interpretations then

$$abs_{\sigma\to\tau}(f) \sqsubseteq f^\#$$

It is from these facts that the soundness theorem for strictness analysis follows. In the case of a function of one variable, for example, if $f^\#0 = 0$ then the proof of the strictness of f is given by :

$$\begin{aligned} abs_\tau(f\perp) &\sqsubseteq abs_{\sigma\to\tau}(f)(abs_\sigma(\perp)) \\ &= abs_{\sigma\to\tau}(f)(0) \\ &\sqsubseteq f^\#0 \\ &= 0 \end{aligned}$$

Therefore,

$$abs_\tau(f\perp) = 0.$$

Since abs_τ is $\perp$-reflecting, $f\perp = \perp$.

1.1.3 Data Structures

Although Mycroft's method can be used in the analysis of data structures, the result it gives is too weak to be of practical significance. If a function's argument

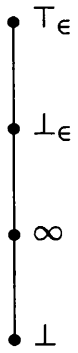
is a list, then there are several levels in the way the function may use its argument. To illustrate this, we look at some examples given by Wadler [52]. Consider the functions `isempty`, `length` and `sum` defined by

```
isempty    nil       = True
isempty (cons x xs) = False

length    nil       = 0
length (cons x xs) = 1 + length xs

sum       nil     = 0
sum (cons x xs) = x + sum xs
```

It is not difficult to observe that the extent to which these functions have to evaluate their arguments varies. Wadler introduced a set whose elements model these differences [52]. He used the abstract domain $\{\bot, \infty, \bot_\epsilon, \top_\epsilon\}$. The ordering is as given in the diagram below.



Notice that the notation uses $\bot$ and $\top$ instead of 0 and 1 of the previous sections.

Wadler first provided a table representing $cons^\#$, *i.e*, the function from $\mathbf{2} \times \{\bot, \infty, \bot_\epsilon, \top_\epsilon\}$ to $\{\bot, \infty, \bot_\epsilon, \top_\epsilon\}$ that models the behaviour of `cons`. In addition, he also defined $nil^\# = \top_\epsilon$. Now, if `h` is the function defined by

```
h     nil      =   a
h (cons x xs) =   f x xs
```

then $h^{\#}$ (Wadler's abstract interpretation of h) is given by

$$\begin{aligned} h^{\#}\top_{\epsilon} &= a^{\#} \sqcup (f^{\#}\top\ \top_{\epsilon}) \\ h^{\#}\bot_{\epsilon} &= (f^{\#}\bot\ \top_{\epsilon}) \sqcup (f^{\#}\top\ \bot_{\epsilon}) \\ h^{\#}\infty &= f^{\#}\top\ \infty \\ h^{\#}\bot &= \bot \end{aligned}$$

The results of the abstract interpretation are to be understood as follows :

(i) If $h^{\#}\bot = \bot$ then it is safe to evaluate its argument to expose the first `cons` or `nil`.

(ii) If $h^{\#}\infty = \bot$ then it is safe to evaluate its argument and all the tails recursively.

(iii) If $h^{\#}\bot_{\epsilon} = \bot$ then it safe to evaluate its argument and recursively all the tails and heads.

Returning to the examples at the beginning of the section, it is not difficult to see that `isempty`, `length` and `sum` have properties (i), (ii) and (iii) respectively.

What we have seen so far works for lists of integers, and more generally for lists over any flat domain. Wadler also showed how to build the abstract domains appropriate for lists of arbitrary types, and also described the abstract interpretation of functions defined on lists of lists of integers as examples. We will return to this in more detail in Chapter 7.

1.2 Polymorphism

Most functional languages have polymorphic type systems of some kind. Monomorphic instances of any polymorphic function can be analysed by the methods described earlier. However, different instances of a polymorphic function have similar strictness properties. For instance, Abramsky has shown that abstract interpretation-based strictness analysis of a language with the Hindley-Milner type

system is *polymorphically invariant* [1]. This means that the analysis can detect the strictness of one instance of a polymorphic function if and only if it can detect the strictness of all instances. Therefore, it is claimed that it is sufficient to deal with the smallest instance of a polymorphic function. It must be said that this applies to a specific technique. The moment one decides to use a different analysis technique it is necessary to check if polymorphic invariance still holds.

If different instances of a polymorphic function are called at many points in a program, polymorphic invariance may not be very useful. To obtain as much information as possible, it is necessary to compute the abstract functions of all these instances. To see this consider the following example. Let

```
f x y = if x = 0 then y else f (x-1) y
```

The abstract interpretation of the smallest instance of `f` is a function in $\mathbf{2} \to \mathbf{2} \to \mathbf{2}$, and it may be given by a table. Now, consider the function g given by

```
g x = fst (f 0 x)
```

where `fst` is the function which takes a pair and returns the first argument. To compute the abstract interpretation of g, we need that of some instance of `f`. This instance of `f`, however, is not the smallest one. Therefore, the table mentioned earlier will not be of any use here. Thus, we need to build another table for the appropriate instance. Clearly, computing the abstract functions of all instances used in a program is inefficient. What is desirable is to use the information obtained from one instance to study properties of other instances. In order to do this, it is helpful to establish a relationship between the semantics of all instances of a polymorphic function.

1.2.1 Semantics

A very simple way of looking at the semantics of a polymorphic function is to regard it as a mere collection of continuous functions where each continuous function corresponds to a monomorphic instance. This view does not express any relationship

between the instances. But the function bodies of all the instances of a polymorphic function are essentially the same. The similarity at the syntactic level should certainly imply some semantic relationship between the different instances. Before dealing with polymorphic functions, it is useful to consider some semantic properties of terms in the simply typed λ-calculus.

λ-definability

Plotkin establishes some semantic properties that must hold about terms definable in the simply typed λ-calculus [40]. To see what these properties are, first assume that there is only one base type A. The full type hierarchy (or full set-theoretic interpretation), which is a collection $\{X_\sigma\}$ of sets is then defined by induction on types. This is done by starting with some non-empty set X that is taken to be X_A. For any higher type $\alpha \to \beta$, $X_{\alpha\to\beta}$ is defined to be the set of all functions from X_α to X_β. Now, any closed term of the typed λ-calculus may be interpreted by assigning it a value (in a way similar to what *sem* does in the previous section) from the set of the appropriate type. Plotkin reports that such values satisfy certain uniformity conditions (see also Coquand [14]). To express this, consider any permutation (bijection) $\pi_A : X_A \to X_A$. Then for each type $\alpha \to \beta$, $\pi_{\alpha\to\beta} : X_{\alpha\to\beta} \to X_{\alpha\to\beta}$ is defined by induction as

$$\pi_{\alpha\to\beta}(f) = \pi_\beta \circ f \circ \pi_\alpha^{-1}.$$

Here, π_α^{-1} is the inverse of the permutation π_α which obviously is itself a permutation. It is also easy to see that $\pi_{\alpha\to\beta}$ is a permutation.

The invariance condition that definable terms satisfy is that, if f denotes some closed term of type σ then $\pi_\sigma(f) = f$. It is important to emphasise that so far this property is only stated and holds for the pure system. If the system is extended to include constants of type A, then the invariance holds provided some restrictions are made to the definition of π_A—it is let to be an identity on the constants.

The invariance described here is only a necessary condition; there are objects not definable in the language which have this property. The equality operation $=$ is an example, which was given by Plotkin, that is invariant under the appropriate permutations, but is not definable in the pure system.

Logical relations

In an attempt to give a better characterisation of the definable objects Plotkin introduced a semantics based on relations. To do this a collection of relations $\{R_\sigma\}$, known as *logical relations*, is defined. This is done by starting with any relation R_A corresponding to the base type A, and then defining $R_{\alpha\to\beta}$ by

$$(f,g) \in R_{\alpha\to\beta} \text{ if and only if for all } (x,y) \in R_\alpha : (f(x),g(y)) \in R_\beta$$

Now, the invariance theorem stated in terms of relations is that if f denotes some closed term of type σ then $(f,f) \in R_\sigma$. This is true for all logical relations. This theorem is stronger than the invariance theorem involving permutations; in fact it is possible to show that the permutations generate logical relations, but not all logical relations are generated by permutations. These invariance results are about the simply typed λ-calculus.

The relational approach was also used by Reynolds to study terms written in an extended typed λ-calculus [43] . The extension was made by allowing expressions of the form `lettype` $\tau = \sigma$ `in` `e`, where τ is a type variable and σ is a type expression. Reynolds's *abstraction theorem* establishes a relationship between the values of `lettype`-expressions with different σ's. Such expressions may be regarded as instances of some polymorphic term.

The category of domains and embedding-projection pairs was used to present a model of the polymorphic (second-order) λ-calculus by Coquand et al [15]. The language does not allow recursive definitions. In this thesis we take a similar approach except that, for reasons to be explained in future chapters, in the strictness analysis of polymorphic functions it is convenient to use the category of domains and embedding-closure pairs instead (see also Baraki [6]).

Wadler also used the relational approach to derive theorems about polymorphic functions definable in the polymorphic λ-calculus [53]. Types are interpreted as relations and closed terms satisfy conditions similar to those involving logical relations in [40] in the simply typed λ-calculus. Concentrating on relations that arise out of functions, equations involving different instances of polymorphic functions are obtained. These equations are the theorems. Wadler also shows that the language

can be extended by adding fixpoints, but then similar theorems only hold if certain restrictions are imposed on the kind of relations used.

1.2.2 First-Order Functions

Hughes investigated an abstract interpretation of polymorphic first-order functions [21]. He showed how monotypes could be interpreted as domains and type constructors as functors in the category of domains and strict functions. In this framework, it was shown that polymorphic functions are natural transformations. The components of the natural transformation, corresponding to a polymorphic function, are the semantics of monomorphic instances of the polymorphic function. Thus any two instances are related in a certain manner. Such relationships are used in establishing relationships between abstract interpretations of different instances of a polymorphic function.

For example, let $F(t)$ and $G(t)$ be parameterised types and f be the natural transformation from F to G which is the semantics of some polymorphic function `f`. Then for any domains A and B, if $\alpha : A \to B$ is a strict function then

$$G(\alpha) \circ f_A \;=\; f_B \circ F(\alpha)$$

Moreover, Hughes also showed that for any type τ, there is a term $E(f)$ such that

$$f_{Abs\,\tau} \;=\; E(f_{\mathbf{2}})$$

Assuming that **int** and **bool** as the only basic types, $Abs\ \tau$ is the type obtained from τ by substituting the occurrence of these basic types by **2**. It was then shown that

$$f^{\#}_{\tau} \;\sqsubseteq\; E'(f^{\#}_{\mathbf{2}})$$

where E' is obtained from E by replacing the terms by their abstract values. Thus, once $f^{\#}_{\mathbf{2}}$ is computed, $E'(f^{\#}_{\mathbf{2}})$ provides an approximation to $f^{\#}_{\tau}$.

The difficulty with the higher-order case comes from the contravariance of the function type constructor in its first argument.

1.2.3 Semantic Polymorphic Invariance

A result more powerful than the polymorphic invariance of strictness analysis was recently proved by Abramsky and Jensen [5]. They showed the semantic polymorphic invariance of strictness for a higher-order language. That is, an instance of a polymorphic function is strict if and only if all instances are. To do this, they use the notion of a *relator*. A relator is a mapping that is defined in a similar way as a functor but is not required to preserve composition. In this application, *transformations* are collections of functions which may also be viewed as relations satisfying a condition weaker than naturality. Using relators and transformations, they gave a semantics to a polymorphic higher-order language, where types are modelled by relators and polymorphic functions by transformations.

The significance of semantic polymorphic invariance is that if any analysis technique detects that an instance is strict then all instances must be strict. This holds even for instances whose strictness may not be detected by that analysis.

1.3 The Main Problem and a Solution

As illustrated by an example in the previous section, when different instances of a polymorphic function are used in a program, the abstract interpretation of each must be computed. The necessity of doing so and the fact that results from polymorphic invariance are not satisfactory was first pointed out by Burn [8]. On the other hand, it is obvious that the computation is very expensive; this is especially the case with recursive functions. This is the problem addressed in this thesis.

We have already seen how Hughes [21], in the first-order case, obtains an approximation to the abstract interpretation of any instance of a polymorphic function from that of the smallest. In this thesis, we develop a method similar to that of Hughes. Our method, however, is not restricted to first-order-functions. It applies to higher-order functions, and also to functions defined on lists. In most cases, the method gives good approximations and, moreover, these approximations are significantly cheaper to compute. We also show how similar techniques could be used in

abstract interpretations for binding-time analysis and termination analysis.

1.4 Other Approaches to Strictness Analysis

Before we move on to the next chapter, we give a summary of other methods used in strictness analysis.

1.4.1 Projections

Methods of doing strictness analysis for data types, lazy lists for example, over non-flat domains were developed by Hughes [20] (see also [22]). His approach is entirely different from the abstract interpretation described in the previous sections. The idea is to look at an expression to be evaluated and find out how much of its sub-expressions are needed. For example if
`f x y = x + 1`, looking at the body of `f`, *i.e* `x + 1`, it is easy to observe that it is strict in `x` but not in `y`. In general, assuming that an expression is to be evaluated, the aim is to determine the sub-expressions that must necessarily be evaluated. In the example, if `x + 1` is to be evaluated then `x`, but not `y`, must be evaluated. Thus information propagates from expressions to their sub-expressions. We also note that the function never needs to evaluate its second argument; this kind of information may be useful. On the other hand, Mycroft's method would only detect that it is strict in `x`, and gives no information about `y`.

To capture, in terms of semantics, the propagation of information from an expression to its sub-expressions Wadler and Hughes [51] introduced the idea of a *projection.* Let D be a domain. A continuous function $\alpha : D \to D$ is called a projection on D, if

$$\begin{aligned} \alpha \circ \alpha &= \alpha \text{ and} \\ \alpha &\sqsubseteq id, \end{aligned}$$

where id is the identity function on D. For example, if D is the domain of lists of

integers then the following two functions are projections on D.

$$H\ xs = \begin{cases} \bot & \text{if } xs = \bot : xs' \\ x : (H\ xs') & \text{if } xs = x : xs' \text{ and } x \neq \bot \\ xs & \text{otherwise} \end{cases}$$

$$T\ xs = \begin{cases} \bot & \text{if } xs \text{ is (or an approximation to) an infinite list} \\ xs & \text{otherwise} \end{cases}$$

Projections are used in describing the extent to which expressions will be evaluated. As can be seen from the previous example, we normally start from an expression and a context in which it is going to be evaluated and then find out about the context in which the sub-expressions will be evaluated. Semantically, this may be expressed by an equation of the form $\beta \circ f = \beta \circ f \circ \alpha$, where $f : D \to E$ is a continuous function and α and β are projections on D and E respectively. The equation is normally read as : f is α-strict in a β-strict context. Such an equation expresses the fact that whenever the body is to be evaluated in a β-strict context, then it is safe to evaluate the argument in an α-strict context.

We consider an example that is given in [51]

```
before      []       =  []
before  (cons x xs)  =  [],  if x = 0
                     =  cons x : (before xs)
```

Now, it is easy to see that

$$id \circ before = id \circ before \circ H$$

From this we say that *before* is *head-strict.* Kamin [31] has shown that such a property cannot be detected by abstract interpretation over finite domains representing Scott-closed sets. A comparison of the analyses, which use abstract interpretation and projections, is made by Burn [9]. He also introduces a notion of head-strictness which can be detected by abstract interpretation. Hunt [27] has developed an abstract interpretation where properties are partial equivalence relations, rather than Scott-closed sets; this is used to detect head-strictness.

The notion of polymorphic projections was introduced by Hughes and Launchbury [23]. Polymorphic projections are natural transformations in the category of domains

where the morphisms are strict and $\perp$-reflecting continuous functions. It is shown that polymorphic first-order functions are natural transformations in the category of domains and strict functions. They also prove a more general kind of an invariance theorem; that is, if f_τ is α_τ-strict in a β_τ-strict context for some type τ, then it is true for all types.

1.4.2 Strictness Analysis by Type Inference

Both abstract interpretation and projection analysis use denotational semantics in defining strictness. Kuo and Mishra introduced a type system for a language to express the strictness properties of programs [32]. Their approach is operational. First, they define an evaluation method for the language and then terms that do not have head normal forms are classified as divergent. Now, a term `F` is said to be strict if for every divergent term `E`, `F E` is divergent.

The type system used to express the strictness properties of programs has two constants (or basic types), ϕ and $\Box$. In general (strictness) types are defined by the following syntax.

$$\alpha \ ::= \ \phi \mid \Box \mid \alpha \rightarrow \alpha$$

Semantically, ϕ stands for the set of all divergent terms, and $\Box$ denotes the set of all terms, $\alpha \rightarrow \beta$ stands for all terms `F` such that `F E` is in β whenever `E` is in α. For example integer constants have the type $\Box$, and $\phi \rightarrow \phi$ is a type of strict terms. Primitive operators, for example `+`, may have more than one strictness type. Both $\phi \rightarrow \Box \rightarrow \phi$ and $\Box \rightarrow \phi \rightarrow \phi$ are types of `+`. Moreover, there are terms which may have infinitely many types. For example λ`x.x` has types of the form $\alpha \rightarrow \alpha$ for all types α. To manage the situation of more than one type, type variables are introduced. Unfortunately functions need not have a *principal type.* For example `+` has several types and no sensible ordering can be defined on them. A single representation of the strictness type of such operators is only obtained by modifying the definition of types. The new types are the old types together with a set of constraints. The set of constraints is similar to the ones one finds in subtyping [16].

Jensen [28] has investigated the relationship between analyses by type-inference and by abstract interpretation. Whereas Kuo and Mishra concentrated on the algorithmic aspects of the type inference, Jensen's contribution was on the logical aspects via Stone duality. He enriched the type system by introducing operators, such as conjunction, and he proved that, as far as strictness is concerned, both methods are equally powerful. Furthermore, in [29] he considered a system of disjunctive types to develop a logic for strictness analysis. From this he obtained a disjunctive abstract interpretation. Both works are about the simply-typed λ-calculus. A more general and detailed account of the relationship between abstract interpretation and logic-based methods for program analysis is given in his thesis [30]. Benton [7] has also investigated strictness analysis by type-inference, and has proved that the analysis is polymorphically invariant. He also studied polymorphic invariance of higher-order properties. An example is the property of mapping strict functions to strict functions. It may not make sense to ask if every instance of a polymorphic function has this property. For instance, all instances of the identity function where it makes sense to ask such a question have the property. But, there are instances, and the simplest instance of the identity function is an example, where this doesn't apply.

In abstract interpretation, expensive computations have to be performed. To find fixed points of functions, iterations and tests for equality of functions at each iterative step have to be done. It is, therefore, claimed in [32] that the strictness analysis by type-inference is more efficient. This is partly because it is hoped that much that has been done in developing efficient type checkers will also apply to this analysis. However, carrying the sets of constraints around and checking if a type satisfies several constraints is likely to introduce inefficiency. We have not yet seen any reported results of experiments comparing the efficiency of abstract interpretation and type-inference.

Chapter 2

Some Mathematical Structures

In this chapter we introduce definitions and properties of some mathematical structures that are used in the thesis. All these structures are fairly standard and may be found, for example, in Plotkin [41].

2.1 Partial Orders

A binary relation $\sqsubseteq$ on a set D is called a *partial order*, if the relation is :

- *reflexive*, *i.e*, $\forall x \in D : x \sqsubseteq x$
- *transitive*, *i.e*, $\forall x, y, z \in D$: if $x \sqsubseteq y$ and $y \sqsubseteq z$ then $x \sqsubseteq z$
- *antisymmetric*, *i.e*, $\forall x, y \in D$: if $x \sqsubseteq y$ and $y \sqsubseteq x$ then $x = y$

We call $(D, \sqsubseteq)$ (or just D) a *partially ordered* set.

Let X be a subset of a partially ordered set D and $a \in D$, a is an *upper bound* of X if $x \sqsubseteq a$ for all $x \in X$. If on the other hand, $a \sqsubseteq x$ for all $x \in X$ then a is called a *lower bound* of X. An upper bound a of X is called a *least upper bound* (or *supremum* or simply *sup*) of X if $a \sqsubseteq b$ for all upper bounds b of X. Similarly, a lower bound a of X is a *greatest lower bound* (or *infimum* or simply *inf*) of X if

$b \sqsubseteq a$ for all lower bounds b of X. Least upper bounds and greatest lower bounds are unique when they exist. This is a consequence of antisymmetry in the definition of partial order. The least upper bound and greatest lower bound of a set X are denoted by $\bigsqcup X$ and $\bigsqcap X$ respectively.

A non-empty subset X of a partially ordered set D is called a *directed* set if for every $a, b \in X$ there is a $c \in X$ such that $a \sqsubseteq c$ and $b \sqsubseteq c$. That is, every two-element subset of X has an upper bound in X. Equivalently, a set X is directed if every finite subset has an upper bound in X. A partially ordered set is said to be *complete* (or a *cpo*) if it has a least element (which is unique and usually denoted by $\perp$) and every directed set has a least upper bound. The least upper bound of a directed set is not necessarily an element of the set.

Let D and E be cpos and f be a function from D to E. f is said to be *continuous* if for every sequence $\{x_n\}$ in D with $x_1 \sqsubseteq x_2 \sqsubseteq x_3 \sqsubseteq \ldots$ (such a sequence is called a *chain*) we have

$$f(\bigsqcup x_i) \;=\; \bigsqcup f(x_i)$$

It is simple to observe that continuity implies monotonicity. That is, whenever $x \sqsubseteq y$ we have $f(x) \sqsubseteq f(y)$.

Normally, continuity is defined in terms of directed sets rather than chains. However, in the structures which we use in this thesis, the two definitions are equivalent.

A complete partial order where all sets with upper bounds have least upper bounds is said to be *consistently complete.* An element a of a cpo D is called *finite* if for any directed set X : $a \sqsubseteq \bigsqcup X$ implies that there is an $x \in X$ such that $a \sqsubseteq x$. A cpo where for every $x, \{e \mid e \sqsubseteq x$ and e is finite $\}$ is directed and $x = \bigsqcup\{e \mid e \sqsubseteq x$ and e is finite $\}$ is called *algebraic.* Algebraic cpos where the set of finite elements is countable are called *ω-algebraic.* A *Scott domain* (which we will simply call *domain* from now on) is a consistently complete ω-algebraic cpo.

2.2 Constructions

There are many ways of building new cpos out of a given class of cpos. In this section two examples of constructions are given.

(i) If $\{D_i \mid i \in I\}$ is a family of cpos, then the *product*

$$\prod_{i \in I} D_i$$

is defined to be the set of all objects $\{a_i\}$ where for each i, $a_i \in D_i$. Thus, these objects may be viewed as function. That is the product may also be defined as

$$\{f : I \to \bigcup_{i \in I} D_i \mid \forall i \in I : f(i) \in D_i\}$$

An ordering can be defined on the product by letting $a \sqsubseteq b$ if and only if for every i we have $a_i \sqsubseteq b_i$ in D_i. It can be shown that this ordering turns the product into a partially ordered set that is complete. If I is a finite set, say $\{1, 2, ..., n\}$, then the product is usually denoted by $D_1 \times D_2 \times ... \times D_n$.

(ii) If D and E are cpos then the set of all continuous functions from D to E is denoted by $D \to E$. The ordering, where $f \sqsubseteq g$ if and only if $f(a) \sqsubseteq g(a)$ for all $a \in D$, is a complete partial order. This set is called the *function space.*

2.3 Scott Topology and Powerdomains

Let D be a non-empty set. A collection of subsets, Ω, of D that contains both D and $\emptyset$, and that is closed under arbitrary unions and finite intersections is called a *topology* on D. We also say that (D, Ω) is a *topological space.* Elements of the topology are called *open* sets. Complements of open sets are called *closed* sets. A collection $\mathcal{B}$ of open subsets of D is called a *base* for the topology Ω if every element of Ω is a union of some elements of $\mathcal{B}$. Clearly Ω itself is a base, but normally the bases of interest are the ones that are minimal in some sense.

Although continuity of functions between cpos was already defined, continuity is a topological concept. If (D_1, Ω_1) and (D_2, Ω_2) are two topological spaces and $f : D_1 \to D_2$ is a function, then f is said to be continuous if for every open set $\mathcal{O}$ in D_2, $f^{-1}(\mathcal{O})$ is open in D_1. That is, the inverse image of an open set is open.

We will consider topologies that arise out of domains. Let D be a domain and Ω be the collection of subsets $\mathcal{O}$ of D that are upwards closed, *i.e*, for all $x \in \mathcal{O}$ and $y \in D$, if $x \sqsubseteq y$ then $y \in \mathcal{O}$, and moreover when S is a directed subset of D with $\bigsqcup S$ in $\mathcal{O}$ we have $S \bigcap \mathcal{O} \neq \emptyset$. It is not difficult to show that Ω is indeed a topology on D. It is called the *Scott topology* on D. Moreover, functions that are continuous between domains are also continuous in the topological sense and *vice-versa.*

Let D be a domain. For every $a \in D$, define $a^{\uparrow}$ and $a^{\downarrow}$ by :

$$\begin{aligned} a^{\uparrow} &= \{x \in D \mid a \sqsubseteq x\} \\ a^{\downarrow} &= \{x \in D \mid x \sqsubseteq a\} \end{aligned}$$

$a^{\uparrow}$ and $a^{\downarrow}$ are respectively called the *upward closure* and *downward closure* of $\{a\}$ (sometimes the notation $\{a\}^{\uparrow}$ and $\{a\}^{\downarrow}$ are respectively used instead of the above two). If $\{a_1, a_2, ...\}$ is the set of finite elements of D then each $a_i^{\uparrow}$ is open and moreover every open set $\mathcal{O}$ is a union of such open sets. Hence, these special open sets form a base for the Scott topology. They are referred to as *basic* open sets. An element a of D is finite if and only if $a^{\uparrow}$ is open; thus obtaining a characterisation of finiteness. However, there is no way of describing all closed sets in terms of the $a_i^{\downarrow}$'s, though $a^{\downarrow}$ is itself closed for any a.

Given some ordering, which is not necessarily complete, on a set D, there are several ways of defining orderings on the powerset (the set of all subsets) of D. Then by introducing some identifications, domains that are subsets of the powerset are obtained. Such domains are called *powerdomains.* For our purposes we simply start with a domain D and consider some subsets of the powerset that form domains, without trying to define general constructions.

The *Smyth powerdomain* of a domain D, with $\{a_1, a_2, ...\}$ as its set of finite elements, is the domain whose finite elements are finite unions of basic open sets of D. The ordering is the superset ordering. Thus a typical element of the powerdomain is

a (set-theoretic) intersection of a decreasing chain of such elements. On the other hand, the *Hoare powerdomain* is a domain whose finite elements are finite unions of downward closure of finite elements of D. The ordering is the subset ordering. Any element of this powerdomain is a (set-theoretic) union of an increasing chain of such sets.

2.4 Complete Lattices and Fixed Points

A *lattice* is a partially ordered set where every finite subset has both a least upper bound and a greatest lower bound. It is said to be *complete* if every subset has both a least upper bound and a greatest lower bound. Thus a complete lattice A has both a least element ($\bot = \bigsqcap A$) and a largest element ($\top = \bigsqcup A$). Clearly a complete lattice is a cpo. An element a of a lattice is said to be *meet-irreducible*, if $a = b \sqcap c$ implies that $a = b$ or $a = c$. It is called *join-irreducible* if $a = b \sqcup c$ implies that $a = b$ or $a = c$.

Tarski [49] proved that any monotonic function f from a complete lattice A into itself has a *fixed point*, *i.e*, there is an $a \in A$ such that $f(a) = a$. Moreover, the set of all fixed points is again a complete lattice.

The structures commonly used in the semantics of programming languages are not complete lattices but some kind of partial orders, and continuous functions on them. Continuous functions on cpos also have fixed points. Normally we are interested in least fixed points, and it is easy to show that for any continuous function f from a cpo to itself

$$\bigsqcup_{n \in \omega} f^n(\bot)$$

is the least fixed point of f. Here ω is the set of natural numbers $\{0, 1, 2, ...\}$, f^0 is the identity function, and for each positive integer n, $f^n = f \circ f^{n-1}$.

2.5 Categories

Definition 2.5.1 *A* category $\mathcal{K}$ *is a structure defined by the following set of data and properties.*

(i) *We have a collection* $Obj(\mathcal{K})$ *of objects.*

(ii) *For every pair of objects* A *and* B *in* $Obj(\mathcal{K})$*, there is a set* $Hom_{\mathcal{K}}(A, B)$*, called the set of morphisms from* A *to* B*.*

(iii) *Given* $f \in Hom_{\mathcal{K}}(A, B)$ *and* $g \in Hom_{\mathcal{K}}(B, C)$*, we may form the composite* $g \circ f \in Hom_{\mathcal{K}}(A, C)$*.*

(iv) *For every object* A *there is a morphism* $id_A \in Hom_{\mathcal{K}}(A, A)$*, called the identity morphism.*

(v) *The composition defined in (iii) is associative.*

(vi) *For every* $f \in Hom_{\mathcal{K}}(A, B)$*, we have* $f \circ id_A = f$ *and* $id_B \circ f = f$

Examples

(i) The collection of sets forms a category, where sets are objects, functions are morphisms, and ordinary function composition is the composition and the identity function is the identity morphism.

(ii) The collection of partially ordered sets forms a category, where the objects are partially ordered sets and the morphisms are continuous functions.

(iii) The collection of domains forms a category where domains are objects, continuous functions are morphisms, ordinary function composition becomes the composition of morphisms and the identity function becomes the identity morphism.

Given a category $\mathcal{K}$, its *opposite category* $\mathcal{K}^{op}$ is defined to be the category with the same objects as $\mathcal{K}$, and for every pair of objects A and B, $f \in Hom_{\mathcal{K}^{op}}(A, B)$ if

and only if $f \in Hom_{\mathcal{K}}(B, A)$. Now it is easy to see how the composition should be defined, and then verifying that $\mathcal{K}^{op}$ is indeed a category is also easy.

Let $\mathcal{C}$ and $\mathcal{D}$ be two categories. Then the *product category*, $\mathcal{C} \times \mathcal{D}$, is the category whose objects are pairs of the form (A, B) where A and B are objects of $\mathcal{C}$ and $\mathcal{D}$ respectively. A morphism between (A, B) and (C, D) will be of the form (f, g) where f and g are morphisms from A to C and from B to D respectively. Composition of morphisms is done component-wise. If for any positive integer n we have categories $\mathcal{C}_1, ..., \mathcal{C}_n$, then we can define their product in a similar way.

Definition 2.5.2 *A functor F from a category $\mathcal{C}$ to a category $\mathcal{D}$ is a mapping that maps objects of $\mathcal{C}$ to objects of $\mathcal{D}$, and for every pair of objects A and B in the category $\mathcal{C}$, F maps every morphism $f \in Hom_{\mathcal{C}}(A, B)$ to $F(f) \in Hom_{\mathcal{D}}(F(A), F(B))$ and satisfies the following conditions*

$$\begin{aligned} F(id_A) &= id_{F(A)} \quad \textit{for every object } A \in Obj(\mathcal{C}), \textit{ and} \\ F(g \circ f) &= F(g) \circ F(f) \quad \textit{whenever } g \circ f \textit{ is defined in } \mathcal{C} \end{aligned}$$

Such functors are called *covariant* functors. If instead of the second condition above, F satisfies the condition : $F(g \circ f) = F(f) \circ F(g)$, then F is called a *contravariant* functor.

Definition 2.5.3 *Let F and G be two functors from the category $\mathcal{C}$ to the category $\mathcal{D}$. Let $\{f_A\}$ be a collection of morphisms indexed by the objects of the category $\mathcal{C}$, with each $f_A \in Hom_{\mathcal{D}}(F(A), G(A))$. Such a collection is called a natural transformation from F to G if for every pair of objects A and B in the category $\mathcal{C}$, and any morphism $h \in Hom_{\mathcal{C}}(A, B)$ we have*

$$f_B \circ F(h) = G(h) \circ f_A$$

Chapter 3

Lax Natural Transformations

In this chapter the main theoretical results which will be used in the remainder of the thesis are introduced. To do this, we define a category which we shall call the category of domains and embedding-closure pairs. We concentrate on a certain class of collections of functions satisfying a condition weaker than naturality. These will be used in the following chapters in investigating some semantic properties of polymorphic functions. Moreover, we study the category of finite lattices and embedding-closure pairs in detail. This category will be very useful in Chapters 6, 7, and 8, where we establish a relationship between the abstract interpretation of different instances of any polymorphic function. The category of domains and embedding-projection pairs is also studied here; it is actually a category that has been used elsewhere, for example by Coquand et al [15].

From this point on, $\mathcal{C}$ will denote the category of domains and continuous functions.

3.1 Embedding-Closure Pairs

Consider the category $\mathcal{C}^{ec}$ whose objects are the same as that of $\mathcal{C}$, and with morphisms of the form $(e, c) : A \rightarrow^{ec} B$, where $e : A \rightarrow B$ and $c : B \rightarrow A$ are morphisms in $\mathcal{C}$ satisfying the conditions

$$c \circ e = id \quad \text{and} \quad e \circ c \sqsupseteq id$$

We shall call any such pair of continuous functions an *embedding-closure* pair. The set of embedding-closure pairs from A to B, *i.e*, $Hom_{\mathcal{C}^{ec}}(A, B)$, will be denoted by $A \to^{ec} B$. If (e_1, c_1) and (e_2, c_2) are morphisms in $A \to^{ec} B$ and $B \to^{ec} C$ respectively, then their composition $(e_2, c_2) \circ (e_1, c_1) : A \to^{ec} C$ is defined as $(e_2 \circ e_1, c_1 \circ c_2)$. It is trivial to check that this is indeed a morphism. $\mathcal{C}^{ec}$ is called the *category of domains and embedding-closure pairs.* If (e, c) is a morphism, we shall call e an *embedding* and c a *closure.* For any morphism f in $A \to^{ec} B$ we write $f = (f^e, f^c)$, where f^e is the embedding and f^c is the closure.

Remarks

If h and k are morphisms in $A \to^{ec} B$ then

- $h^e \circ h^c$ is a closure on B, in the sense used by Scott in [45].
  This is because

$$\begin{aligned}(h^e \circ h^c) \circ (h^e \circ h^c) &= h^e \circ (h^c \circ h^e) \circ h^c \\ &= h^e \circ h^c\end{aligned}$$

  and by definition $h^e \circ h^c \sqsupseteq id$

- h^e is one-to-one and h^c is strict and onto.
  To see this, suppose $h^e(x) = h^e(y)$. Then applying h^c to both sides of the equation, we obtain $x = y$, which means that h^e is one-to-one. Since $h^c(h^e(\perp)) = \perp$, we must have that $h^c(\perp) = \perp$, and hence h^c is strict. To show that h^c is onto, assume that $x \in A$. Let $y = h^e(x)$. Then $h^c(y) = h^c(h^e(x)) = x$. Thus, h^c is onto.

- h^e is $\perp$-*reflecting*, *i.e*, $h^e(a) = \perp \Rightarrow a = \perp$.
  This is because $h^e(\perp) \sqsubseteq h^e(a)$ and h^e is one-to-one.

- $h^e \sqsubseteq k^e$ if and only if $h^c \sqsupseteq k^c$, and also h^e is uniquely determined by h^c and vice versa.

3.1.1 Functors on $(\mathcal{C}^{ec})^n$

Since we want to model type constructions by functors, it is sufficient to consider a certain inductively defined class of functors. We call such functors *type functors.* These include the identity functor Id and constant functors. Using a notation similar to Abramsky's [2], corresponding to product and function space constructors the mappings $\times$ and $\rightarrow$ are defined as follows.

(i) $\times : \mathcal{C}^{ec} \times \mathcal{C}^{ec} \rightarrow \mathcal{C}^{ec}$

$$\begin{aligned} \times(A, B) &= A \times B \\ \times(f, g) &= (f^e \times g^e, f^c \times g^c) \end{aligned}$$

(ii) $\rightarrow : \mathcal{C}^{ec} \times \mathcal{C}^{ec} \rightarrow \mathcal{C}^{ec}$

$$\begin{aligned} \rightarrow (A, B) &= A \rightarrow B \\ \rightarrow (f, g) &= (\lambda h.g^e \circ h \circ f^c, \lambda h.g^c \circ h \circ f^e) \end{aligned}$$

Here, $A \times B$ denotes the product of A and B in $\mathcal{C}$, and $A \rightarrow B$ denotes the space of continuous functions from A to B. We shall write $f \rightarrow g$ for $\rightarrow (f, g)$. In general, corresponding to type constructors that involve n type variables a functor from $(\mathcal{C}^{ec})^n$ to $\mathcal{C}^{ec}$ is defined.

For any type functor F defined in this way there are two functors

$$F^e : (\mathcal{C}^{ec})^n \rightarrow \mathcal{C} \text{ and } F^c : (\mathcal{C}^{ec})^n \rightarrow \mathcal{C}^{op}$$

which act like F on objects. On morphisms, F^e selects the embedding component of $F(h)$, and F^c selects the closure. That is, for any morphism h $F^e(h) = (F(h))^e$ and $F^c(h) = (F(h))^c$. In much of the discussion that follows we consider the case $n = 1$. That is, we concentrate on functors defined on $\mathcal{C}^{ec}$.

It is possible to introduce an ordering on $A \rightarrow^{ec} B$ by letting $f \sqsubseteq g$ whenever $f^e \sqsubseteq g^e$. Thus, it makes sense to talk about monotonicity of functors. For example, we have the following proposition about the function type functor.

Proposition 3.1.1 *Suppose F is the functor $\rightarrow$. Then the functors F^e and F^c are monotonic in their second and first argument respectively. On the other hand, they are anti-monotonic with respect to their first and second argument respectively.*

We omit the proof because it is very easy.

For any type functors F and G, we define the type functors $F \times G$ and $F \rightarrow G$ by

$$\begin{array}{lrcl} (i) & (F \times G)(A) & = & F(A) \times G(A) \\ & (F \times G)(h) & = & ((F^e \times G^e)(h), (F^c \times G^c)(h)) \\ (ii) & (F \rightarrow G)(A) & = & F(A) \rightarrow G(A) \\ & (F \rightarrow G)(h) & = & (\lambda k.G^e(h) \circ k \circ F^c(h), \lambda k.G^c(h) \circ k \circ F^e(h)) \end{array}$$

3.1.2 Lax Natural Transformations between Functors

Definition 3.1.1 *Let F and G be functors on $\mathcal{C}^{ec}$ and f be a family $\{f_A\}$ of continuous functions indexed by domains and $f_A : F(A) \rightarrow G(A)$. We say that f is a lax natural transformation from F to G, if for any morphism $h : A \rightarrow^{ec} B$ we have*

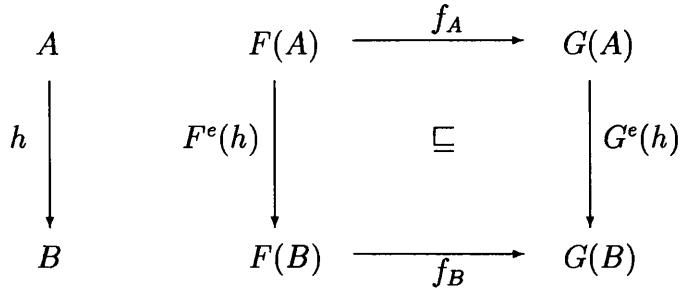
$$G^c(h) \circ f_B \sqsubseteq f_A \circ F^c(h)$$

We call f_A a *component* or an *instance* of the lax natural transformation.

Remarks

- It is important to observe that each f_A is a continuous function, not an embedding-closure pair.
- It is not difficult to see that the inequality in the definition above is equivalent to

$$f_B \circ F^e(h) \sqsubseteq G^e(h) \circ f_A$$



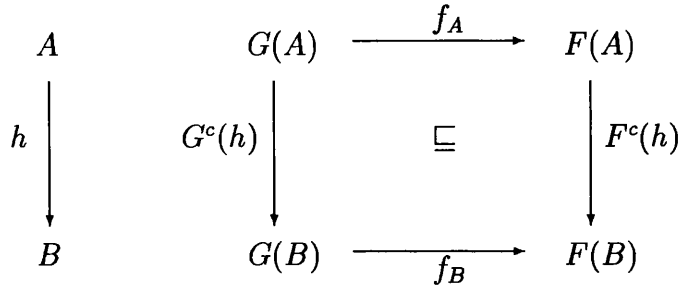
- The definition easily generalises to lax natural transformations between functors which have more than one argument.

Although the definition of lax natural transformations may be given by one of the two equivalent ways, when natural transformations are considered, it is important to note that the statements $f_B \circ F^e(h) = G^e(h) \circ f_A$ and $G^c(h) \circ f_B = f_A \circ F^c(h)$ are not equivalent. To see this, we consider the following example.

- Suppose $F(A) = A \to A$ and $G(A) = A$. If Y_A denotes the fixed point operator, $\{Y_A\}$ is a lax natural transformation from F to G. For any $h : A \to^{ec} B$, it is not difficult to show that
$$Y_B \circ F^e(h) \;=\; G^e(h) \circ Y_A$$
On the other hand, consider the case where $A = \mathbf{2}$ and $B = \{a, b, c\}$, with $a \sqsubset b \sqsubset c$. Define $h : A \to^{ec} B$ and $f : B \to B$ such that $h^e(0) = b$ and $h^e(1) = c$ (it is not difficult to find h^c from this), and $f(a) = a$ and $f(b) = f(c) = c$. Now, $G^c(h)(Y_B(f)) = 0$ and $Y_A(F^c(h)(f)) = 1$. Therefore,
$$G^c(h) \circ Y_B \;\neq\; Y_A \circ F^c(h)$$

Before the end of this chapter, we will encounter an example of a lax natural transformation which is a natural transformation when the closures are used as morphisms, but not with embeddings.

Note also that using the embedding parts, we are describing naturality from F^e to G^e, which are functors from $\mathcal{C}^{ec}$ to $\mathcal{C}$, whereas in the other case it describes naturality from G^c to F^c which are functors from $\mathcal{C}^{ec}$ to $\mathcal{C}^{op}$.



This figure is about morphisms in $\mathcal{C}^{op}$. If we reverse the four arrows in order to read it in $\mathcal{C}$, the resulting inequality is exactly the one given in the definition of a lax natural transformation above.

3.2 Finite Lattices and Embedding-Closure Pairs

When doing strictness analysis, calculation is performed on finite lattices. These are complete and moreover they are domains. Therefore, we study the category $\mathcal{A}^{ec}$ of finite lattices as a subcategory of $\mathcal{C}^{ec}$ more closely. Of special interest will be the morphisms from the two-element domain **2** to other finite lattices and their images under the various functors.

Definition 3.2.1 *Let A be a finite lattice and $a \in A$ be different from the top element $\top_A$ of A. The functions $h_a^e : \mathbf{2} \to A$ and $h_a^c : A \to \mathbf{2}$ are defined by*

$$h_a^e(x) = \begin{cases} a & \text{if } x = 0 \\ \top_A & \text{if } x = 1 \end{cases} \quad \text{and} \quad h_a^c(x) = \begin{cases} 0 & \text{if } x \sqsubseteq a \\ 1 & \text{otherwise} \end{cases}$$

It is now simple to show that $h = (h_a^e, h_a^c)$ is a morphism in $\mathbf{2} \to^{ec} A$. Moreover, every morphism in $\mathbf{2} \to^{ec} A$ has this form. This is because if (f, g) is such a morphism, then letting $a = f(0)$ it is easy to show that $(f, g) = (h_a^e, h_a^c)$.

Our aim is to establish certain relationships between different components or instances of lax natural transformations. In particular, we would like to express the components corresponding to big lattices in terms of those of the smaller ones.

We now come to a very important theorem about lax natural transformations in the category of finite lattices and embedding-closure pairs. It provides an approximation to a component of a lax natural transformation corresponding to any finite lattice from the component corresponding to the lattice **2**.

Theorem 3.2.1 *If f is a lax natural transformation from the functor F to the functor G, and A is any finite lattice then we have*

$$f_A \sqsubseteq \bigsqcap_a G^e(h_a) \circ f_{\mathbf{2}} \circ F^c(h_a)$$

where a ranges over the non-top elements of A.

Proof
Let h be any morphism in $\mathbf{2} \rightarrow^{ec} A$. Then,

$$\begin{array}{lccc} & f_A \circ F^e(h) & \sqsubseteq & G^e(h) \circ f_{\mathbf{2}} \\ \Rightarrow & f_A \circ F^e(h) \circ F^c(h) & \sqsubseteq & G^e(h) \circ f_{\mathbf{2}} \circ F^c(h) \\ \Rightarrow & f_A & \sqsubseteq & G^e(h) \circ f_{\mathbf{2}} \circ F^c(h) \end{array}$$

Since h is arbitrary and is of the form h_a for some a, we have

$$f_A \sqsubseteq \bigsqcap_a G^e(h_a) \circ f_{\mathbf{2}} \circ F^c(h_a)$$

□

Thus, from $f_{\mathbf{2}}$ we obtain an approximate value to f_A. The greatest lower bound in the theorem is taken over $n-1$ values, where n is the size of the lattice A, that is, the number of a's that are allowed in forming h_a. For a certain class of functors, some optimisations that cut down the number of functions involved exist. We will be more precise about these matters later.

If F and G are functors with n parameters, the theorem above would be that for finite lattices $A_1, ..., A_n$

$$f_{A_1 ... A_n} \sqsubseteq \bigsqcap G^e(h_1, ..., h_n) \circ f_{\mathbf{2}...\mathbf{2}} \circ F^c(h_1, ..., h_n)$$

where the greatest lower bound is taken over all possible n-tuples of morphisms $(h_1, ..., h_n)$ with $h_i : \mathbf{2} \rightarrow^{ec} A_i$ for each i.

3.2.1 First-Order Type Functors

Type functors that do not involve the function space functor $\rightarrow$ will be referred to as *first-order type functors.* We now state and prove a number of properties of such functors and lax natural transformations between them.

Proposition 3.2.1 *If F is a first-order type functor then F^e is monotonic and F^c is anti-monotonic. That is, if h and k are embedding-closure pairs in $A \rightarrow^{ec} B$, then whenever $h^e \sqsubseteq k^e$ we have $F^e(h) \sqsubseteq F^e(k)$ and $F^c(h) \sqsupseteq F^c(k)$.*

The proof is straightforward.

Proposition 3.2.2 *Let F be a first-order type functor. Let A be a finite lattice with $a, b \in A$. Suppose $k : \mathbf{2} \rightarrow^{ec} A$ is a morphism. Then*

(i) $F^e(h_{a \sqcap b}) = F^e(h_a) \sqcap F^e(h_b)$ *and* $F^c(h_{a \sqcap b}) = F^c(h_a) \sqcup F^c(h_b)$

(ii) $F^e(k)$ *is distributive over both* $\sqcup$ *and* $\sqcap$, *and* $F^c(k)$ *is distributive over* $\sqcup$

Proof
We will use induction on the structure of the functors.

(i) In the cases where $F = Id$ or F is a constant functor it is trivial to show that the properties hold. Suppose $F = G \times H$ and assume that the properties hold for G and H. Then,

$$\begin{aligned} F^e(h_{a \sqcap b}) &= G^e(h_{a \sqcap b}) \times H^e(h_{a \sqcap b}) \\ &= (G^e(h_a) \sqcap G^e(h_b)) \times (H^e(h_a) \sqcap H^e(h_b)) \\ &= (G^e(h_a) \times H^e(h_a)) \sqcap (G^e(h_b) \times H^e(h_b)) \\ &= F^e(h_a) \sqcap F^e(h_b) \end{aligned}$$

Replacing, where appropriate, e and $\sqcap$ by c and $\sqcup$ respectively in the above proof gives a proof of the second statement.

(ii) Again, the cases where $F = Id$ or F is a constant functor are trivial. Suppose $F = G \times H$ and assume that the properties hold for G and H. Let $x, y \in F(\mathbf{2})$.Then,

$$\begin{aligned} F^e(k)(x \sqcup y) &= G^e(k)(x \sqcup y) \times H^e(k)(x \sqcup y) \\ &= (G^e(k)(x) \sqcup G^e(k)(y)) \times (H^e(k)(x) \sqcup H^e(k)(y)) \\ &= (G^e(k) \times H^e(k))(x) \sqcup (G^e(k) \times H^e(k))(y) \\ &= F^e(k)(x) \sqcup F^e(k)(y) \end{aligned}$$

The other statements can also be proved in a similar way.

□

To see why $F^c(k)$ does not in general distribute over $\sqcap$, consider the case where $F = Id$ and $k \in \mathbf{2} \to^{ec} (\mathbf{2} \times \mathbf{2})$ is the embedding-closure pair (k^e, k^c), where for each x, $k^e(x) = (x, x)$. Now, k^c maps $(0,0)$ to 0 and the rest to 1. Letting $a = (0,1)$ and $b = (1,0)$ we have $F^c(k)(a \sqcap b) = 0$, but $F^c(k)(a) \sqcap F^c(k)(b) = 1$.

We now prove a property of natural transformations between first-order type functors. Before stating the property we prove the following lemmas. The finite lattice A used in the following lemmas and the proposition is assumed to have more than one elements.

Lemma 3.2.1 *Let A be a finite lattice. Then,*

$$\bigsqcap_a h_a^e \circ h_a^c = id_A$$

Proof

Let $a \neq \top_A$. Then $h_a^e(0) = a$ and $h_a^c(a) = 0$, and hence $(h_a^e \circ h_a^c)(a) = a$. Moreover, $(h_a^e \circ h_a^c)(\top_A) = \top_A$. By definition $h_a^e \circ h_a^c \sqsupseteq id_A$. From this it easy to observe that the statement holds. (NB. In oreder for the h_a's to be defined A must have more than one element.) □

Lemma 3.2.2 *Let F be a first-order functor and A be a finite lattice. Then*

$$\bigsqcap_a F^e(h_a) \circ F^c(h_a) = id_{F(A)}$$

Proof
The proof is given by induction on the structure of F.

(i) If $F = Id$ then $F^e(h_a) \circ F^c(h_a) = h_a^e \circ h_a^c$, and hence the statement follows from the previous lemma.

(ii) If F is a constant functor then for every h, $F(h) = id$ (the identity pair). Therefore, $F^e(h) \circ F^c(h) = id_{F(A)}$ (the identity function on $F(A)$) and hence the statement holds.

(iii) Suppose $F = G \times H$.

$$\begin{aligned}
&\textstyle\prod_a (G \times H)^e(h_a) \circ (G \times H)^c(h_a) \\
&\quad = \textstyle\prod_a ((G^e(h_a) \circ G^c(h_a)) \times (H^e(h_a) \circ H^c(h_a))) \\
&\quad = (\textstyle\prod_a G^e(h_a) \circ G^c(h_a), \prod_a H^e(h_a) \circ H^c(h_a)) \\
&\quad = (id_{G(A)}, id_{H(A)}) \\
&\quad = id_{(G \times H)(A)} \\
&\quad = id_{F(A)}
\end{aligned}$$

□

We consider the case of natural transformations between first-order type functors. Here, the naturality that is considered involves the closures.

Proposition 3.2.3 *Let F and G be first-order type functors and A be a finite lattice. If f is a natural transformation from F to G then*

$$f_A \;=\; \textstyle\prod_a G^e(h_a) \circ f_2 \circ F^c(h_a)$$

where a ranges over all non-top elements of A.

Proof
Since f is a natural transformation, for any a in A we have

$$G^c(h_a) \circ f_A \;=\; f_2 \circ F^c(h_a)$$

This implies

$$G^e(h_a) \circ G^c(h_a) \circ f_A = G^e(h_a) \circ f_2 \circ F^c(h_a)$$

since a is arbitrary

$$\bigsqcap_a G^e(h_a) \circ G^c(h_a) \circ f_A = \bigsqcap_a G^e(h_a) \circ f_2 \circ F^c(h_a)$$

But, from the preceding lemma we have

$$\bigsqcap_a G^e(h_a) \circ G^c(h_a) = id_{G(A)}$$

and hence,

$$f_A = \bigsqcap_a G^e(h_a) \circ f_2 \circ F^c(h_a)$$

□

The following proposition is used to reduce the number of functions involved in the computation of the greatest lower bound.

Proposition 3.2.4 *Let F and G be first-order type functors and f be a lax natural transformation from F to G. If a and b are elements of a finite lattice A then*

$$G^e(h_a) \circ f_2 \circ F^c(h_a) \sqcap G^e(h_b) \circ f_2 \circ F^c(h_b) \sqsubseteq G^e(h_{a \sqcap b}) \circ f_2 \circ F^c(h_{a \sqcap b})$$

Proof
Since the functors are first-order, we have

$$\begin{aligned} G^e(h_{a \sqcap b}) \circ f_2 \circ F^c(h_{a \sqcap b}) &= (G^e(h_a) \sqcap G^e(h_b)) \circ f_2 \circ F^c(h_{a \sqcap b}) \\ &= G^e(h_a) \circ f_2 \circ F^c(h_{a \sqcap b}) \sqcap G^e(h_b) \circ f_2 \circ F^c(h_{a \sqcap b}) \\ &\sqsupseteq G^e(h_a) \circ f_2 \circ F^c(h_a) \sqcap G^e(h_b) \circ f_2 \circ F^c(h_b) \end{aligned}$$

□

We now have the following corollary.

Corollary 3.2.1 *If F,G,f and A are as in preceding proposition then*

$$\bigsqcap_a G^e(h_a) \circ f_2 \circ F^c(h_a) \;=\; \bigsqcap_t G^e(h_t) \circ f_2 \circ F^c(h_t)$$

where a ranges over elements of $A - \{\top_A\}$*, and t is restricted to the meet-irreducible ones.*

The consequence of this corollary is that the computation of the greatest lower bound of the functions can be made efficient by using only those which correspond to the meet-irreducible elements. In the case where A is a chain, there is no improvement because all elements are meet-irreducible.

3.2.2 Some Special Transformations

Later, we will use the collections $\{\sqcup_A\}$ and $\{\rhd_A\}$ of functions indexed by finite lattices A. The function $\sqcup_A$ is the usual least upper bound operation on the lattice A. The function $\rhd_A$: $\mathbf{2} \times A \to A$ is defined by

$$\rhd_A (x,y) = \begin{cases} \bot & x = 0 \\ y & x = 1 \end{cases}$$

Clearly both $\sqcup_A$ and $\rhd_A$ are continuous functions. Moreover, both the collections $\{\sqcup_A\}$ and $\{\rhd_A\}$ form lax natural transformations. To see this consider a morphism $h : A \to^{ec} B$ between the finite lattices A and B. For all $x, y \in A$, using infix notation, it is easy to show that

$$h^e(x) \sqcup_B h^e(y) \;\sqsubseteq\; h^e(x \sqcup_A y)$$

Hence, $\{\sqcup_A\}$ is a lax natural transformation from the functor $Id \times Id$ to the identity functor Id. The above inequality cannot, in general, be replaced by equality; there are examples for which equality does not hold. On the other hand, the condition is equivalent to

$$h^c \circ \sqcup_B \;\sqsubseteq\; \sqcup_A \circ (h^c \times h^c)$$

However, we also know that

$$h^c \circ \sqcup_B \sqsupseteq \sqcup_A \circ (h^c \times h^c)$$

Therefore,

$$h^c \circ \sqcup_B = \sqcup_A \circ (h^c \times h^c)$$

Hence $\{\sqcup_A\}$ is a natural transformation when one uses the closure components as morphisms.

Also, again using infix notation, for any $x, y \in A$, we have

$$id(x) \rhd_B h^e(y) \sqsubseteq h^e(x \rhd_A y)$$

Hence $\{\rhd_A\}$ is a lax natural transformation from $F \times Id$ to Id, where F is the constant functor returning **2**. If $x = \perp$ then we have $\perp$ on the left hand side, and $h^e(\perp)$ on the right. Thus, equality does not hold in general; in particular, if we choose h such that h^e is not strict. On the other hand, it can also be shown that

$$h^c \circ \rhd_B = \rhd_A \circ (id \times h^c)$$

Therefore, $\{\rhd_A\}$ is also a natural transformation when we use the closure components as morphisms.

3.3 Embedding-Projection Pairs

The *category of domains and embedding-projection pairs*, $\mathcal{C}^{ep}$, is defined to be the category whose objects are domains and the morphisms are of the form $(e, p) : A \to^{ep} B$ where both $e : A \to B$ and $p : B \to A$ are continuous functions satisfying

$$p \circ e = id \quad \text{and} \quad e \circ p \sqsubseteq id$$

We call e an *embedding* and p a *projection.* We denote the set of all embedding-projection pairs from A to B by $A \to^{ep} B$. If (e_1, p_1) and (e_2, p_2) are morphisms in $A \to^{ep} B$ and $B \to^{ep} C$ respectively, then their composition $(e_2, p_2) \circ (e_1, p_1) : A \to^{ep}$

C is defined to be the pair $(e_2 \circ e_1, p_1 \circ p_2)$. For any morphism f in $A \to^{ep} B$ we will write $f = (f^e, f^p)$, where f^e is the embedding and f^p is the projection. This category has been studied and used extensively in foundational issues in domain theory, and in particular for describing a semantics of the polymorphic λ-calculus by Coquand et al [15].

Remarks

If k and l are morphisms in $A \to^{ep} B$ then

- $k^e \circ k^p$ is a projection on B, in the sense used by Scott in [45].
- k^e is one-to-one and k^p is onto.
- both k^e and k^p are strict.
- since k^e is one-to-one and strict, k^e is $\perp$-reflecting.
- $k^e \sqsupseteq l^e$ if and only if $k^p \sqsubseteq l^p$, and hence k^e is uniquely determined by k^p and *vice-versa.*

3.3.1 Functors on $(\mathcal{C}^{ep})^n$

Here, as for $(\mathcal{C}^{ec})^n$, type constructions involving n type variables may be modelled by functors from $(\mathcal{C}^{ep})^n$ to $\mathcal{C}^{ep}$. The definitions of the functors $\times$ and $\to$, for example, can be given in a similar way; all we have to do is replace the c, wherever it occurs as a superscript in the earlier section, by p. The definition of the other functors are also given in a similar way.

3.3.2 Lax Natural Transformations between Functors on $(\mathcal{C}^{ep})^n$

Definition 3.3.1 *Let F and G be functors and f be a family $\{f_A\}$ of continuous functions indexed by domains and $f_A : F(A) \to G(A)$. We say that f is a lax natural*

transformation from F to G if for every morphism $k : A \to^{ep} B$ we have

$$G^p(k) \circ f_B \sqsupseteq f_A \circ F^p(k)$$

It is easy to show that the inequality above is equivalent to

$$f_B \circ F^e(k) \sqsupseteq G^e(k) \circ f_A$$

However, as before *naturality* in terms of the embeddings is not equivalent to naturality in terms of the projections.

3.4 Finite Lattices and Embedding-Projection Pairs

The category of finite lattices and embedding-projection pairs $\mathcal{A}^{ep}$ is a subcategory of $\mathcal{C}^{ep}$. The morphisms from **2** to any finite lattice my be given by the following definition.

Definition 3.4.1 *Let A be a finite lattice, and let $a \in A$ be different from $\perp_A$. The continuous functions $k_a^e : \mathbf{2} \to A$ and $k_a^p : A \to \mathbf{2}$ are defined by*

$$k_a^e(x) = \begin{cases} a & \text{if } x = 1 \\ \perp_A & \text{if } x = 0 \end{cases} \quad \text{and} \quad k_a^p(x) = \begin{cases} 1 & \text{if } x \sqsupseteq a \\ 0 & \text{otherwise} \end{cases}$$

Clearly (k_a^e, k_a^p) is an embedding-projection pair, and moreover every embedding-projection pair from **2** to A is of this form. Furthermore, we state the following proposition which is not difficult to prove.

Proposition 3.4.1 *For any finite lattice A, we have*

$$\bigsqcup_a k_a^e \circ k_a^p = id_A$$

We now come to a theorem which relates any component, of a lax natural transformation, corresponding to a finite lattice with the component which corresponds to **2**.

Theorem 3.4.1 *If f is a lax natural transformation between the functors F and G, and A is any finite lattice then we have :*

$$f_A \sqsupseteq \bigsqcup_a G^e(k_a) \circ f_2 \circ F^p(k_a)$$

where a ranges over the non-bottom elements of A.

The proof is straightforward.

Again from f_2 we get an approximate value to f_A from below. The least upper bound is taken over $n-1$ functions, where n is the size of A.

3.4.1 First-Order Type Functors

For natural transformations between first-order type functors, the theorem above could be strengthened; in fact we have an equality. We first give a lemma that may be used in its proof.

Lemma 3.4.1 *Let F be a first-order type functor and A be a finite lattice. Then*

$$\bigsqcup_a F^e(k_a) \circ F^p(k_a) = id_{F(A)}$$

Here again, a ranges over non-bottom elements of A.

Next, the case of natural transformations between first-order type functors are considered. The naturality used here is the one that involves the projections.

Proposition 3.4.2 *Let F and G be first-order type functors and A be a finite lattice. If f is a natural transformation from F to G then*

$$f_A = \bigsqcup_a G^e(k_a) \circ f_2 \circ F^p(k_a)$$

where a ranges over all non-bottom elements of A.

It can be shown that in this proposition, the least upper bound may be taken over a's that are join-irreducible. Thus, in the case where A is not a chain, we have a cheaper way of computing it.

3.4.2 A Special Transformation

We will need the collection $\{\sqcap_A\}$ of continuous functions indexed by finite lattices later, where $\sqcap_A$ is the usual meet operation on the finite lattice A. It is a lax natural transformation from the functor $Id \times Id$ to Id. To show this consider any morphism $k : A \to^{ep} B$ between finite lattices A and B. Let $x, y \in B$. Using the infix notation,

$$\begin{aligned} k^p(x \sqcap_B y) &\sqsupseteq k^p(k^e(k^p(x)) \sqcap_B k^e(k^p(y))) \\ &\sqsupseteq k^p(k^e(k^p(x) \sqcap_A k^p(y))) \\ &= k^p(x) \sqcap_A k^p(y) \end{aligned}$$

Hence, it is a lax natural transformation. On the other hand,

$$k^p(x \sqcap_B y) \sqsubseteq k^p(x) \sqcap_A k^p(y)$$

Therefore, k^p distributes over $\sqcap_B$ and hence the collection is a natural transformation when projections are used as morphisms. It can be shown that k^e also distributes over $\sqcap_A$. Therefore, the collection is also a natural transformation if we use the embeddings as morphisms.

We will see how to apply these results in the course of the next few chapters.

Chapter 4

Semantics of a Polymorphic Language

In this chapter we introduce a polymorphic language that will be used in later chapters to illustrate our work on abstract interpretation of polymorphic functions. A semantics for this language is also presented. In this semantics any polymorphic function is viewed as a collection of functions where each function in the collection is the semantics of some instance of the polymorphic function. In Chapter 5, a relationship between the semantics of different instances of any polymorphic function is established.

4.1 Introduction

There are two commonly used forms of polymorphism in programming languages. One arises from the Hindley-Milner type system [35] found in languages such as ML and Miranda† [50]. Normally, expressions in programs are untyped and their types are inferred at compile-time. Girard [17] and Reynolds [42] independently developed a calculus of the second form of polymorphism where types appear explicitly in expressions. In this explicitly typed system type-checking is done at compile-time to ensure that the type of every expression conforms with the typing rules.

†Miranda is a trademark of Research Software Limited.

The important difference between the two forms of polymorphism comes from the definition of types used in both systems and from the rules used in assigning types to expressions. We will see this more clearly in the next section.

Extensive work has been done, and still continues to be done, on compilers for languages with the Hindley-Milner type system. Moreover such compilers are widely available. We will therefore concentrate on languages with the Hindley-Milner type system. It is also the case that the semantics of such languages is easier to describe. However, in order to use type information for various program analyses, it is preferable to work with explicitly typed languages. This should not be regarded as a restriction because types of expressions can be obtained by inference at compile-time. A good example of such a language is *Core-XML*, a language introduced by Mitchell and Harper in [36]. *Core-XML* is an explicitly typed fragment of Standard ML.

First we define an explicitly typed language similar to *Core-XML*. The difference is that it has the fixed-point operator `fix`, a pairing operation `< , >`, and some more operations. Mitchell and Harper point out that *Core-XML* has classical set-theoretic models, and Ohori [39] actually describes such a model. The semantics provided here uses domains instead of sets so that `fix` can be assigned a meaning.

4.2 Syntax

4.2.1 Type Expressions

We assume that there are some base types, the type of integers **int** and the type of booleans **bool** are among them. We use b to denote any base type. There are two classes of types, and the first is that of types defined by the grammar

$$\tau ::= t \mid b \mid \tau \times \tau \mid \tau \to \tau$$

where t stands for type variables. Types ranged over by τ and not containing type variables are called *monotypes*.

The second class is given by

$$\sigma ::= \tau \mid \forall t.\sigma$$

Types ranged over by σ are called *type-schemes* and types of the form $\forall t.\sigma$ are called *polytypes*. We shall write $\forall t_1...t_n.\sigma$ for $\forall t_1...\forall t_n.\sigma$. Free and bound type variables, and the substitution $\sigma_2[\sigma_1/t]$ of a type σ_1 for a free type variable t in the type σ_2 are defined in the usual way.

It is important to observe that here every polytype σ is of the form $\forall t_1...t_n.\tau$ and hence type-schemes are not closed under $\times$ or $\rightarrow$. For example,
$(\forall t.t \rightarrow t) \rightarrow (\forall t.t \rightarrow t)$ is not a type in this system. On the other hand, the types used by Girard and Reynolds may be given by the single definition

$$\xi ::= t \mid b \mid \xi \times \xi \mid \xi \rightarrow \xi \mid \forall t.\xi$$

Therefore, they are closed under both $\times$ and $\rightarrow$.

4.2.2 Terms

The language we are interested in is an extension of *Core-XML*. The *pre-terms* of the language are given by the following grammar.

$$\begin{aligned} E \quad ::= \quad & x \mid c \mid \lambda x : \tau.E \mid (E\ E) \mid \texttt{if}\ \tau\ E\ E\ E \mid \texttt{fix}\ \tau\ E \\ & \mid < E, E > \mid \texttt{fst}\ \tau \times \tau\ E \mid \texttt{snd}\ \tau \times \tau\ E \\ & \mid \texttt{let}\ x : \sigma = E\ \texttt{in}\ E \mid \Lambda t.E \mid E[\tau] \end{aligned}$$

Not all expressions defined in this way are terms of the language. Only well-typed terms are permitted. It is therefore necessary to provide typing rules to decide if a term is well-typed and hence legal. Free and bound (term) variables are defined in the usual way. The expression $E_2[E_1/x]$ is used to denote the term that is obtained by substituting E_1 for the free occurrences of x in E_2.

A type variable t is said to be free in an expression E, if it is free in any of the types occurring in E. The substitution $E_2[\tau/t]$ is defined as the expression that is obtained by substituting τ for the free occurrences of t. It is not necessary to define $E_2[\sigma/t]$ for arbitrary σ (and hence possibly belonging to the second class of types) because such expressions never arise in the process of evaluation of terms.

The most significant difference of this language from the simply-typed λ-calculus is that here there are type abstraction and type application. Moreover, `let` provides a restricted version of binding a variable with a polytype. On the other hand, it is very important to note that the type τ in the abstraction $\lambda x : \tau.E$ and the application $E[\tau]$ is not a polytype. In the second-order λ-calculus, the calculus developed by Girard and by Reynolds, there is no such restriction. It is this generality which makes the description of the semantics of this calculus very complicated.

4.2.3 Type Checking Rules

We introduce some notation for typing rules. A statement which expresses that a term E has a type σ is written as $E : \sigma$. The type of a term depends on the types of the free term variables occurring in it. Thus, a statement of the form $E : \sigma$ is made under a *type assumption* (or *type assignment*), which is an association of term variables with types. More precisely, a type assumption Γ is a partial function from the set of term variables to types. It may also be expressed as a set $\{x_1 : \sigma_1, ..., x_n : \sigma_n\}$. If x is not in the domain of Γ (written $dom(\Gamma)$), we shall write $\Gamma, x : \sigma$ for $\Gamma \cup \{x : \sigma\}$.

We write the formula $\Gamma \vdash E : \sigma$ to say that E has the type σ under the type assumption Γ. We assume that for each constant c we have a rule
$\{\} \vdash c : \sigma$ for some type σ. The type rules are given in Figure 4.1.

(VAR) $$\Gamma, x : \sigma \vdash x : \sigma$$

(ABS) $$\frac{\Gamma, x : \tau \vdash E : \tau'}{\Gamma \vdash \lambda x : \tau . E : \tau \to \tau'}$$

(APP) $$\frac{\Gamma \vdash E_1 : \tau \to \tau' \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash (E_1 E_2) : \tau'}$$

(PROD) $$\frac{\Gamma \vdash E_1 : \tau \quad \Gamma \vdash E_2 : \tau'}{\Gamma \vdash < E_1, E_2 > : \tau \times \tau'}$$

(PROJ) $$\frac{\Gamma \vdash E : \tau \times \tau'}{\Gamma \vdash \mathtt{fst}\ \tau \times \tau'\ E : \tau} \qquad \frac{\Gamma \vdash E : \tau \times \tau'}{\Gamma \vdash \mathtt{snd}\ \tau \times \tau'\ E : \tau'}$$

(TABS) $$\frac{\Gamma \vdash E : \sigma}{\Gamma \vdash \Lambda t . E : \forall t . \sigma} \qquad (t \text{ not free in } \Gamma)$$

(TAPP) $$\frac{\Gamma \vdash E : \forall t . \sigma}{\Gamma \vdash E[\tau] : \sigma[\tau / t]}$$

(LET) $$\frac{\Gamma \vdash E_1 : \sigma \quad \Gamma, x : \sigma \vdash E_2 : \tau}{\Gamma \vdash \mathtt{let}\ x : \sigma\ =\ E_1\ \mathtt{in}\ E_2 : \tau}$$

(COND) $$\frac{\Gamma \vdash E_0 : \mathbf{bool} \quad \Gamma \vdash E_1 : \tau \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash \mathtt{if}\ \tau\ E_0\ E_1\ E_2 : \tau}$$

(FIX) $$\frac{\Gamma \vdash E : \tau \to \tau}{\Gamma \vdash \mathtt{fix}\ \tau\ E : \tau}$$

Figure 4.1: Type Rules

Notice that the rule (TABS) has the restriction that t is not free in Γ. This means that t is not free in any of the types that are associated with the variables by Γ. That is, t is not free in $\Gamma(x)$ for every $x \in dom(\Gamma)$.

4.3 Semantics

The semantics of the simply typed λ-calculus is normally given by assigning a set to each base type, and then sets appropriate to higher-order types are constructed by induction on the structure of the types. The meaning of each term is then defined to be some value in the set corresponding to the type of the term. For languages that allow recursion, sets with some order structure on them are used.

In this section, we use domains to interpret types from the first class (which are either monotypes or type variables). Then, the semantic functions which assign values to terms are defined. We will not need to be more explicit about the semantics of polymorphic terms except to define the semantics of any such term to be a collection of values, where each value is the semantics of some instance of the polymorphic term.

4.3.1 Semantics of Types

The semantics of a monotype τ is normally specified by associating it with a domain D_τ. Each base type b is associated with a domain D_b. We shall normally write Int and $Bool$ for the domains of integers and booleans respectively. That is,

$$Int = \{\bot\} \cup \mathcal{Z} \text{ and } Bool = \{\bot, True, False\}$$

where $\mathcal{Z}$ is the set of all integers. The domain, $D_{\tau_1 \times \tau_2}$, associated with the type $\tau_1 \times \tau_2$ is defined to be the product $D_{\tau_1} \times D_{\tau_2}$. $D_{\tau_1 \to \tau_2}$ is defined to be the set, $D_{\tau_1} \to D_{\tau_2}$, of continuous functions from D_{τ_1} to D_{τ_2}.

The semantics of a type τ which contains type variables depends on the domains associated with the variables. Therefore, we start with an assignment S of domains to type variables. We shall call such an assignment *domain assignment.* Any domain

assignment can be extended to a function $\Im$ which provides a semantics to all type expressions in the first class. The semantics of τ is given as $\Im[\![\tau]\!]\ S$, where it is defined by induction on the structure of τ as follows

$$\begin{aligned}
\Im[\![b]\!]\ S &= D_b, \qquad \text{for any base type } b \\
\Im[\![t]\!]\ S &= S(t) \\
\Im[\![\tau_1 \times \tau_2]\!]\ S &= (\Im[\![\tau_1]\!]\ S) \times (\Im[\![\tau_2]\!]\ S) \\
\Im[\![\tau_1 \to \tau_2]\!]\ S &= (\Im[\![\tau_1]\!]\ S) \to (\Im[\![\tau_2]\!]\ S)
\end{aligned}$$

4.3.2 Semantics of Terms

The meaning of a term will be defined to be a value in the domain that corresponds to its type. The domain depends on the domain assignment the type is evaluated under. Since the term may have ordinary variables occurring freely in it, obviously its value will depend on the values bound to these variables. Such an association of values with variables is given by an *environment*, which is a partial function from the set of ordinary variables to a set of values.

In the following definition, we follow Reynolds's approach [43]. For any type assignment Γ and type τ, we let $A_{\Gamma,\tau}$ to be the set of all terms E such that $\Gamma \vdash E : \tau$. Now a semantic function $\mu_{\Gamma,\tau}$ will be defined on this set; it will also have domain assignments and environments as parameters.

Let S be a domain assignment and η be an environment. We define $\mu_{\Gamma,\tau}[\![E]\!]\ S\ \eta$ to be some value in $\Im[\![\tau]\!]\ S$. From now on, when we deal with this kind of expression we will assume that the variables in the environment are bound to values of the correct type, *i.e*, $\eta(x) \in \Im[\![\Gamma(x)]\!]\ S$. Moreover, we will assume that $\eta(x)$ is defined only when $\Gamma(x)$ is either a type variable or a monotype.

As in Reynolds [43], we assume that for each base type b and the set K_b of constants of type b, there is a function $\alpha_b : K_b \to D_b$ that provides an interpretation to the constants.

Now, the semantic functions are defined by structural induction on terms.

(1) If $k \in K_b$ then $\mu_{\Gamma,b}[\![k]\!]\ S\ \eta = \alpha_b(k)$

(2) If $x \in dom(\Gamma)$ then $\mu_{\Gamma,\Gamma(x)}[\![x]\!]\ S\ \eta = \eta(x)$

(3) If $E_1 \in A_{\Gamma,\tau\to\tau'}$ and $E_2 \in A_{\Gamma,\tau}$ then

$$\mu_{\Gamma,\tau'}[\![(E_1 E_2)]\!]\ S\ \eta \;=\; (\mu_{\Gamma,\tau\to\tau'}[\![E_1]\!]\ S\ \eta)(\mu_{\Gamma,\tau}[\![E_2]\!]\ S\ \eta)$$

(4) If $E \in A_{(\Gamma,x:\tau),\tau'}$ then

$$\mu_{\Gamma,\tau\to\tau'}[\![\lambda x:\tau.E]\!]\ S\ \eta \;=\; \lambda d \in (\Im[\![\tau]\!]\ S).\mu_{(\Gamma,x:\tau),\tau'}[\![E]\!]\ S\ (\eta[d/x])$$

(5) If $E_1 \in A_{\Gamma,\tau}$ and $E_2 \in A_{\Gamma,\tau'}$ then
$\mu_{\Gamma,\tau\times\tau'}[\![< E_1, E_2 >]\!]\ S\ \eta = (\mu_{\Gamma,\tau}[\![E_1]\!]\ S\ \eta, \mu_{\Gamma,\tau'}[\![E_2]\!]\ S\ \eta)$

(6) If $E \in A_{\Gamma,\tau\times\tau'}$ then

$\mu_{\Gamma,\tau}[\![\mathtt{fst}\ \tau\times\tau'\ E]\!]\ S\ \eta = fst(\mu_{\Gamma,\tau\times\tau'}[\![E]\!]\ S\ \eta)$, and
$\mu_{\Gamma,\tau'}[\![\mathtt{snd}\ \tau\times\tau'\ E]\!]\ S\ \eta = snd(\mu_{\Gamma,\tau\times\tau'}[\![E]\!]\ S\ \eta)$

(7) Let $c = \mu_{\Gamma,\mathbf{bool}}[\![E_0]\!]\ S\ \eta$. Then

$$\mu_{\Gamma,\tau}[\![\mathtt{if}\ \tau\ E_0\ E_1\ E_2]\!]\ S\ \eta = \begin{cases} \mu_{\Gamma,\tau}[\![E_1]\!]\ S\ \eta & \text{if } c = True \\ \mu_{\Gamma,\tau}[\![E_2]\!]\ S\ \eta & \text{if } c = False \\ \bot & \text{if } c = \bot \end{cases}$$

(8) $\mu_{\Gamma,\tau}[\![\mathtt{fix}\ \tau\ E]\!]\ S\ \eta = \bigsqcup_{n=0}^{\infty}(\mu_{\Gamma,\tau\to\tau}[\![E]\!]\ S\ \eta)^n(\bot)$

(9) $\mu_{\Gamma,\tau}[\![\mathtt{let}\ x:\sigma = E_1\ \mathtt{in}\ E_2]\!]\ S\ \eta = \mu_{\Gamma,\tau}[\![E_2[E_1/x]]\!]\ S\ \eta$

(10) If $\Gamma \vdash \Lambda t_1...\Lambda t_n.E : \forall t_1..t_n.\tau$ then for types $\tau_1, ..., \tau_n$

$$\mu_{\Gamma,\tau}[\![(\Lambda t_1...\Lambda t_n.E)[\tau_1]...[\tau_n]]\!]\ S\ \eta \ = \ \mu_{\Gamma,\tau}[\![E]\!]\ S[D_1/t_1]...[D_n/t_n]\ \eta$$

where for each i

$$D_i = \Im[\![\tau_i]\!]\ S$$

Remarks

(i) The functions *fst* and *snd*, used in (6) above, are the usual projection functions on products of two domains.

(ii) If a term has no type variables occurring freely in it then the definition above is independent of S. It is also Independent of η if it is closed, *i.e*, if there are no term variables occurring freely in it.

(iii) Consider any closed polymorphic term $\Lambda t_1...\Lambda t_n.E$ of type $\forall t_1...t_n.\tau$. The semantics of this term, because of (10) above, can be given as the collection $\{f_{D_1...D_n}\}$, where for any Γ, η, and S mapping each t_i to D_i

$$f_{D_1...D_n} = \mu_{\Gamma,\tau}[\![E]\!]\ S\ \eta$$

4.4 Summary

The semantics provided above is similar to the one given by Ohori [39]. Essentially, we start with a model for the simply typed λ-calculus and use the usual set-theoretic constructions to obtain a model for the polymorphic language. Now, the semantics of a polymorphic function is regarded as a collection of continuous functions. The semantics of each monomorphic instance is a member of this collection. Recall that our interest is in establishing a relationship between any pair of such continuous functions. This is done in the next chapter.

Chapter 5

Relational Semantics

Different abstract interpretations are presented in Chapters 6, 7 and 8. The properties studied are to do with strictness, binding-time and termination. The abstract interpretation used in each analysis may be viewed as a (non-standard) semantics of the language. In all three cases any polymorphic function is interpreted as a collection of continuous functions. In each analysis, different instances of the same polymorphic function are somehow to be related.

In order not to be repetitive, we first consider the standard semantics defined in the previous chapter. In this chapter, we establish a relationship between the semantics of different instances of any polymorphic function. Once this is done, the case of all the non-standard semantics that we study in later chapters becomes significantly easier. Also, as a result of the relationship we establish in this chapter, it is possible to show the semantic polymorphic invariance of strictness.

5.1 Introduction

For a polymorphic language, we showed that type constructors may be interpreted as functors and polymorphic functions as lax natural transformations (Baraki [6]). The category used there has domains as objects and embedding-closure pairs as morphisms. The lax natural transformation condition relates the values of any two

instances of a polymorphic function. From such a relationship we obtain an approximate value to an instance of a polymorphic function from the value of the smallest instance. Any property, relating to strictness, that holds for the approximate value also holds for the actual value. Thus the procedure gives correct results. It was for this reason that we chose to work with the category of domains and embedding-closure pairs. As one might expect from any approximation, the information may not always be as precise as the one one would obtain from the actual value.

In this chapter we prove similar results for the language defined earlier. This language may be seen as an extension of the one used in [6] because, among other things, it allows `let`-declarations. It will be shown that the results follow from what we shall call, following Reynolds [42], the *representation theorem.* The representation theorem is a statement about the relationship between meanings of an expression obtained by evaluating in different environments. It is very similar to Reynolds's theorem about an extended typed λ-calculus in [43], where he proves an *abstraction theorem* which generalises his representation theorem of [42].

The statement and proof of the representation theorem will not involve any terminology from category theory. However, we show later that one of its consequences is that polymorphic functions are lax natural transformations. From this we obtain a proof of a partial result about the semantic polymorphic invariance of strictness. That is, if the smallest instance of a polymorphic function is strict then so is any other instance. In order to prove the full result, we will later look at how the category of domains and embedding-projection pairs may be used in describing another semantics of the language.

5.2 The Representation Theorem

Reynolds [42] defines a *representation* between domains D and D' as a pair (ϕ, ψ) of continuous functions $\phi : D \to D'$ and $\psi : D' \to D$ satisfying

$$\psi \circ \phi \sqsupseteq id \quad \text{and} \quad \phi \circ \psi \sqsubseteq id$$

An element $d \in D$ is said to *represent* (or be *related* to by (ϕ, ψ)) an element $d' \in D'$ if $d \sqsubseteq \psi(d')$ (or equivalently $\phi(d) \sqsubseteq d'$). What Reynolds's representation theorem then essentially says is that, if two environments are related then the values of a term obtained by evaluating under the two environments are also related. Reynolds also remarks that the theorem could also be stated where the first condition above is changed to $\psi \circ \phi = id$, that is, if (ϕ, ψ) is an embedding-projection pair. His initial attempt to prove the abstraction theorem was for the set-theoretic semantics of the Girard-Reynolds calculus. He later showed, however, that this cannot be done [44].

We work with embedding-closure pairs here. If $h : A \rightarrow^{ec} B$ is an embedding-closure pair then $h^e \circ h^c \sqsupseteq id$. The pair can also be seen as a representation (between B and A) because $h^c \circ h^e = id$ and hence $h^c \circ h^e \sqsubseteq id$. We choose to have the condition $h^c \circ h^e = id$ in the definition because h^e is one-to-one and hence A is normally "smaller" than B. The applications we consider in the next three chapters involve relating the two-element lattice **2** and arbitrary finite lattices. This is because we want to use the properties of the smallest instance of a polymorphic function in analysing other instances.

Now, we say that $d \in D$ and $d' \in D'$ are said to be related by $h : D \rightarrow^{ec} D'$ if $h^e(d) \sqsupseteq d'$, or equivalently if $d \sqsupseteq h^c(d')$. Thus, this definition is essentially the same as the definition of relatedness given above. However, the functions expressing the relationship here have to satisfy the condition $h^c \circ h^e = id$

The language we are investigating allows recursive definitions and polymorphic functions. Recall that the polymorphism used here is the one found in the Hindley-Milner type system [35].

We have already seen that the semantics of a term depends on the domains assigned to its free type variables, and also on the values assigned to its free variables. Suppose S_1 and S_2 are domain assignments, and that η_1 and η_2 are environments. Given a formula $\Gamma \vdash E : \tau$, our aim is to establish some relationship between the values

$$\mu_{\Gamma,\tau}[\![E]\!]\ S_1\ \eta_1 \quad \text{and} \quad \mu_{\Gamma,\tau}[\![E]\!]\ S_2\ \eta_2$$

Clearly these values belong to the spaces $\Im[\![\tau]\!]\ S_1$ and $\Im[\![\tau]\!]\ S_2$ respectively. In order to relate the two values we assume that we have embedding-closure pairs

between $S_1(t)$ and $S_2(t)$ for type variables t. The two environments are now said to be *related* if for each x, $\eta_1(x)$ and $\eta_2(x)$ are related. Such values are related by some embedding-closure pair between the spaces associated with the type of x; the appropriate embedding-closure pair must be obtainable from S_1 and S_2.

We have already seen in Chapter 4 that every domain assignment, which is an assignment of domains to type variables, extends to a semantics of types. Now, suppose that ρ is an assignment of embedding-closure pairs to type variables, such that for each t, $\rho(t)$ is an embedding-closure pair from $S_1(t)$ to $S_2(t)$. For each type τ, we can extend ρ to an embedding-closure pair from $\Im[\![\tau]\!]\ S_1$ to $\Im[\![\tau]\!]\ S_2$; this extension establishes a relationship between these spaces corresponding to τ. It shall be denoted by $\Im[\![\tau]\!]\ \rho$. The definition is given by induction on the structure of types.

$$\begin{aligned}
\Im[\![b]\!]\ \rho &= id_{D_b}, \qquad \text{for base types } b \\
\Im[\![t]\!]\ \rho &= \rho(t) \\
\Im[\![\tau_1 \times \tau_2]\!]\ \rho &= (\Im[\![\tau_1]\!]\ \rho) \times (\Im[\![\tau_2]\!]\ \rho) \\
\Im[\![\tau_1 \to \tau_2]\!]\ \rho &= (\lambda k.(\Im[\![\tau_2]\!]\ \rho)^e \circ k \circ (\Im[\![\tau_1]\!]\ \rho)^c, \lambda k.(\Im[\![\tau_2]\!]\ \rho)^c \circ k \circ (\Im[\![\tau_1]\!]\ \rho)^e)
\end{aligned}$$

If τ is closed then $\Im[\![\tau]\!]\ S_1$ and $\Im[\![\tau]\!]\ S_2$ are the same and $\Im[\![\tau]\!]\ \rho$ is the identity.

We are now ready to state and prove the representation theorem.

Theorem 5.2.1 *Let S_1 and S_2 be domain assignments. Let Γ be a type assignment. Let η_1 and η_2 be environments. Suppose for each type variable t, we have an embedding-closure pair $\rho(t) : S_1(t) \to^{ec} S_2(t)$. Let h_τ stand for $\Im[\![\tau]\!]\ \rho$.*

If η_1 and η_2 are related, i.e, *if for all $x \in dom(\Gamma)$*

$$h^e_{\Gamma(x)}(\eta_1(x)) \ \sqsupseteq\ \eta_2(x),$$

then for each formula $\Gamma \vdash E : \tau$ we have

$$h^e_\tau(\mu_{\Gamma,\tau}[\![E]\!]\ S_1\ \eta_1) \ \sqsupseteq\ \mu_{\Gamma,\tau}[\![E]\!]\ S_2\ \eta_2$$

The proof is given in the Appendix.

5.3 Implications of the Theorem

Our aim is to develop a relationship between the semantics of different instances of a polymorphic function. This relationship is an important consequence of the representation theorem. To see this we first consider the case of one type variable.

Suppose that we have the formula $\Gamma \vdash \Lambda t.E : \forall t.\tau$ for some type assignment Γ, and t not free in Γ. For any types τ_1 and τ_2, we want to see how the semantics of $(\Lambda t.E)[\tau_1]$ and $(\Lambda t.E)[\tau_2]$ are related. Let S be a domain assignment and η be an environment. The relationship between the instances depends on the relationship between τ_1 and τ_2. Since such relationships are expressed by embedding-closure pairs, assume that h is an embedding-closure pair from $\Im[\![\tau_1]\!]\ S$ to $\Im[\![\tau_2]\!]\ S$. For $i = 1, 2$ define the domain assignments S_i by

$$S_i(s) = \begin{cases} \Im[\![\tau_i]\!]\ S & \text{if s = t} \\ S(s) & \text{otherwise} \end{cases}$$

Sometimes the notation $S[(\Im[\![\tau_i]\!]\ S)/t]$ is use to express S_i. In any case, S_1 and S_2 are the same except possibly at t. Also define ρ by

$$\rho(s) = \begin{cases} h & \text{if s = t} \\ id & \text{otherwise} \end{cases}$$

Now, for every type variable s, $\rho(s)$ is an embedding-closure pair from $S_1(s)$ to $S_2(s)$. For each type τ, let h_τ be the extension

$$\Im[\![\tau]\!]\ \rho : \Im[\![\tau]\!]\ S_1 \rightarrow^{ec} \Im[\![\tau]\!]\ S_2$$

Since t is not free in Γ, for each $x \in dom(\Gamma)$, $h_{\Gamma(x)}$ must be the identity. Hence η is related to itself. Thus, by the representation theorem,

$$h^e_\tau(\mu_{\Gamma,\tau}[\![E]\!]\ S_1\ \eta) \ \sqsupseteq\ \mu_{\Gamma,\tau}[\![E]\!]\ S_2\ \eta$$

From the previous chapter we know that for $i = 1, 2$,

$$\mu_{\Gamma,\tau}[\![E]\!]\ S_i\ \eta \ = \ \mu_{\Gamma,\tau[\tau_i/t]}[\![(\Lambda t.E)[\tau_i]]\!]\ S\ \eta$$

Hence,

$$h^e_\tau(\mu_{\Gamma,\tau[\tau_1/t]}[\![(\Lambda t.E)[\tau_1]]\!]\ S\ \eta) \ \sqsupseteq\ \mu_{\Gamma,\tau[\tau_2/t]}[\![(\Lambda t.E)[\tau_2]]\!]\ S\ \eta$$

Assuming that $\Im[\![\tau]\!]\ S_1$ and $\Im[\![\tau]\!]\ S_2$ are related by the embedding-closure pair h, what we have shown here is how the two instances $(\Lambda t.E)[\tau_1]$ and $(\Lambda t.E)[\tau_2]$ are related. In general, if we have the formula

$$\Gamma \vdash \Lambda t_1 ... \Lambda t_n.E : \forall t_1 .....t_n.\tau$$

and types $\tau_1, ..., \tau_n, \tau'_1, ..., \tau'_n$, and embedding-closure pairs

$$h_i : \Im[\![\tau_i]\!]\ S \rightarrow^{ec} \Im[\![\tau'_i]\!]\ S$$

for $i = 1, ..., n$, then it can be shown that

$$h^e_\tau(\mu_{\Gamma,\tau[\tau_1/t_1,...,\tau_n/t_n]}[\![E_0]\!]\ S\ \eta) \ \sqsupseteq\ \mu_{\Gamma,\tau[\tau'_1/t_1,...,\tau'_n/t_n]}[\![E'_0]\!]\ S\ \eta$$

where

$$\begin{aligned} E_0 &= (\Lambda t_1 ... \Lambda t_n.E)[\tau_1]...[\tau_n] \\ E'_0 &= (\Lambda t_1 ... \Lambda t_n.E)[\tau'_1]...[\tau'_n] \end{aligned}$$

Here, h_τ is obtained by extending the h_i's. This is done by first defining two domain assignments S_1 and S_2, where

$$\begin{aligned} S_1(t_i) &= \Im[\![\tau_i]\!]\ S, \quad \text{and} \\ S_2(t_i) &= \Im[\![\tau'_i]\!]\ S, \end{aligned}$$

for $i = 1, ..., n$. For each type variable t different from t_i, we let $S_1(t) = S_2(t) = S(t)$. We also define ρ by letting $\rho(t_i) = h_i$ for each i, and $\rho(t) = id$ for all t different from t_i.

5.4 Polymorphic Functions as Lax Transformations

Hughes studied a polymorphic first-order language, where types are interpreted as domains and type constructors as functors between some category of domains [21]. The functors are defined from the category of domains and strict continuous functions to the category of domains and continuous functions. It was shown that

polymorphic first-order functions are natural transformations. When higher-order functions are introduced one immediately observes that the functor corresponding to the function type constructor $\rightarrow$ is contravariant in its first argument. We overcome this difficulty by using the category of domains and embedding-closure pairs that was introduced in Chapter 3; the details are given in this section.

5.4.1 Type Constructors as Functors

In the previous chapter, for domain assignment S and type τ we defined the semantics of τ to be $\Im[\![\tau]\!]\ S$. Now, suppose that S_1 and S_2 are domain assignments, and that for each type variable t, $\rho(t)$ is an embedding-closure pair from $S_1(t)$ to $S_2(t)$. For any τ, we showed how $\Im[\![\tau]\!]\ \rho$ is defined as an embedding-closure pair from $\Im[\![\tau]\!]\ S_1$ to $\Im[\![\tau]\!]\ S_2$.

Suppose $\forall t_1...t_n.\tau$ is closed. Consider the mapping which

(i) takes any n-tuple of domains $(D_1, ..., D_n)$ and returns $\Im[\![\tau]\!]\ S[D_1/t_1, ..., D_n/t_n]$, and

(ii) given any n-tuple $(h_1, ..., h_n)$ of embedding-closure pairs, with $h_i : D_i \rightarrow^{ec} D_i'$ for each i, it returns $\Im[\![\tau]\!]\ \rho'$, where

$$\rho'(t_i) = \begin{cases} h_i & \text{for each } i : 1 \leq i \leq n \\ \rho(t) & \text{otherwise} \end{cases}$$

It is not difficult to verify that this mapping is actually a functor from $(\mathcal{C}^{ec})^n$ to $\mathcal{C}^{ec}$.

Given the closed type $\forall t_1...t_n.\tau$, it is perhaps easier to see what the associated functor from $(\mathcal{C}^{ec})^n$ to $\mathcal{C}^{ec}$ is, if it is given as $\mathcal{F}[\![\forall t_1...t_n.\tau]\!]$ and is defined by induction on the structure of τ

$$\begin{aligned}
\mathcal{F}[\![\forall t_1...t_n.b]\!] &= Id \\
\mathcal{F}[\![\forall t_1...t_n.t_i]\!] &= Sel_i \\
\mathcal{F}[\![\forall t_1...t_n.\tau_1 \times \tau_2]\!] &= (\mathcal{F}[\![\forall t_1...t_n.\tau_1]\!]) \times (\mathcal{F}[\![\forall t_1...t_n.\tau_2]\!]) \\
\mathcal{F}[\![\forall t_1...t_n.\tau_1 \rightarrow \tau_2]\!] &= (\mathcal{F}[\![\forall t_1...t_n.\tau_1]\!]) \rightarrow (\mathcal{F}[\![\forall t_1...t_n.\tau_2]\!])
\end{aligned}$$

where, Sel_i is the functor which maps any n-tuple of domains to the i-th domain, and also the i-th morphism when provided with an n-tuple of morphisms. The operators on functors used in the right hand side of the last two equations in the above definition are as defined in Chapter 3.

What we have shown here is how closed polytypes are interpreted as functors on a product of the category of domains and embedding-closure pairs.

5.4.2 Polymorphic Functions

Because of what we have already shown, we use type constructors and their corresponding functors interchangeably. Also, we use the notation $\vec{t}$ for the n-tuple of type variables $(t_1, ..., t_n)$. Now, returning to the representation theorem, suppose

$$\Gamma \vdash \Lambda\vec{t}.E : \forall\vec{t}.F(\vec{t}) \to G(\vec{t})$$

where $\Lambda\vec{t}.E$ is a closed term. Let S be any domain assignment and η be any environment. For any n-tuple of domains $(D_1, ..., D_n)$, the semantics of the instance corresponding to this n-tuple is given by

$$f_{D_1...D_n} = \mu_{\Gamma,F(\vec{t})\to G(\vec{t})}[\![E]\!]\ S[D_i/t_i]\ \eta$$

If for each i, D_i is the semantics of some τ_i then

$$f_{D_1...D_n} = \mu_{\Gamma,F(\tau_1,...,\tau_n)\to G(\tau_1,...,\tau_n)}[\![(\Lambda\vec{t}.E)[\tau_1]...[\tau_n]]\!]\ S[D_i/t_i]\ \eta$$

Now, if for each $i(1 \le i \le n)$ we have embedding-closure pairs $h_i : D_i \to^{ec} D'_i$, what the representation theorem says is that

$$(F \to G)^e(h_1, ..., h_n)(f_{D_1...D_n}) \sqsupseteq f_{D'_1...D'_n}$$

From this, letting $\vec{h} = (h_1, ..., h_n)$, we have

$$\begin{array}{rrcl}
 & (F \to G)^e(\vec{h})(f_{D_1...D_n}) & \sqsupseteq & f_{D'_1...D'_n} \\
\Rightarrow & (\lambda k.G^e(\vec{h}) \circ k \circ F^c(\vec{h}))(f_{D_1...D_n}) & \sqsupseteq & f_{D'_1...D'_n} \\
\Rightarrow & G^e(\vec{h}) \circ f_{D_1...D_n} \circ F^c(\vec{h}) & \sqsupseteq & f_{D'_1...D'_n} \\
\Rightarrow & G^e(\vec{h}) \circ f_{D_1...D_n} & \sqsupseteq & f_{D'_1...D'_n} \circ F^e(\vec{h})
\end{array}$$

Therefore, $\{f_{D_1...D_n}\}$ is a lax natural transformation from F to G.

From one of the inequalities above we obtain

$$f_{D'_1 \ldots D'_n} \sqsubseteq G^e(h_1, \ldots, h_n) \circ f_{D_1 \ldots D_n} \circ F^c(h_1, \ldots, h_n)$$

Thus, $G^e(h_1, \ldots, h_n) \circ f_{D_1 \ldots D_n} \circ F^c(h_1, \ldots, h_n)$ is an approximation to $f_{D'_1 \ldots D'_n}$. Since each h_i is a morphism from D_i to D'_i, h_i^e is a one-to-one function. Thus, each D_i is in some sense "smaller" than D'_i. Therefore, the inequality above is a means of obtaining approximations to instances of a polymorphic function from smaller instances.

The language in Chapter 4 does not allow a polymorphic equality function. It was excluded because its semantics fails to be a lax natural transformation. To see this, consider the collection $\{eq_D\}$ of continuous functions $eq_D : D \times D \to Bool$ where

$$eq_D(x, y) = \begin{cases} \bot & \text{if } x = \bot \text{ or } y = \bot \\ True & \text{if } x = y \\ False & \text{if } x \neq y \end{cases}$$

If we were to treat equality as a polymorphic function it would have the type $\forall t.t \times t \to \mathbf{bool}$. To test if the collection is a lax natural transformation, we consider the functors F and G where $F(A) = A \times A$ and $G(A) = Bool$ for all domains A. The actions of these functors on morphisms are not difficult to find out. Now, choose A, B and $h : A \to^{ec} B$ such that h^e is not strict. Clearly,

$$(eq_B \circ (h^e \times h^e))(\bot, \bot) = True \quad \text{and} \quad eq_A(\bot, \bot) = \bot$$

But

$$eq_B \circ (h^e \times h^e) = eq_B \circ F^e(h) \quad \text{and} \quad eq_A = G^e(h) \circ eq_A$$

Thus, the condition

$$eq_B \circ F^e(h) \sqsubseteq G^e(h) \circ eq_A$$

does not hold. Hence, $\{eq_D\}$ is not a lax natural transformation. Not treating equality as a polymorphic function should not be surprising. In fact, it is consistent with the approach followed in the Haskell language [18].

5.4.3 The First-Order Case

We have shown that polytypes can be interpreted as functors and polymorphic functions as lax natural transformations. In the case of the first-order subset of the language, it can be shown that polymorphic functions are natural transformations when one uses the closure components as morphisms—actually strictness is all that is necessary. On the other hand, naturality is not obtained if we use the embedding components as morphisms.

5.5 Semantic Polymorphic Invariance

We have already mentioned that Abramsky and Jensen [5] have used a relational approach to describe the semantics of a polymorphic language. The main result in [5] is the semantic polymorphic invariance of strictness. It is possible to give an alternative proof of this result by using the semantics described in this thesis. To see this, suppose A has a top element so that we are guaranteed to have a morphism $h : \mathbf{2} \rightarrow^{ec} A^{\dagger}$. We can choose h so that h^e is strict. Let f be a lax natural transformation from F to G and $f_{\mathbf{2}}$ be strict. Since h^e is strict so is $G^e(h)$. Now, since

$$f_A \sqsubseteq G^e(h) \circ f_{\mathbf{2}} \circ F^c(h),$$

and $F^c(h)$ is always strict, we have

$$f_A(\bot) = \bot.$$

That is, if a function is strict at the **2**-instance then it is strict at all instances. On the other hand, we need to prove that the strictness of f_A implies that of $f_{\mathbf{2}}$. To do this we need to use embedding-projection pairs.

The Representation Theorem using Embedding-Projection Pairs

Embedding-projection pairs can be used to provide a relational semantics to the language. Except for the difference in the languages, this semantics gives a rep-

[†]It is interesting to note that in [5] also, for the purpose of the proof of invariance, it is assumed that the cpo's used have top elements.

resentation theorem which is essentially Reynolds's theorem of [42]. Polymorphic functions now become lax natural transformations in $\mathcal{C}^{ep}$. A consequence of this is that from the smallest instance of a polymorphic function one can build an approximation from below to any instance. In particular, if f is a lax natural transformation between the functors F and G from $\mathcal{C}^{ep}$ to itself, then for any morphism $k : \mathbf{2} \rightarrow^{ep} A$ we have

$$f_A(\bot) \sqsupseteq G^e(k)(f_2(F^p(k)(\bot)))$$

If f_A is strict then

$$G^e(k)(f_2(F^p(k)(\bot))) = \bot$$

Since $F^p(k)$ is strict and $G^e(k)$ is $\bot$-reflecting, $f_2(\bot) = \bot$. Therefore, this provides the other half of the invariance proof. Thus the combination of the two results gives the semantic polymorphic invariance of strictness.

5.6 Summary

The relational approach has been used in various ways in providing semantics to several languages. The relations which are used in the semantics described in this chapter arise out of embedding-closure pairs. This in turn gives rise to a relationship between different instances. By working in a different category we have managed to overcome the problem which arose out of the contravariance of the function type functor in the category of domains and continuous functions. To a certain extent, therefore, we have a generalisation of Hughes's results on first-order functions, where type constructors are interpreted as functors. In the category of domains and embedding-closure pairs, polymorphic functions are not necessarily natural transformations except in the first-order case. Our primary concern in this thesis is to work with approximations obtainable from the smallest instances of polymorphic functions. However, the combination of results from this semantics, and the one which uses embedding-projection pairs has also given us a proof of the semantic polymorphic invariance of strictness.

Chapter 6

Strictness Analysis

6.1 Introduction

Abstract interpretation was applied in the strictness analysis of a simply typed λ-calculus with higher-order functions over flat domains by Burn et al [11]. The aim here is to show how it can be applied in the strictness analysis of the language defined in Chapter 4. The significance of this language is that it allows polymorphic function definitions.

We have already mentioned that Abramsky proved the polymorphic invariance of strictness analysis [1], and that a proof of the semantic polymorphic invariance of strictness was given by Abramsky and Jensen [5]. However, in practice, invariance is not entirely satisfactory and hence it is necessary to look for other approaches to polymorphic functions. When different instances of the same polymorphic function are used in a program, there is useful information that cannot be obtained from invariance. On the other hand, as pointed out by Abramsky in [1], performing the computation of different instances of the same function as if the instances have no connection between them is inefficient. These issues were discussed in greater detail and an example which highlights the problem was given in Chapter 1. In the case of first-order functions, Hughes proposed a way of obtaining strictness information about any instance of a polymorphic function from that of the simplest [21]. To do this, he first provided a semantics of the language where type constructors are

interpreted as functors on some category of domains, and polymorphic functions are natural transformations.

Here, we use the relational semantics described earlier to obtain, in some sense, a generalisation to higher-order functions of Hughes's results about first-order functions. It is a generalisation because type constructors are interpreted as functors, albeit on a slightly different category of domains. Although not all polymorphic functions are natural transformations, they are lax natural transformations. This provides a way of obtaining strictness information about any instance of a polymorphic function from an analysis of the smallest.

6.2 Abstraction of Types

The abstract interpretation for strictness analysis is done by interpreting every term on a lattice. Before describing how this is done, we introduce the notion of a *lattice type*. Lattice types are types that are defined by the type system in Chapter 4 when restricted to have **2** as the only basic type. To analyse a term in the original language, the type expressions occurring in it are replaced by lattice type expressions, and then the constants are replaced by new appropriate constants. The new term is then evaluated to obtain a value in a lattice. Note that, strictly speaking, there are two type systems (ordinary types and lattice types); for simplicity the same notation is used.

To describe the procedure more formally, first, we define a function mapping ordinary types to lattice types. The function *Abs*, mapping ordinary types to lattice types is defined by induction on the structure of types as follows.

$$
\begin{aligned}
Abs\ \mathbf{int} &= \mathbf{2} \\
Abs\ \mathbf{bool} &= \mathbf{2} \\
Abs\ t &= t \\
Abs\ (\tau_1 \times \tau_2) &= (Abs\ \tau_1) \times (Abs\ \tau_2) \\
Abs\ (\tau_1 \to \tau_2) &= (Abs\ \tau_1) \to (Abs\ \tau_2) \\
Abs\ (\forall t.\sigma) &= \forall t.(Abs\ \sigma)
\end{aligned}
$$

It simply replaces **int** and **bool** by **2** wherever they occur in a type.

6.3 Abstract Functions

The abstract interpretation that will be defined later can be regarded as a (non-standard) semantics. From such an interpretation, information about semantic properties of functions is obtained. In order to show that this information is correct, it is necessary to relate this semantics to the standard semantics.

The semantics of a lattice type is a lattice. For each lattice type τ (which is not a polytype), a lattice B_τ over which terms of this type are interpreted is defined. The symbol **2** is used in two ways; as a lattice type, and the two-element lattice which is the semantics of the type. Lattices corresponding to other types are defined in the same way as the semantics of ordinary types in Chapter 4. It is actually possible to use the same notation, that is, if S is a lattice assignment (in this case, it is a domain assignment associating type variables with finite lattices), $\Im[\![\tau]\!]\ S$ stands for the semantics of the lattice type τ. This may be defined by induction on the structure of τ. Note that $\Im[\![\mathbf{2}]\!]\ S = \mathbf{2}$; the rest is straightforward.

For each monotype τ, the abstraction map abs_τ is defined from the domain corresponding to τ to the lattice corresponding to $Abs\ \tau$. Except when τ is a product of types, these functions are defined in [11]; they are also provided in Chapter 1. Now, for monotypes τ_1 and τ_2, $abs_{\tau_1 \times \tau_2}$ is defined by

$$abs_{\tau_1 \times \tau_2}(a, b) = (abs_{\tau_1} a, abs_{\tau_2} b)$$

The function Abs may also be defined to act on the domains corresponding to monotypes; it is given by

$$\begin{aligned} Abs\ Int &= \mathbf{2} \\ Abs\ Bool &= \mathbf{2} \\ Abs\ (D_1 \times D_2) &= (Abs\ D_1) \times (Abs\ D_2) \\ Abs\ (D_1 \to D_2) &= (Abs\ D_1) \to (Abs\ D_2) \end{aligned}$$

Notice that abs_τ is a function from $\Im[\![\tau]\!]\ S$ to $\Im[\![Abs\ \tau]\!]\ S'$. Since τ does not have variables occurring freely in it, these spaces are independent of S and S'. Several properties of the abstract functions are given in [11].

6.4 Abstract Interpretation

In the notation used here, the abstract interpretation in [11] is done by replacing each type τ by $Abs\ \tau$, integer and boolean constants are replaced by `1`, which is a constant of type **2**. New operators are introduced to replace the usual operators including, for example, the arithmetic ones. The new term obtained when the replacing process is complete is evaluated on lattices to get strictness information.

6.4.1 A Language for Abstract Interpretation

For reference purposes we will use $\mathcal{L}$ to denote the language of concrete terms that was introduced in Chapter 4. Here we shall study a similar language $\mathcal{L}'$ of abstract terms. The difference, as far as types are concerned, is that the only basic type in $\mathcal{L}'$ is **2**. Thus its type system involves only lattice types. Obviously such a language cannot have terms involving integer and boolean constants, and thus nor will it have the conditional expression $\texttt{if}\ \tau\ E_0\ E_1\ E_2$. On the other hand, the following are added to the definition of terms

$$\texttt{guard}\ \tau\ E_0\ E_1 \quad \text{and} \quad \texttt{or}\ \tau\ E_0\ E_1,$$

where τ is a lattice type. It is necessary to make changes to the typing rules as well. The typing rules for $\mathcal{L}'$ are all the rules for $\mathcal{L}$ except COND, the rule involving the typing of $\texttt{if}\ \tau\ E_0\ E_1\ E_2$. Obviously, the typing rules for this and certain constants should be dropped. On the other hand, two new rules are introduced.

$$\text{(AND)} \qquad \frac{\Gamma \vdash E_0 : \mathbf{2} \quad \Gamma \vdash E_1 : \tau}{\Gamma \vdash \mathtt{guard}\ \tau\ E_0\ E_1 : \tau}$$

$$\text{(OR)} \qquad \frac{\Gamma \vdash E_0 : \tau \quad \Gamma \vdash E_1 : \tau}{\Gamma \vdash \mathtt{or}\ \tau\ E_0\ E_1 : \tau}$$

There are also two constants, 0 and 1, in $\mathcal{L}'$ and it is assumed that they have **2** as their type.

6.4.2 Semantics of $\mathcal{L}'$

To define a semantics of $\mathcal{L}'$, we use a category of finite lattices. For a lattice type environment Γ and a lattice type τ, we define the semantic functions $\nu_{\Gamma,\tau}$ in a similar way as the semantic functions (the μ's of the previous chapter) for the language $\mathcal{L}$. Because of the similarity in the definitions of the semantic functions for both languages, we consider only the new terms; they are of the form $\mathtt{guard}\ \tau\ E_0\ E_1$ and $\mathtt{or}\ \tau\ E_0\ E_1$. The family of functions $\{\rhd_A\}$ and $\{\sqcup_A\}$ were introduced in Chapter 3 for this purpose. Now, using the infix notation, the definitions are given by

$$\nu_{\Gamma,\tau}[\![\mathtt{guard}\ \tau\ E_0\ E_1]\!]\ S\ \eta \;=\; \nu_{\Gamma,\mathbf{2}}[\![E_0]\!]\ S\ \eta \ \rhd_A\ \nu_{\Gamma,\tau}[\![E_1]\!]\ S\ \eta$$

and

$$\nu_{\Gamma,\tau}[\![\mathtt{or}\ \tau\ E_0\ E_1]\!]\ S\ \eta \;=\; \nu_{\Gamma,\tau}[\![E_0]\!]\ S\ \eta \ \sqcup_A\ \nu_{\Gamma,\tau}[\![E_1]\!]\ S\ \eta$$

where $A = \Im[\![\tau]\!]\ S$.

We can now use the results of the investigation of the semantics of $\mathcal{L}$ in Chapter 5 for $\mathcal{L}'$. Several properties of the semantic functions of $\mathcal{L}$ hold for the semantic functions of $\mathcal{L}'$. In particular, it is not difficult to show that the representation theorem holds. Observe that the only new collections are $\{\rhd_A\}$ and $\{\sqcup_A\}$, and it was already shown in Chapter 3 that they are both lax natural transformations. Therefore, all polymorphic functions in $\mathcal{L}'$ are lax natural transformations between functors on the category of finite lattices and embedding-closure pairs.

Polymorphic first-order functions are actually natural transformations between functors on the category of finite lattices which has the closure components of embedding-closure pairs as morphisms. Recall that this is not the case when the embedding components are used.

The fact that the representation theorem holds allows us to relate separate instances of abstract polymorphic functions, and deduce facts about all instances by studying just one.

6.4.3 Connection between $\mathcal{L}$ and $\mathcal{L}'$

We prefer to work with two languages for reasons of uniformity in the semantics, that is, in order that all polymorphic functions are lax natural transformations. If we had a single language containing terms like $\mathtt{or}\ \tau\ E_0\ E_1$ whose semantics is given in terms of the collection $\{\sqcup_A\}$, then the collection cannot be a lax natural transformation between functors on the category of domains and embedding-closure pairs. This is because $\sqcup_A$ is not defined for arbitrary domains A (non-lattices).

To perform strictness analysis on a term E in $\mathcal{L}$, first, it is translated to a term E' in $\mathcal{L}'$ and then the new term is interpreted over finite lattices. The interpretation is simply to evaluate E' using the semantics described above. The translation is done by replacing every type τ, occurring in the expression, by $Abs\ \tau$, and then integer and boolean constants are replaced by the appropriate terms of type **2**. Any term of the form

$$\mathtt{if}\ \tau\ E_0\ E_1\ E_2$$

is translated to

$$\mathtt{guard}\ \tau'\ E_0'\ (\mathtt{or}\ \tau'\ E_1'\ E_2'),$$

where $\tau' = Abs\ \tau$ and each E_i' is the result of translating E_i. Also every term of the form

$$E_1 \oplus E_2$$

is translated to

$$\mathtt{guard}\ \mathbf{2}\ E_1'\ E_2',$$

where $\oplus$ is any one of the arithmetic operators : +, -, * or / (note that the semantics of guard **2** is $\sqcap_2$). In fact this should be done for all built-in binary operators that are strict in both their arguments. The translation of other terms is straightforward.

Suppose Γ is the type environment $\{x_1 : \sigma_1, \ldots x_n : \sigma_n\}$. If the formula

$$\Gamma \vdash E : \tau$$

can be obtained from the typing rules of $\mathcal{L}$, then

$$\Gamma' \vdash E' : Abs\ \tau$$

can be obtained from those of $\mathcal{L}'$, where

$$\Gamma' = \{x_1 : Abs\ \sigma_1, \ldots x_n : Abs\ \sigma_n\},$$

by straightforward substitution in the derivation of $\Gamma \vdash E : \tau$. Now, the standard semantics is given by

$$\mu_{\Gamma,\tau}[\![E]\!]\ S\ \eta,$$

and the non-standard semantics of its translation is given by

$$\nu_{\Gamma', Abs\ \tau}[\![E']\!]\ S'\ \eta',$$

where S' is a lattice environment, *i.e*, an assignment of lattices to type variables, and η' is an assignment of values from lattices to ordinary variables. Here, $S'(t) = Abs\ S(t)$ and $\eta'(x) = abs(\eta(x))$.

Notation. The smallest family of domains containing *Bool* and *Int*, and that is closed under both $\times$ and $\rightarrow$ will be denoted by $\mathcal{D}$. Similarly, $\mathcal{B}$ will denote the smallest family of finite lattices that contains **2**, and that is also closed under $\times$ and $\rightarrow$.

6.5 Safety

In a previous section abstract functions were introduced. The abstract functions relate the semantics of terms in the language $\mathcal{L}$ and that of their translation in $\mathcal{L}'$. The following proposition describes the relationship.

Proposition 6.5.1 *Suppose we have the formula* $\Gamma \vdash E : \tau$, *and that* S *is a domain assignment that assigns type variables domains from* $\mathcal{D}$. *Let* η *be an environment. Let* S' *be the lattice assignment such that for any type variable* t, $S'(t) = Abs\ S(t)$.

If η' *is any environment such that for any variable* $x \in dom(\Gamma)$

$$abs_{\Gamma(x)}(\eta(x)) \sqsubseteq \eta'(x),$$

then

$$abs_\tau(\mu_{\Gamma,\tau}[\![E]\!]\ S\ \eta) \sqsubseteq \nu_{\Gamma', Abs\,\tau}[\![E']\!]\ S'\ \eta'$$

The proof is given in the Appendix.

From this proposition it is not difficult to show that the information relating to the strictness of functions obtained from the non-standard semantics is correct. Moreover, for any monomorphic term E, the semantics of E', given by the semantic functions for $\mathcal{L}'$, is the same as the value of the function *asem* at E (the definition of *asem* is given in Chapter 1).

6.6 Polymorphic Functions

We have now provided the definition of the ν's on terms with types from the first class (that is, not polytypes). The definition is similar to that of μ. The semantics of a polymorphic function is the collection of the semantics of all its monomorphic instances. From what was shown in the previous chapter, these instances are related in a certain way.

Suppose $\Lambda\vec{t}.E$ is a closed polymorphic term in $\mathcal{L}$ whose type is $\forall\vec{t}.F(\vec{t}) \to G(\vec{t})$. The term can be translated into $\Lambda\vec{t}.E'$ in $\mathcal{L}'$ and it has the type $\forall\vec{t}.F'(\vec{t}) \to G'(\vec{t})$, where $F'(\vec{t})$ and $G'(\vec{t})$ are obtained by replacing the types **int** and **bool** in $F(t)$ and $G(t)$ by **2**. Now, both F' and G' can be interpreted as functors from $(\mathcal{A}^{ec})^n$ to $\mathcal{A}^{ec}$. Moreover, the semantics of this new term, $\Lambda\vec{t}.E'$, is a lax natural transformation from F' to G'. Normally, we will drop the superscripts and write F and G for F' and G'.

Suppose $\tau_1, ..., \tau_n, \tau'_1, ..., \tau'_n$ are monotypes. For $i = 1, ..., n$, let B_i and B'_i denote the finite lattices which are the semantics of $Abs\ \tau_i$ and $Abs\ \tau'_i$ respectively. Suppose also that for each i, $h_i : B_i \rightarrow^{ec} B'_i$ is any embedding-closure pair, and $\vec{h} = (h_1, ..., h_n)$. Then

$$\begin{aligned}&\nu[\![(\Lambda\vec{t}.E')[Abs\ \tau'_1]...[Abs\ \tau'_n]]\!] \\ &\quad \sqsubseteq\ G^e(\vec{h}) \circ \nu[\![(\Lambda\vec{t}.E')[Abs\ \tau_1]...[Abs\ \tau_n]]\!] \circ F^c(\vec{h})\end{aligned}$$

Since $\Lambda\vec{t}.E'$ is closed, the value $\nu[\![\Lambda\vec{t}.E']\!]\ S\ \eta$ is independent of lattice assignments S and environments η. That is why they are omitted here. Also, the usual subscripts that come with ν are omitted.

In general, the semantics of $\Lambda\vec{t}.E'$ may be expressed as the collection $\{g^{\mathcal{S}}_{B_1...B_n}\}$, where the B_i's range over all finite lattices and

$$g^{\mathcal{S}}_{B_1...B_n} = \nu[\![E']\!]\ S\ \eta$$

where $S(t_i) = B_i$ for each i $(1 \leq i \leq n)$. The superscript $\mathcal{S}$ (for strictness) is used to distinguish this from other non-standard semantics.

If each h_i is a morphism from $\mathbf{2}$ to B_i, then

$$g^{\mathcal{S}}_{B_1...B_n} \ \sqsubseteq\ G^e(\vec{h}) \circ g^{\mathcal{S}}_{\mathbf{2}...\mathbf{2}} \circ F^c(\vec{h})$$

Therefore, we obtain an approximate value for $g^{\mathcal{S}}_{B_1...B_n}$ from $g^{\mathcal{S}}_{\mathbf{2}...\mathbf{2}}$. The significance of this is that the latter is normally less expensive to compute.

This particular approximation is from above, and is thus appropriate for strictness analysis. Using $G^e(h_1, ..., h_n) \circ g^{\mathcal{S}}_{\mathbf{2}...\mathbf{2}} \circ F^c(h_1, ..., h_n)$ instead of $g^{\mathcal{S}}_{B_1...B_n}$ in the analysis never leads to any incorrect strictness information. This is the central reason why we chose to use embedding-closure pairs in describing the semantics of the language.

At this point it is legitimate to ask how good the approximation is. We will assume that $n = 1$ for simplicity in presentation. Recall that if $h : \mathbf{2} \rightarrow^{ec} B$ is chosen so that h^e is strict, then whenever $g^{\mathcal{S}}_{\mathbf{2}}$ is strict so is $G^e(h) \circ g^{\mathcal{S}}_{\mathbf{2}} \circ F^c(h)$. But if h^e is not strict, the strictness of $g^{\mathcal{S}}_{\mathbf{2}}$ does not imply that of $G^e(h) \circ g^{\mathcal{S}}_{\mathbf{2}} \circ F^c(h)$. In general, although these new functions may be used instead of $g^{\mathcal{S}}_B$, they may not be very good

approximations. To obtain a better result, we take the greatest lower bound of all possible approximations. This is because the following holds

$$g_B^{\mathcal{S}} \sqsubseteq \bigsqcap G^e(h) \circ g_2^{\mathcal{S}} \circ F^c(h)$$

The greatest lower bound is taken over all possible h's.

There are many cases where computing a single $G^e(h) \circ g_2^{\mathcal{S}} \circ F^c(h)$ is significantly cheaper than computing $g_B^{\mathcal{S}}$. There are $(|B| - 1)$ embedding-closure pairs from $\mathbf{2}$ to B, where $|B|$ is the number of elements of B. As a result, if B is a big lattice then computing the greatest lower bound may be expensive. As was shown in Chapter 3, there are situations where a smaller number of embedding-closure pairs is enough to obtain the same approximation. This is particularly the case with first-order functions; recall the use of meet-irreducible elements of B. Moreover, the greatest lower bound of the functions in the first-order case is exactly $g_B^{\mathcal{S}}$, *i.e*,

$$g_B^{\mathcal{S}} = \bigsqcap G^e(h) \circ g_2^{\mathcal{S}} \circ F^c(h)$$

This follows from the fact that polymorphic first-order functions are natural transformations when using the closure components as morphisms, and that $\bigsqcap G^e(h) \circ G^c(h) = id$ for all first-order type functors (recall that the details were given Chapter 3).

Example

To illustrate the ideas discussed above, let us consider the following simple example.

$$f = \Lambda\alpha.\mathtt{fix}\ \tau\ (\lambda g : \tau.\lambda x : \mathbf{bool}.\lambda y : \mathbf{int}.\lambda z : \alpha.\ \mathtt{if}\ \ \alpha\ x\ z\ (g\ x\ (y - 1)\ z))$$

where $\tau = \mathbf{bool} \to \mathbf{int} \to \alpha \to \alpha$. Notice that α is a type variable.

If the abstract interpretation of f is given to be $f^{\mathcal{S}}$, then its value at the simplest type, *i.e* $f_2^{\mathcal{S}}$, is given by the following table

x	y	z	$f_2^{\mathcal{S}}\ x\ y\ z$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

This table may be obtained by iterative means. Now, if $f_{\mathbf{2}\times\mathbf{2}}^{\mathcal{S}}$, the abstract interpretation of the function at the type $\mathbf{2} \times \mathbf{2}$ has to be computed, then a number of iterations must again be performed. But performing such computations every time we consider an instance of the same polymorphic function is very expensive. The method described above provides an approximation to $f_{\mathbf{2}\times\mathbf{2}}^{\mathcal{S}}$ by using $f_{\mathbf{2}}^{\mathcal{S}}$. That is,

$$f_{\mathbf{2}\times\mathbf{2}}^{\mathcal{S}} \sqsubseteq \bigsqcap_a (\lambda u.(\lambda l.h_a^e \circ l \circ h_a^c) \circ u \circ id^c) \circ f_{\mathbf{2}}^{\mathcal{S}} \circ id^c$$

The a's range over all non-$\top$ elements of $\mathbf{2} \times \mathbf{2}$, and thus the h_a's are the embedding-closure pairs from $\mathbf{2}$ to $\mathbf{2} \times \mathbf{2}$. Note that id^c is the identity function, where id is the identity embedding-closure pair. In this example, the greatest lower bound may be taken over two a's to obtain the same result. To see this, for any non-$\top$ element a of $\mathbf{2} \times \mathbf{2}$ let

$$k_a = (\lambda u.(\lambda l.h_a^e \circ l \circ h_a^c) \circ u \circ id^c) \circ f_{\mathbf{2}}^{\mathcal{S}} \circ id^c$$

Now, for any x, y, and z we have

$$\begin{aligned} k_a\ x\ y\ z &= (\lambda u.(\lambda l.h_a^e \circ l \circ h_a^c) \circ u \circ id^c) \circ f_{\mathbf{2}}^{\mathcal{S}} \circ id^c\ x\ y\ z \\ &= (\lambda u.(\lambda l.h_a^e \circ l \circ h_a^c) \circ u \circ id^c)(f_{\mathbf{2}}^{\mathcal{S}}\ x)\ y\ z \\ &= (\lambda l.h_a^e \circ l \circ h_a^c)(f_{\mathbf{2}}^{\mathcal{S}}\ x\ y)\ z \\ &= (h_a^e \circ (f_{\mathbf{2}}^{\mathcal{S}}\ x\ y) \circ h_a^c)\ z \\ &= h_a^e(f_{\mathbf{2}}^{\mathcal{S}}\ x\ y\ (h_a^c\ z)) \end{aligned}$$

From this and the definition of the h_a's it is not difficult to build the following table.

z	$k_{(0,0)}\ x\ y\ z$	$k_{(0,1)}\ x\ y\ z$	$k_{(1,0)}\ x\ y\ z$
$(0,0)$	$(0,0)$	$(0,1)$	$(1,0)$
$(0,1)$	$h^e_{(0,0)}(x)$	$(0,1)$	$h^e_{(1,0)}(x)$
$(1,0)$	$h^e_{(0,0)}(x)$	$h^e_{(0,1)}(x)$	$(1,0)$
$(1,1)$	$h^e_{(0,0)}(x)$	$h^e_{(0,1)}(x)$	$h^e_{(1,0)}(x)$

It is now easy to see that

$$k_{(0,1)} \sqcap k_{(1,0)} \sqsubseteq k_{(0,0)}$$

Therefore, when taking the greatest lower bound, $k_{(0,0)}$ can be ignored. Moreover, in this particular case the result obtained this way gives $f^{\mathcal{S}}_{\mathbf{2\times 2}}$ exactly. That is,

$$f^{\mathcal{S}}_{\mathbf{2\times 2}} = k_{(0,1)} \sqcap k_{(1,0)}$$

Therefore, in this example no price, in terms of accuracy, is paid by resorting to the method which normally gives approximate values.

In general, whenever B is large it may be necessary to consider the greatest lower bound of only a smaller number of functions. In the first-order case restricting to morphisms h_a, where a is meet-irreducible gives the exact result. It may be a good idea to use this in the general case as well. Obviously, this in turn is an approximation of the greatest lower bound. For exact values one has to compute $f^{\mathcal{S}}_B$ directly. But this is exactly what we wanted to avoid in the first place. However, in the case of non-recursive functions it is better to compute all the necessary instances directly, because it is efficient and we get exact values.

6.7 Monomorphic Functions

So far, we have shown that a useful approximation to the abstract interpretation of an instance of a polymorphic function may be found from that of the smallest. However, there are monomorphic functions that are not instances of some polymorphic function. The computation of the abstract interpretation of such functions may be expensive; this normally involves finding fixed points.

To deal with such problems, Hunt introduced approximate fixed ponts [27]. Suppose B is a large lattice and $f : B \to B$ is a continuous function. He proposes to pick

a lattice A "smaller" than B, and then define a function $g : A \to A$ that in some sense "represents" f. Since A is smaller, computing the fixed point of g is hopefully less expensive than that of f. From this, an approximation to the fixed point of f is obtained. The function f and g are related by an inequality of the form

$$f \sqsubseteq h_2 \circ g \circ h_1,$$

where $h_1 : B \to A$ and $h_2 : A \to B$ are some functions which are defined by induction on the structure of the lattices A and B. Seward observed that (h_1, h_2) forms a special embedding-closure pair from A to B, and suggested the use of more embedding-closure pairs so that a better approximation may be obtained [47]. When A is the lattice $\mathbf{2}$, the embedding-closure pair used by Hunt is $h_\perp$, in our notation of Chapter 3.

Although the function f is not necessarily an instance of a polymorphic function, the procedure above is related to the analysis of polymorphic functions. Recall that fix is a lax natural transformation. Now, if $h : A \to^{ec} B$ is an embedding-closure pair then

$$fix_B \sqsubseteq h^e \circ fix_A \circ (\lambda k.h^c \circ k \circ h^e)$$

Hence,

$$fix_B(f) \sqsubseteq h^e(fix_A(h^c \circ f \circ h^e))$$

Since $h^c \circ f \circ h^e$ is a function from A to A, if A is small then computing $h^e(fix_A(h^c \circ f \circ h^e))$ is likely to be less expensive. To obtain a better approximation, it is possible to consider the greatest lower bound of several functions of this form. If in the above inequality, A is the lattice $\mathbf{2}$ then

$$fix_B(f) \sqsubseteq \bigsqcap h_a^e(fix_{\mathbf{2}}(h_a^c \circ f \circ h_a^e)),$$

where a ranges over the non-$\top$ elements of B. It is not difficult to show that

$$\bigsqcap h_a^e(fix_{\mathbf{2}}(h_a^c \circ f \circ h_a^e)) = \bigsqcap \{a \in (B - \{\top\}) \mid f(a) \sqsubseteq a\}$$

Observe that, in the verification of this statement, for any function $k : \mathbf{2} \to \mathbf{2}$ it is always the case that

$$fix_{\mathbf{2}}(k) = k(0)$$

Hence,

$$\begin{aligned} h_a^e(fix_2(h_a^c \circ f \circ h_a^e)) &= h_a^e(h_a^c(f(h_a^e(0)))) \\ &= h_a^e(h_a^c(f(a))) \end{aligned}$$

On the other hand, one of Tarski's descriptions of the least fixed point of f is given as

$$\bigsqcap\{a \in B \mid f(a) \sqsubseteq a\}$$

in [49]. This expression is the same as the right hand side of the above inequality, when it is assumed that $\bigsqcap\{\} = \top$. Therefore,

$$fix_B(f) = \bigsqcap h_a^e(fix_2(h_a^c \circ f \circ h_a^e))$$

It is important to note that this is a statement about accuracy. An approximate value may be obtained by using only a few functions. In any case, the preceding discussion presents an improvement on Hunt's proposal because of its use of the greatest lower bound. In fact, using his method gives the approximation

$$h_\perp^e(fix_2(h_\perp^c \circ f \circ h_\perp^e))$$

Since $h_\perp^e$ is a two-valued function, this expression is equal to $\perp_B$ or to $\top_B$ and hence, in some cases, is not a good approximation to $fix_B(f)$. If the expression is $\perp_B$ then $fix_B(f)$ is also $\perp_B$, which is the exact value. Otherwise, it is $\top_B$ and this does not carry any information at all, because we already know that every element of B is bounded by $\top_B$ from above. It must be said, however, that Hunt's suggestion does not always imply the use of **2**. In particular, if for example B is the product $B_1 \times B_2$, then a lattice of the form $A_1 \times A_2$ where each A_i is smaller than B_i is used. Therefore, an approximation better than the one which results from the use of **2** is obtainable even when only a single embedding-closure pair is used.

6.8 Summary

The semantic approach to polymorphic functions, which we have developed, has provided a way of using the result of an analysis of simple instances to obtain information about more complex instances. Seward has implemented this method, giving

impressive performance figures [48]. He discusses these results and the usefulness of the information so obtained; we will see concrete examples in the next chapter.

Chapter 7

Analysis of Lists

7.1 Introduction

The language we have been investigating so far does not have some of the common structure types. In this chapter, we consider the case of lists. In the monomorphic case, Wadler defined an abstract interpretation that detects some semantic properties of functions defined on lists [52]. He introduced abstract domains appropriate for doing this interpretation.

Our main concern here is polymorphism. We extend our language to include lists, and establish a relationship between the abstract interpretation of different instances of polymorphic functions. Hence, as in Chapter 6, we have a method of using the abstract interpretation of the smallest instance in other instances.

Unfortunately, we are unable to prove the correctness of our method when using Wadler's abstract domains; though we could not find counter examples either. F. Nielson and H.R. Nielson in their work on generalising Wadler's method have used different abstract domains [38]. We use these domains in our study of polymorphic functions.

One of the results that we proved in Chapter 5 is the semantic polymorphic invariance of strictness. We have already remarked that this is not a new result;

Abramsky and Jensen have already given a proof in [5]. Our proof was given simply to demonstrate that this property follows also from the semantics which involves embedding-closure pairs. As explained in Chapter 1, in the monomorphic case, Wadler's analysis detects different kinds of strictness of functions defined on lists. Consequently, in the case of polymorphic functions, it may be desirable to investigate the semantic polymorphic invariance of such properties. This would generalise the work of Abramsky and Jensen [5]. This will be done in Section 6.

7.2 Strictness Properties

We consider functions defined in terms of the **case**-statement. First, we define

```
case(a,f,nil)       =  a
case(a,f,cons x xs) =  f(x,xs)
```

The type of this function is $\forall t.\forall s.s \times ((t \times [t]) \to s) \times [t] \to s$. We investigate the semantic properties of the function **g** which is given by

```
g ys = case(a,f,ys)
```

for some fixed **a** and **f**. For the purposes of the discussion in this and the next section, we assume that **case** and **g** are monomorphic. Suppose *case* and g are their respective semantics, where

$$\begin{aligned} case(a, f, \bot) &= \bot \\ case(a, f, []) &= a \\ case(a, f, cons\ x\ xs) &= f(x, xs) \end{aligned}$$

and

$$g(ys) = case(a, f, ys)$$

Among the properties studied by Wadler are tail-strictness and hyper-strictness. The function g is said to be

(i) tail-strict, if $g(xs) = \bot$ for all lists xs which are infinite or partial, and

(ii) hyper-strict, if $g(xs) = \bot$ for every list xs which is not "total". We say that an element a of a domain is *total* if it is maximal, *i.e* if there is no b with $a \sqsubset b$, and is finite (in the domain-theoretic sense). In particular, if zs is a list which is total then it must be of the form $[z_1, ..., z_n]$ for some $n \geq 0$, where each z_i is total.

In the case of a list of integers, to say that it is total means that it is finite and does not contain $\bot$ as a member. However, for lists of more complex types, the two definitions are not equivalent. Consequently, for example, if h is a function defined on lists of lists of integers and is hyper-strict then we must have $h([[8], [2, \bot, 12]]) = \bot$, although the list does not contain $\bot$ as one of its members.

To detect such properties, Wadler defined an abstract interpretation. One way of looking at the abstract domain used by Wadler in the analysis of lists of elements of some type τ is as a domain obtained by double-lifting the abstract domain used in the analysis of expressions of type τ. For example, if $\tau = \mathbf{int}$ then his abstract domain, $\{\bot, \infty, \bot_\in, \top_\in\}$, may be seen to be the same as $(\mathbf{2}_\bot)_\bot$. Recall that the abstract function $cons^S$ of the operation **cons** was defined as follows. For any $x, y \in \mathbf{2}$

$$\begin{aligned} cons^S\ x\ \bot &= \infty \\ cons^S\ x\ \infty &= \infty \\ cons^S\ x\ y_\in &= (x \sqcap y)_\in \end{aligned}$$

It is necessary to add that $nil^S = \top_\in$. In general for any finite lattice A, we can define $cons^S_A : A \times (A_\bot)_\bot \to (A_\bot)_\bot$ in this way.

Recall that our ultimate aim, as in Chapter 6, is to somehow relate the abstract interpretation of different instances of polymorphic functions. Unfortunately, with this choice of abstract domains and definition of $cons^S_A$, we are unable to prove that the desired relationship holds. We overcome this difficulty by using the abstract domains used by Nielson and Nielson [38]. In explaining it in the next section, we will mainly concentrate on the monomorphic case.

7.3 Abstract Interpretation

7.3.1 Nielson and Nielson's Approach

If A is the abstract domain corresponding to the type τ, Nielson and Nielson use certain subsets of A as elements of the abstract domain for the type of lists whose members are of type τ [38]. We need to give a number of definitions before introducing these subsets. Except for some difference in notation, these definitions are to be found in [38].

Let A be a finite lattice. Recall that any subset $X \subseteq A$ is said to be open if X is upwards closed; that is, X is open in the Scott topology induced by the lattice structure on A. Let $\mathcal{O}(A)$ denote the set of all non-empty open subsets of A, that is

$$\mathcal{O}(A) = \{X^{\uparrow} \mid X \subseteq A \text{ and } X \neq \emptyset\},$$

where $X^{\uparrow}$ is the upward closure of X.

The relation $\supseteq$ makes $\mathcal{O}(A)$ a lattice in which A is the smallest element and $\{\top\}$ is the greatest element, where $\top$ is the greatest element of A. The lattice operations $\sqcap$ and $\sqcup$ are respectively given by the set-theoretic operations $\cup$ and $\cap$.

We now come to the set of points used in the abstract interpretation of lists whose members use A in their abstract interpretation. Define $L(A)$ by

$$L(A) = \{\bot, \infty\} \cup \{X_{\in} \mid X \in \mathcal{O}(A)\}$$

Another way of looking at this set is as $(\mathcal{O}(A)_{\bot})_{\bot}$. To make this a lattice, first an ordering $\sqsubseteq$ is defined on $L(A)$. To do this let Z, $X_{\in}$ and $Y_{\in}$ be any elements of this set. Then the ordering is given by

$$\begin{aligned} \bot &\sqsubseteq Z \\ \infty &\sqsubseteq Z \qquad \text{if } Z \neq \bot \\ X_{\in} &\sqsubseteq Y_{\in} \qquad \text{if } X \sqsubseteq Y \text{ in } \mathcal{O}(A) \end{aligned}$$

The last statement is equivalent to saying that $X \supseteq Y$.

It is now necessary to provide an interpretation to the elements of $L(A)$. That is,

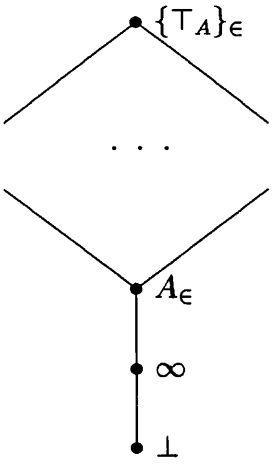


Figure 7.1: The Lattice $L(A)$

what does it mean for a list xs to have the property represented by an element of $L(A)$?

- $\bot$ describes the list $\bot$.
- ∞ describes all infinite lists, all partial lists and $\bot$.
- If $X = \{a_1, a_2, ..., a_k\}$ is an open set, then $X_\in$ describes all lists described by $\bot$ and also those described by ∞, and all non-empty finite lists $[x_1, x_2, ..., x_n]$ in which each a_i describes some x_j. (NB : two distinct a_i may describe some x_j.)
- $\{\top\}_\in$ describes all lists. (the empty list is described only by this point.)

Later we will express this more clearly by defining the mapping *abs* which relates the standard semantics with the abstract semantics.

7.3.2 Comparison of $L(A)$ with Wadler's Domains

Since **2** is used in the abstract interpretation of integers, $L(\mathbf{2})$ is used in that of lists of integers. Now, $\{1\}_\in$ describes all lists, whereas $\{0,1\}_\in$ describes all except finite lists which do not have $\bot$ as a member. Therefore, in this example Wadler's domain

and $L(\mathbf{2})$ may be regarded as the same by identifying $\top_\in$ with $\{1\}_\in$ and $\bot_\in$ with $\{0,1\}_\in$.

One of the differences between the points of Wadler's abstract domains and those which we just described is easily observed when lists of pairs of integers are considered. Now, if $(0,0)_\in$ describes the finite list $[x_1, x_2, ..., x_n]$, it is not necessarily the case that there is some x_j described by $(0,0)$. For example, the list $[(4,\bot),(\bot,5)]$ is described by $(0,0)_\in$ using Wadler's definition, but none of its members is described by $(0,0)$. Clearly, as was pointed out, this is not the case when the abstract domains which involve open sets are used. The point $\{(0,0)\}^\uparrow_\in$ does not describe this list because the list does not have a member describable by $(0,0)$. The smallest point which describes this list is $\{(0,1),(1,0)\}^\uparrow_\in$. Notice also that in this example Wadler's domain has six elements whereas the one used by Nielson and Nielson has seven.

In general, whenever A is a chain, $L(A)$ is also a chain with two more elements. Therefore, for the many examples considered by Wadler in [52], the lattices whose definitions were just outlined are the same (up to isomorphism) as Wadler's. But whenever the operator $\times$ is involved, two different abstract domains are obtained.

7.3.3 Some Operations on Lists

The abstract interpretation of some operations on lists will be defined. For any finite lattice A, define $cons^{\mathcal{S}}_A : A \to L(A) \to L(A)$ by

$$\begin{aligned} cons^{\mathcal{S}}_A\ x\ \bot &= \infty \\ cons^{\mathcal{S}}_A\ x\ \infty &= \infty \\ cons^{\mathcal{S}}_A\ x\ X_\in &= (\{x\}^\uparrow \sqcap X)_\in \end{aligned}$$

where $\sqcap$ is the set union $\cup$. Also, let $nil^{\mathcal{S}}_A = \{\top_A\}_\in$; recall that this is the greatest element of $L(A)$. It is easy to show that $cons^{\mathcal{S}}_A$ is continuous.

In this section, recall that only monomorphic functions are considered. Suppose A and B are finite lattices and b in B, and also that $f : A \times L(A) \to B$ is a continuous function. Define $case^{\mathcal{S}}$, the abstract interpretation of `case`, by

$$
\begin{aligned}
case^{\mathcal{S}}(b,f,\bot) &= \bot \\
case^{\mathcal{S}}(b,f,\infty) &= f(\top,\infty) \\
case^{\mathcal{S}}(b,f,\{\top\}_\in) &= b \sqcup f(\top,\{\top\}_\in) \\
case^{\mathcal{S}}(b,f,Y_\in) &= \bigsqcup_{Y' \in \mathcal{U}_Y} f(\sqcap Y', (Y \ominus Y')_\in), \quad \text{if } Y \neq \{\top\}
\end{aligned}
$$

where $\mathcal{U}_Y = \{Y' \mid Y' \subseteq Y,\ Y' \text{ open and non-empty}\}$, and for any set X, $Y \ominus X = (Y - X^\uparrow)^\uparrow \cup \{\top\}$. In the fourth case of the definition, it is assumed that $Y \neq \{\top\}$ because it has been treated in the third case.

As described in [38], the motivation to the last equation in the definition of $case^{\mathcal{S}}$ is as follows. If $Y_\in$ ($Y \neq \{\top\}$) describes the list xs then we may assume that the list is of the form $cons\ x\ xs'$. Now, if the open set $Y' \subseteq Y$ consists of the points which describe x then $\sqcap Y'$ also describes x. This is because, if $abs(x) \sqsubseteq y$ for every $y \in Y'$ then $abs(x) \sqsubseteq \sqcap Y'$. We may assume that Y' is non-empty because $\top$ describes x. If $y \in Y - Y'$ then by definition of Y', y does not describe x. But $Y_\in$ describes $cons\ x\ xs'$, and hence y must describe some member of xs'. Therefore, $(Y \ominus Y')_\in$ describes xs'; more details are given in [38].

To show that $case^{\mathcal{S}}$ is continuous we consider the last case only; the other cases are very easy to check. Suppose $X_\in \sqsubseteq Y_\in$. It is necessary to show that

$$case^{\mathcal{S}}(b,f,X_\in) \sqsubseteq case^{\mathcal{S}}(b,f,Y_\in)$$

By assumption $Y \subseteq X$. Let U be any non-empty open subset of X. Then

$$(Y - U) \subseteq (X - U)$$

Consider the non-empty open subset V_U of Y, which is defined by $V_U = Y \cap U$; it is non-empty because both Y and U contain at least $\top$. Observe that

$$(X - U) \supseteq (Y - U) = (Y - V_U)$$

Hence,

$$(X \ominus U)_\in \sqsubseteq (Y \ominus V_U)_\in$$

Also,

$$\sqcap U \sqsubseteq \sqcap V_U$$

Thus,

$$f(\sqcap U, (X \ominus U)_\in) \sqsubseteq f(\sqcap V_U, (Y \ominus V_U)_\in)$$

What is shown is that for any expression in the set whose least upper bound gives the left hand side of the inequality which we want to prove, there is an expression

in the corresponding set for the right hand side which dominates it. Now, it is easy to observe that

$$\begin{aligned} case^{\mathcal{S}}(b, f, X_{\in}) &= \bigsqcup_{U \in \mathcal{U}_X} f(\sqcap U, (X \ominus U)_{\in}) \\ &\sqsubseteq \bigsqcup_{U \in \mathcal{U}_X} f(\sqcap V_U, (Y \ominus V_U)_{\in}) \\ &\sqsubseteq \bigsqcup_{V \in \mathcal{U}_Y} f(\sqcap V, (Y \ominus V)_{\in}) \\ &= case^{\mathcal{S}}(b, f, Y_{\in}) \end{aligned}$$

Therefore, $case^{\mathcal{S}}$ is continuous.

7.3.4 Strictness Analysis and Safety

Among the information that can be obtained from the analysis described above, we concentrate on tail-strictness and hyper-strictness. Consider the function g which was defined earlier in terms of a, f and case. Let A and B be the finite abstract domains which correspond to the semantic domains D and E respectively. Now, $g^{\mathcal{S}} : L(A) \to B$ is given by

$$g^{\mathcal{S}}(Z) = case^{\mathcal{S}}(a^{\mathcal{S}}, f^{\mathcal{S}}, Z)$$

and this function is continuous because $case^{\mathcal{S}}$ is. On the other hand, if $List(D)$ is the domain of all lists whose members are elements of D, then $g : List(D) \to E$, the semantics of g, is defined by

$$g(ys) = case(a, f, ys)$$

where a and f are the respective semantics of a and f. The properties of interest are

- If $g^{\mathcal{S}}(\infty) = \bot$ then g is tail-strict.
- If $g^{\mathcal{S}}(X_{\in}) = \bot$ for all $X \neq \{\top\}$, then g is hyper-strict.

At this point, it is necessary to prove that these statements are correct. In order to give a BHA-style proof, we extend the definition of *abs*. The new case here is that of lists, otherwise the definition of *abs* is as given in [11] (see also Chapter 1). If D

is a semantic domain whose corresponding abstract domain is A, in the notation of the previous chapter $Abs\ D = A$, then the abstract mapping $abs : List(D) \to L(A)$ is given by

$$abs(xs) = \begin{cases} \bot & \text{if } xs = \bot \\ \{\top\}_\in & \text{if } xs = [] \\ \infty & \text{if } xs \text{ is infinite or a partial list} \\ \{abs(x_i) \mid 1 \leq i \leq n\}^\uparrow_\in & \text{if } xs = [x_1, ..., x_n] \end{cases}$$

The abs in the right hand side in the fourth case of this definition is the one that relates D and A. Strictly speaking abs should be indexed by the type which is interpreted by its argument domain. Now, it is easy to see that abs is continuous.

With this extension of the definition of abs, the following two properties still hold.

$$\begin{aligned} abs(h(x)) &\sqsubseteq abs(h)(abs(x)) \\ abs(x) = \bot &\Rightarrow x = \bot \end{aligned}$$

Moreover,

$$abs(xs) \neq \{\top\}_\in \text{ if and only if } xs \text{ is not total.}$$

Since essentially open sets are being used as elements of abstract domains, the significant adjustment which has to be made in the definition of abs is seen in the fourth case of the definition above. Recall that Wadler's original definition is given by

$$abs([x_1, ..., x_n]) = (\sqcap\{abs(x_1), ..., abs(x_n)\})_\in$$

We need the following statement to prove the correctness of the abstract interpretation.

Proposition 7.3.1 $abs(case) \sqsubseteq case^{\mathcal{S}}$

Proof

It is not difficult to see that it is sufficient to show that for any b, g, and Z, if $abs(a) \sqsubseteq b$, $abs(f) \sqsubseteq g$, and $abs(xs) \sqsubseteq Z$ then

$$abs(case(a, f, xs)) \sqsubseteq case^{\mathcal{S}}(b, g, Z)$$

We consider the different possible values of Z.

(i) $Z = \bot$

If $abs(xs) \sqsubseteq Z$ then $xs = \bot$. Now,

$$\begin{aligned} abs(case(a, f, xs)) &= abs(case(a, f, \bot)) \\ &= abs(\bot) \\ &= \bot \end{aligned}$$

On the other hand,

$$case^{\mathcal{S}}(b, g, \bot) = \bot$$

In this case, the two values are actually equal.

(ii) $Z = \infty$

If $abs(xs) \sqsubseteq Z$ then either $xs = \bot$ or xs is a partial or an infinite list. The first case is trivial, and hence we assume that $xs = cons\ x\ xs'$. Notice that since $abs(xs) \sqsubseteq Z$, it is also the case that $abs(xs') \sqsubseteq Z$. Now, by definition and by assumption on g

$$\begin{aligned} abs(case(a, f, xs)) &= abs(case(a, f, cons\ x\ xs')) \\ &= abs(f(x, xs')) \\ &\sqsubseteq abs(f)(abs(x), abs(xs')) \\ &\sqsubseteq g(abs(x), abs(xs')) \\ &\sqsubseteq g(\top, \infty) \end{aligned}$$

On the other hand,

$$case^{\mathcal{S}}(b, g, \infty) = g(\top, \infty)$$

Therefore,

$$abs(case(a, f, xs)) \sqsubseteq case^{\mathcal{S}}(b, g, Z)$$

(iii) $Z = Y_\in$, where $Y = \{y_1, ..., y_m\}^\uparrow$

Assume that $Y \neq \{\top\}$; we will consider the other possibility later. Suppose also that $abs(xs) \sqsubseteq Y_\in$. We may also assume that xs is some finite list $[x_1, ..., x_n]$; the remaining possibilities have already been dealt with in the previous cases. Now,

$$abs(xs) \;=\; \{abs(x_1), ..., abs(x_n)\}^\uparrow_\in$$

Moreover,

$$Y \;\subseteq\; \{abs(x_1), ..., abs(x_n)\}^\uparrow,$$

and hence for any open subset Y' of Y

$$(Y \ominus Y')_\in \;\sqsupseteq\; ((abs(xs)) \ominus Y')_\in$$

There are two cases to consider. The first is when $n = 1$. In this case $Y \subseteq \{abs(x_1)\}^\uparrow$, and hence for every $y_i \in Y$ we have $abs(x_1) \sqsubseteq y_i$. Thus, for any subset Y' of Y, $abs(x_1) \sqsubseteq \sqcap Y'$. Therefore, if $Y' = Y$ then $(Y \ominus Y')_\in = \{\top\}_\in$ and hence

$$g(abs(x_1), \{\top\}_\in) \;\sqsubseteq\; g(\sqcap Y', (Y \ominus Y')_\in)$$

Now, we have

$$\begin{aligned}
abs(case(a, f, xs)) &= abs(f(x_1, [])) \\
&\sqsubseteq g(abs(x_1), abs([])) \\
&= g(abs(x_1), \{\top\}_\in) \\
&\sqsubseteq g(\sqcap Y', (Y \ominus Y')_\in) \\
&\sqsubseteq \bigsqcup_{X \in \mathcal{U}_Y} g(\sqcap X, (Y \ominus X)_\in) \\
&= case^{\mathcal{S}}(b, g, Y_\in) \\
&= case^{\mathcal{S}}(b, g, Z)
\end{aligned}$$

Suppose $n \neq 1$. Let $xs' = [x_2, ..., x_n]$ and $Y' = Y \cap \{abs(x_1)\}^\uparrow$. In this case, we have

$$\begin{aligned}
Y - Y' &= Y - (Y \cap \{abs(x_1)\}^\uparrow) \\
&= Y - \{abs(x_1)\}^\uparrow \\
&\subseteq \{abs(x_1), ..., abs(x_n)\}^\uparrow - \{abs(x_1)\}^\uparrow \\
&\subseteq \{abs(x_2), ..., abs(x_n)\}^\uparrow
\end{aligned}$$

Thus,

$$\begin{aligned} abs(xs') &= \{abs(x_2), ..., abs(x_n)\}^{\uparrow}_{\in} \\ &\sqsubseteq (Y \ominus Y')_{\in} \end{aligned}$$

Moreover, $abs(x_1) \sqsubseteq \sqcap Y'$ and hence

$$g(abs(x_1), abs(xs')) \sqsubseteq g(\sqcap Y', (Y \ominus Y')_{\in})$$

On the other hand,

$$\begin{aligned} abs(case(a, f, xs)) &= abs(f(x_1, xs')) \\ &\sqsubseteq abs(f)(abs(x_1), abs(xs')) \\ &\sqsubseteq g(abs(x_1), abs(xs')) \end{aligned}$$

Combining the above results, we obtain

$$\begin{aligned} abs(case(a, f, xs)) &\sqsubseteq g(\sqcap Y', (Y \ominus Y')_{\in}) \\ &\sqsubseteq case^{\mathcal{S}}(b, g, Y_{\in}) \\ &= case^{\mathcal{S}}(b, g, Z) \end{aligned}$$

(iv) $Z = \{\top\}_{\in}$

By definition, for this Z

$$case^{\mathcal{S}}(b, g, Z) = b \sqcup g(\top, Z)$$

If $xs = []$ then $case(a, f, xs) = a$, and since by assumption $abs(a) \sqsubseteq b$, it follows that

$$\begin{aligned} abs(case(a, f, xs)) &= abs(a) \\ &\sqsubseteq b \\ &\sqsubseteq b \sqcup g(\top, \{\top\}_{\in}) \\ &= case^{\mathcal{S}}(b, g, Z) \end{aligned}$$

If xs is a non-empty list we may assume that it is of the form $cons\ x\ xs'$. Now,

$$abs(case(a, f, xs)) = abs(case(a, f, cons\ x\ xs'))$$

$$\begin{aligned} &= abs(f(x, xs')) \\ &\sqsubseteq abs(f)(abs(x), abs(xs')) \\ &\sqsubseteq g(abs(x), abs(xs')) \\ &\sqsubseteq g(\top, \{\top\}_\in) \\ &\sqsubseteq b \sqcup g(\top, \{\top\}_\in) \\ &= case^{\mathcal{S}}(b, g, Z) \end{aligned}$$

□

From this it is easy to see that for the function g defined earlier, we have

$$abs(g) \sqsubseteq g^{\mathcal{S}}$$

Moreover, we can now prove the following proposition.

Proposition 7.3.2 *(i) If $g^{\mathcal{S}}(\infty) = \bot$ then g is tail-strict, and (ii) If $g^{\mathcal{S}}(X_\in) = \bot$ for all $X \neq \{\top\}$, then g is hyper-strict.*

Proof

(i) Suppose $g^{\mathcal{S}}(\infty) = \bot$. Let ys be any infinite or partial list. We want to show that $g(ys) = \bot$. By the previous proposition,

$$\begin{aligned} abs(g(ys)) &\sqsubseteq abs(g)\,(abs(ys)) \\ &\sqsubseteq g^{\mathcal{S}}(\infty) \\ &= \bot \end{aligned}$$

Since *abs* is is $\bot$-reflecting, we have

$$g(ys) = \bot,$$

and hence g is tail-strict.

(ii) Suppose $g^{\mathcal{S}}(X_\in) = \bot$ for all $X \neq \{\top\}$. To prove that g is hyper-strict, let ys be any finite list which is not total. Again,

$$\begin{aligned} abs(g(ys)) &\sqsubseteq abs(g)(abs(ys)) \\ &\sqsubseteq g^{\mathcal{S}}(X_\in), \quad \text{for some } X \neq \{\top\} \\ &= \bot \end{aligned}$$

Since *abs* is is $\perp$-reflecting, we have

$$g(ys) \;=\; \perp$$

Therefore, g is hyper-strict. □

7.3.5 An Example

In a previous subsection, to highlight the difference between the abstract domains used in the original approach, which is Wadler's, and its generalisation by the Nielson and Nielson, a domain of lists of pairs was considered. Here, an example of a function which is defined on lists of pairs of integers is given.

Consider

```
g     nil      = 0
g (cons x xs) = if c then (fst x) + (g xs)
                     else (snd x) + (g xs)
```

Suppose that $c^{\mathcal{S}} = 1$.

(i) It is not difficult to see that Wadler's analysis of g yields the function $g^{\mathcal{S}} : ((\mathbf{2} \times \mathbf{2})_{\perp})_{\perp} \to \mathbf{2}$, which is given by

$$g^{\mathcal{S}}(z) = \begin{cases} 0 & \text{if } z = \perp \text{ or } z = \infty \\ 1 & \text{otherwise} \end{cases}$$

Note that $g^{\mathcal{S}}((0,0)_{\in}) = 1$.

(ii) Using the approach of Nielson and Nielson, $g^{\mathcal{S}} : (L(\mathbf{2}) \times L(\mathbf{2})) \to \mathbf{2}$, is given by

$$g^{\mathcal{S}}(z) = \begin{cases} 0 & \text{if } z = \perp \text{ or } z = \infty \text{ or } z = \{(0,0)\}^{\uparrow}_{\in} \\ 1 & \text{otherwise} \end{cases}$$

Thus, $g^{\mathcal{S}}(\{(0,0)\}^{\uparrow}_{\in}) = 0$.

The first method is more efficient because the domain of the abstract function is normally smaller. As was mentioned earlier, the reason behind our choice of the latter is due to its suitability in dealing with polymorphic functions.

7.4 Polymorphic Functions

We now consider polymorphic functions. It will be shown that the collection of the abstract interpretations of instances of any polymorphic function form a lax natural transformation.

Of particular interest is the function case which will now be regarded as polymorphic. In the next sections, we will show that $case^{\mathcal{S}}$ is a lax natural transformation. To do this, we need some definitions.

7.4.1 L as a functor on $\mathcal{A}^{ec}$

We extend the definition of L so that it becomes a functor on the category of finite lattices and embedding-closure pairs. Its action on objects is already defined. Now, for any embedding-closure pair $h : A \to^{ec} B$ we need to define $L(h) : L(A) \to^{ec} L(B)$. To do this let $L^e(h) : A \to B$ and $L^c(h) : B \to A$ be given by

$$\begin{aligned} L^e(h)(\bot) &= \bot \\ L^e(h)(\infty) &= \infty \\ L^c(h)(\bot) &= \bot \\ L^c(h)(\infty) &= \infty \end{aligned}$$

To complete the definitions we introduce the following notation.
Notation. For any function f and any subset C of the domain of f, $f(C)$ is used to denote the image of C under f, that is, $f(C) = \{f(x) \mid x \in C\}$. Since we use upper-case to represent sets and lower-case to represent points, no confusion should arise as to which application is being referred to a time.

For any $X_\epsilon \in L(A)$ and $Y_\epsilon \in L(B)$, we define

$$\begin{aligned} L^e(h)(X_\epsilon) &= (h^e(X)^\uparrow)_\epsilon, \quad \text{and} \\ L^c(h)(Y_\epsilon) &= (h^c(Y))_\epsilon \end{aligned}$$

$L^e(h)(X_\epsilon)$ is essentially the upwards closure of the image of X under h^e, except that it is an element of $L(A)$ and not $\mathcal{O}(A)$. It can be shown that $h^c(Y)$ is the inverse

image of Y under h^e, that is $h^c(Y) = h^{e^{-1}}(Y)$. This is because,

$$\begin{aligned} x \in h^c(Y) &\Rightarrow x = h^c(y) \quad \text{for some } y \in Y \\ &\Rightarrow h^e(x) \sqsupseteq y \\ &\Rightarrow h^e(x) \in Y \quad \text{since } Y \text{ is open} \\ &\Rightarrow x \in h^{e^{-1}}(Y) \end{aligned}$$

On the other hand,

$$\begin{aligned} x \in h^{e^{-1}}(Y) &\Rightarrow h^e(x) \in Y \\ &\Rightarrow h^c(h^e(x)) \in h^c(Y) \\ &\Rightarrow x \in h^c(Y) \end{aligned}$$

Thus, $h^c(Y)$ is always open because the inverse image of an open set under a continuous function is always open. It appears that there is a lack of symmetry in the above definitions, in the sense that we do not have the operator † in the second case. This is because, unlike h^e, h^c is an *open map*, that is, it maps open sets to open sets, and hence for open sets Y, $h^c(Y)^\dagger = h^c(Y)$.

It is easy to check that both $L^e(h)$ and $L^c(h)$ are continuous. Next, it will be shown that $L(h)$, that is $(L^e(h), L^c(h))$, is an embedding-closure pair. To do this we need the following proposition.

Proposition 7.4.1 *For any non-empty open sets $X \subseteq A$ and $Y \subseteq B$,*

$$(i)\ h^e(h^c(Y))^\dagger \subseteq Y \quad \text{and} \quad (ii)\ h^c(h^e(X)^\dagger) = X$$

Proof

(i) Let $y \in h^e(h^c(Y))^\dagger$. This means that there is some $y_1 \in h^e(h^c(Y))$ such that $y_1 \sqsubseteq y$. Now, $y_1 = h^e(x)$ for some $x \in h^c(Y)$. Also, there is some $y_2 \in Y$ such that $x = h^c(y_2)$. Therefore,

$$y \sqsupseteq y_1 = h^e(x) = h^e(h^c(y_2)) \sqsupseteq y_2 \in Y$$

Since $y_2 \in Y$ and Y is open, we have $y \in Y$. Therefore,

$$h^e(h^c(Y))^\dagger \subseteq Y$$

(ii) Let $x \in h^c(h^e(X)^\dagger)$. Now, $x = h^c(y_1)$ for some $y_1 \in h^e(X)^\dagger$. This implies that $y_1 \sqsupseteq y_2 \in h^e(X)$ for some y_2. Again $y_2 = h^e(x_2)$ for some $x_2 \in X$. Thus,

$$x = h^c(y_1) \sqsupseteq h^c(y_2) = h^c(h^e(x_2)) = x_2$$

Since $x_2 \in X$ and X is open, $x \in X$. Therefore

$$h^c(h^e(X)^\uparrow) \subseteq X$$

Since $h^c \circ h^e = id$, it follows that

$$X = h^c(h^e(X))$$

Clearly,

$$h^c(h^e(X)) \subseteq h^c(h^e(X)^\uparrow)$$

Hence,

$$X \subseteq h^c(h^e(X)^\uparrow)$$

Therefore,

$$h^c(h^e(X)^\uparrow) = X$$

This completes the proof of the Proposition. □

Going back to the definition of the mapping L, we have

$$(L^e(h) \circ L^c(h))(\bot) = \bot \text{ and } (L^e(h) \circ L^c(h))(\infty) = \infty$$

Moreover,

$$(L^c(h) \circ L^e(h))(\bot) = \bot \text{ and } (L^c(h) \circ L^e(h))(\infty) = \infty$$

Also,

$$\begin{aligned} (L^e(h) \circ L^c(h))(Y_\epsilon) &= L^e(h)(h^c(Y)_\epsilon) \\ &= h^e(h^c(Y))^\uparrow_\epsilon \\ &\sqsupseteq Y_\epsilon \end{aligned}$$

and

$$\begin{aligned} (L^c(h) \circ L^e(h))(X_\epsilon) &= L^c(h)(h^e(X)^\uparrow_\epsilon) \\ &= h^c(h^e(X))^\uparrow_\epsilon \\ &= X_\epsilon \end{aligned}$$

It is now easy to see that $L(h)$ is indeed an embedding-closure pair.

It also easy to check that

$$\begin{aligned} L(id) &= id \qquad \text{and} \\ L(h_2 \circ h_1) &= L(h_2) \circ L(h_1) \end{aligned}$$

Therefore, L is a functor from $\mathcal{A}^{ec}$ to $\mathcal{A}^{ec}$.

Some Properties of $L(h)$

The embedding-closure pairs from the two-point lattice **2** to any finite lattice are of special interest; this was seen in the previous chapter. We consider such pairs and see what their images under L are. For every non-top element a of the finite lattice A, consider the embedding-closure pair $h_a : \mathbf{2} \to^{ec} A$, whose definitions is given in Chapter 3. Suppose X and Y are non-empty open subsets of **2** and A respectively. Then, it is not difficult to show that

$$h_a^e(X)^\uparrow = \begin{cases} \{a\}^\uparrow & \text{if } 0 \in X \\ \{\top_A\} & \text{otherwise} \end{cases}$$

Also,

$$h_a^c(Y) = \begin{cases} \{0,1\} & \text{if } a \in Y \\ \{1\} & \text{otherwise} \end{cases}$$

Thus,

$$L^e(h_a)(X_\in) = \begin{cases} \{a\}_\in^\uparrow & \text{if } 0 \in X \\ \{\top_A\}_\in & \text{otherwise} \end{cases}$$

and

$$L^c(h_a)(Y_\in) = \begin{cases} \{0,1\}_\in & \text{if } a \in Y \\ \{1\}_\in & \text{otherwise} \end{cases}$$

In the first and third equations above, the condition $0 \in X$ is equivalent to $X = \{0,1\}$ because X is an open subset of **2**.

In previous chapters, we have already seen how type constructors can be interpreted as functors. In the case of lists, the mapping *List* can be extended to be a functor from $\mathcal{C}^{ec}$ to $\mathcal{C}^{ec}$. With regard to abstract interpretation, what is different about the list type constructor, is that the functors *List* and L are not similar. In the case of the function type constructor $\to$, for example, the functors interpreting it both in the standard semantics and the abstract semantics are defined in the same way. Actually, the second can be seen as the restriction of the first, because it is only defined on finite lattices and embedding-closure pairs between them.

7.4.2 Some Lax Natural Transformations

It is easy to show that the collection of mappings $cons^{\mathcal{S}}_A : A \times L(A) \to L(A)$ is a lax natural transformation; in fact it is a natural transformation. We omit the proof; it follows from the definition very easily. We give only the proof of the fact that $\{case^{\mathcal{S}}_{A,B}\}$ is a lax natural transformation from F to G, where

$$F(A,B) = B \times (A \times L(A) \to B) \times L(A) \text{ and } G(A,B) = B.$$

Suppose $h : A \to^{ec} C$ and $k : B \to^{ec} D$. From the type information, we have

$$F^e(h,k) = k^e \times (\lambda v.\ k^e \circ v \circ h^c \times L^c(h)) \times L^e(h) \text{ and } G^e(h,k) = k^e.$$

The following lemma will be used in the proof of the proposition following it.

Lemma 7.4.1 *Let $h : A \to^{ec} B$ be an embedding-closure pair, and let X and Y be non-empty open subsets of B such that $X \subseteq Y$. Then*

$$h^c(Y) \ominus h^c(X) \subseteq h^c(Y \ominus X)$$

Proof

To show this let $a \in h^c(Y) \ominus h^c(X)$. If $a = \top_A$ then since $h^c(Y \ominus X)$ is a non-empty open set, a must be one of its elements. If $a \neq \top_A$ then $a \in h^c(Y)$ and $a \notin h^c(X)$. Thus, $a = h^c(y)$ for some $y \in Y$. Clearly $y \notin X$, and therefore $y \in (Y - X)$ which implies that $y \in Y \ominus X$ and hence $a \in h^c(Y \ominus X)$. Thus, we have

$$h^c(Y) \ominus h^c(X) \subseteq h^c(Y \ominus X)$$

□

A consequence of this lemma is that

$$L^c(h)((Y \ominus X)_\in) \sqsubseteq (h^c(Y) \ominus h^c(X))_\in$$

Proposition 7.4.2 *$case^{\mathcal{S}}$ is a lax natural transformation from F to G. That is, if A, B, C and D are finite lattices, $h : A \to^{ec} C$ and $k : B \to^{ec} D$ are embedding-closure pairs, then we have*

$$case^{\mathcal{S}}_{C,D} \circ F^e(h,k) \sqsubseteq G^e(h,k) \circ case^{\mathcal{S}}_{A,B}$$

Proof

Let a_B be any element of B, and $f : C \times L(C) \to D$ be any continuous function.

We want to show that for any $Z \in L(A)$, we have

$$(case^{\mathcal{S}}_{C,D} \circ F^e(h,k))(a_B, f, Z) \sqsubseteq (G^e(h,k) \circ case^{\mathcal{S}}_{A,B})(a_B, f, Z)$$

Now, there are four cases to consider.

(i) $Z = \bot_{L(A)}$

$$\begin{aligned}
&(case^{\mathcal{S}}_{C,D} \circ F^e(h,k))(a_B, f, \bot_{L(A)}) \\
&\quad = case^{\mathcal{S}}_{C,D}(k^e(a_B), k^e \circ f \circ (h^c \times L^c(h)), L^e(h)(\bot_{L(A)})) \\
&\quad = case^{\mathcal{S}}_{C,D}(k^e(a_B), k^e \circ f \circ (h^c \times L^c(h)), \bot_{L(C)}) \\
&\quad = \bot_D
\end{aligned}$$

On the other hand,

$$(G^e(h,k) \circ case^{\mathcal{S}}_{A,B})(a_B, f, \bot_{L(A)}) = k^e(\bot_B)$$

Since $\bot_D \sqsubseteq k^e(\bot_B)$, the inequality holds.

(ii) $Z = \infty$

$$\begin{aligned}
&(case^{\mathcal{S}}_{C,D} \circ F^e(h,k))(a_B, f, \infty) \\
&\quad = case^{\mathcal{S}}_{C,D}(k^e(a_B), k^e \circ f \circ (h^c \times L^c(h)), L^e(h)(\infty)) \\
&\quad = case^{\mathcal{S}}_{C,D}(k^e(a_B), k^e \circ f \circ (h^c \times L^c(h)), \infty) \\
&\quad = (k^e \circ f \circ (h^c \times L^c(h)))(\top_C, \infty) \\
&\quad = k^e(f(h^c(\top_C), L^c(h)(\infty))) \\
&\quad = k^e(f(\top_A, \infty))
\end{aligned}$$

Also,

$$(G^e(h,k) \circ case^{\mathcal{S}}_{A,B})(a_B, f, \infty) = k^e(f(\top_A, \infty))$$

Again, the inequality holds.

(iii) $Z = \{\top_A\}_\in$

$$\begin{aligned}
&(case^{\mathcal{S}}_{C,D} \circ F^e(h,k))(a_B, f, \{\top_A\}_\in) \\
&\quad = case^{\mathcal{S}}_{C,D}(k^e(a_B), k^e \circ f \circ (h^c \times L^c(h)), L^e(h)(\{\top_A\}_\in)) \\
&\quad = case^{\mathcal{S}}_{C,D}(k^e(a_B), k^e \circ f \circ (h^c \times L^c(h)), \{\top_C\}_\in)
\end{aligned}$$

$$
\begin{aligned}
&= k^e(a_B) \sqcup (k^e \circ f \circ (h^c \times L^c(h))(\top_C, \{\top_C\}_\epsilon)) \\
&= k^e(a_B) \sqcup k^e(f(h^c(\top_C), L^c(h)(\{\top_C\}_\epsilon))) \\
&\sqsubseteq k^e(a_B \sqcup f(h^c(\top_C), L^c(h)(\{\top_C\}_\epsilon))) \\
&= k^e(a_B \sqcup f(\top_A, \{\top_A\}_\epsilon))
\end{aligned}
$$

On the other hand,

$$(G^e(h,k) \circ case^{\mathcal{S}}_{A,B})(a_B, f, \{\top_A\}_\epsilon) = k^e(a_B \sqcup f(\top_A, \{\top_A\}_\epsilon))$$

(iv) $Z = Y_\epsilon$

We may assume that $Y \neq \{\top_A\}$, because we have treated the other case already. Let $W = h^e(Y)^\dagger$.

$$
\begin{aligned}
&(case^{\mathcal{S}}_{C,D} \circ F^e(h,k))(a_B, f, Y_\epsilon) \\
&\quad = case^{\mathcal{S}}_{C,D}(k^e(a_B), k^e \circ f \circ (h^c \times L^c(h)), L^e(h)(Y_\epsilon)) \\
&\quad = case^{\mathcal{S}}_{C,D}(k^e(a_B), k^e \circ f \circ (h^c \times L^c(h)), h^e(Y)^\dagger_\epsilon) \\
&\quad = \textstyle\bigsqcup_{Z \in \mathcal{U}_W} (k^e \circ f \circ (h^c \times L^c(h)))(\sqcap Z, (h^e(Y)^\dagger \ominus Z)_\epsilon) \\
&\quad = \textstyle\bigsqcup_{Z \in \mathcal{U}_W} k^e(f(h^c(\sqcap Z), L^c(h)((h^e(Y)^\dagger \ominus Z)_\epsilon))) \\
&\quad = \textstyle\bigsqcup_{Z \in \mathcal{U}_W} k^e(f(h^c(\sqcap Z), (h^c(h^e(Y)^\dagger \ominus Z))_\epsilon)) \\
&\quad \sqsubseteq \textstyle\bigsqcup_{Z \in \mathcal{U}_W} k^e(f(\sqcap h^c(Z), (h^c(h^e(Y)^\dagger) \ominus h^c(Z))_\epsilon)) \\
&\quad = \textstyle\bigsqcup_{Z \in \mathcal{U}_W} k^e(f(\sqcap h^c(Z), (Y \ominus h^c(Z))_\epsilon)) \\
&\quad \sqsubseteq k^e(\textstyle\bigsqcup_{Z \in \mathcal{U}_W} f(\sqcap h^c(Z), (Y \ominus h^c(Z))_\epsilon))
\end{aligned}
$$

On the other hand,

$$(G^e(h,k) \circ case^{\mathcal{S}}_{A,B})(a_B, f, Y_\epsilon) = k^e(\textstyle\bigsqcup_{X \in \mathcal{U}_Y} f(\sqcap X, (Y \ominus X)_\epsilon))$$

It is now sufficient to show that for any non-empty open subset V of $h^e(Y)^\dagger$, *i.e* $V \in \mathcal{U}_W$, the following holds

$$f(\sqcap h^c(V), (Y \ominus h^c(V))_\epsilon) \sqsubseteq \textstyle\bigsqcup_{X \in \mathcal{U}_Y} f(\sqcap X, (Y \ominus X)_\epsilon)$$

Let $X' = h^c(V)$. It is easy to check that $X' \subseteq Y$, that is $X' \in \mathcal{U}_Y$. Hence,

$$
\begin{aligned}
f(\sqcap h^c(V), (Y \ominus h^c(V))_\epsilon) &= f(\sqcap X', (Y \ominus X')_\epsilon) \\
&\sqsubseteq \bigsqcup_{X \in \mathcal{U}_Y} f(\sqcap X, (Y \ominus X)_\epsilon)
\end{aligned}
$$

Therefore, $case^{\mathcal{S}}$ is a lax natural transformation. □

Suppose f is a polymorphic function with type $\forall t.t \times [t] \to H(t)$, and g is the polymorphic function defined by `g(xs) = case(a,f,xs)` with type $\forall t.[t] \to H(t)$. For any finite lattices A and C, and an embedding-closure pair $h : A \to^{ec} C$, assuming that

$$f_C^{\mathcal{S}} \circ (h^e \times L^e(h)) \sqsubseteq H^e(h) \circ f_A^{\mathcal{S}}$$

it is possible to show that

$$g_C^{\mathcal{S}} \circ L^e(h) \sqsubseteq H^e(h) \circ g_A^{\mathcal{S}}$$

(Strictly speaking, we should have H' instead of H here.) To show this inequality, all we have to do is use the above proposition where we have $H(A)$ and $H(C)$ instead of B and D, and $H(h)$ instead of k. Therefore,

$$g_C^{\mathcal{S}} \sqsubseteq \bigsqcap H^e(h) \circ g_A^{\mathcal{S}} \circ L^c(h)$$

The greatest lower bound is taken over all possible h's. When A is **2**, from the nature of the embedding-closure pairs defined on **2**, which we studied in Chapter 3, we have

$$g_C^{\mathcal{S}} \sqsubseteq \bigsqcap H^e(h_u) \circ g_2^{\mathcal{S}} \circ L^c(h_u)$$

where u ranges over all non-top elements of C. Here as well, we obtain the same approximation by using only the meet-irreducible elements of C. This, in many cases, reduces the number of functions involved when taking the greatest lower bound.

7.5 Implementation

We reported how to obtain an approximation to the abstract interpretation of any instance of a polymorphic higher-order function from that of the smallest in [6]. The type system of the language used in that work did not include lists. Seward's implementation of this method is for a lazy functional language called **Core** [48]. His implementation also extends to lists, although there is no known proof of correctness for this extension. It was our inability of proving the technique correct that lead

us to consider the method by Nielson and Nielson. However, many of the results in Seward's paper remain valid in this framework.

One of the examples given in [48] involves the function

```
foldr :: (a -> b -> b) -> b -> [a] -> b
    foldr f a    nil       = a
    foldr f a (cons x xs)  = f x (foldr f a xs)
```

The abstract interpretation of the smallest instance of this function, that is $foldr^{\mathcal{S}}_{\mathbf{2},\mathbf{2}}$, would have 48 entries when tabulated. On the other hand, $foldr^{\mathcal{S}}_{L(\mathbf{2}),L(\mathbf{2})}$ has a vast number of entries; in fact Hunt reports that they are of the order of 10^6 [27]. Thus, if one computes the latter by the usual iterative means, checking for equality of functions at each iterative step will be very expensive.

Consider the function

```
concat = foldr append nil
```

where

```
append :: [a] -> [a] -> [a]
append     nil      ys   = ys
append (cons x xs)  ys   = cons x (append xs ys)
```

In the computation of the abstract interpretation of the simplest instance of `concat`, $foldr^{\mathcal{S}}_{L(\mathbf{2}),L(\mathbf{2})}$ is involved. This is because

$$concat^{\mathcal{S}}_{\mathbf{2}} \;=\; foldr^{\mathcal{S}}_{L(\mathbf{2}),L(\mathbf{2})}\; append^{\mathcal{S}}_{\mathbf{2}}\; nil^{\mathcal{S}}$$

Instead of computing $foldr^{\mathcal{S}}_{L(\mathbf{2}),L(\mathbf{2})}$ directly one can use

$$\bigsqcap G^{e}(h_a,k_b) \circ foldr^{\mathcal{S}}_{\mathbf{2},\mathbf{2}} \circ F^{c}(h_a,k_b),$$

where both h_a and k_b range over all the embedding-closure pairs from $\mathbf{2}$ to $L(\mathbf{2})$. Notice that

$$F(A,B) = A \to B \to B \text{ and } G(A,B) = B \to L(A) \to B.$$

These definitions follow from the type of the function.

According to Seward's experiments, the use of the above approximation in analysing `concat` makes it 750 times faster than computing it directly. Moreover, the approximate value coincides with the exact value in this case. In this kind of examples, we are free to use Wadler's domains because, the lattices under consideration are chains.

7.6 Semantic Polymorphic Invariance

We have already promised to prove the semantic polymorphic invariance of some properties of functions defined on lists. The properties which were considered in the monomorphic case are tail-strictness and hyper-strictness. We study the invariance of these properties. In order to do this we need some definitions.

List as a functor on $\mathcal{C}^{ec}$

Corresponding to the list type constructor we define a mapping *List* on the category of domains and embedding-closure pairs, $\mathcal{C}^{ec}$. For any domain D let $List(D)$ denote the domain of lists whose elements come from D. For any embedding-closure pair $h : D \to^{ec} E$, we define $List(h) : List(D) \to^{ec} List(E)$ by $List(h) = (map\ h^e, map\ h^c)$, where *map* is given by

$$\begin{aligned} map\ k\ \bot &= \bot \\ map\ k\ [] &= [] \\ map\ k\ (cons\ x\ xs) &= cons\ (k(x))\ (map\ k\ xs) \end{aligned}$$

To show that $List(h)$ is indeed an embedding-closure pair is not difficult. The only remark which perhaps is necessary to make here is, that when both xs and ys are infinite lists, the proof of

$$\begin{aligned} map\ h^c\ (map\ h^e\ (cons\ x\ xs)) &= cons\ x\ xs \\ map\ h^e\ (map\ h^c\ (cons\ y\ ys)) &\sqsupseteq cons\ y\ ys \end{aligned}$$

follows by using partial lists approximating xs and ys and the continuity of the functions involved; all the remaining cases are easy to verify. It is now easy to check that *List* is a functor from $\mathcal{C}^{ec}$ to itself.

Semantics of case

For domains D and E, let $case_{D,E}$ be the concrete semantics of an instance of case, where

$$\begin{aligned} case_{D,E}(a, f, \bot) &= \bot \\ case_{D,E}(a, f, []) &= a \\ case_{D,E}(a, f, cons\ x\ xs) &= f(x, xs) \end{aligned}$$

Now, we want to show that the collection of such functions is a lax natural transformation from F to G, where

$$F(D, E) = E \times (D \times List(D) \to E) \times List(D) \text{ and } G(D, E) = E.$$

Suppose $h : D \to^{ec} R$ and $k : E \to^{ec} S$ are embedding-closure pairs. Then,

$$F^e(h, k) = (k^e, \lambda g.\ k^e \circ g \circ (h^c \times List^c(h)), List^e(h)) \text{ and } G^e(h, k) = k^e.$$

Now,

$$\begin{aligned} (case_{R,S} \circ F^e(h,k))(a, f, \bot) &= \bot \\ (case_{R,S} \circ F^e(h,k))(a, f, []) &= k^e(a) \\ (case_{R,S} \circ F^e(h,k))(a, f, cons\ x\ xs) &= k^e(f(h^c(h^e(x)), map\ h^c\ (map\ h^e\ xs))) \\ &= k^e(f(x, xs)) \end{aligned}$$

On the other hand,

$$\begin{aligned} (G^e(h,k) \circ case_{D,E})(a, f, \bot) &= k^e(\bot) \\ (G^e(h,k) \circ case_{D,E})(a, f, []) &= k^e(a) \\ (G^e(h,k) \circ case_{D,E})(a, f, cons\ x\ xs) &= k^e(f(x, xs)) \end{aligned}$$

From the two sets of equations above, it is easy to observe that

$$case_{R,S} \circ F^e(h,k) \sqsubseteq G^e(h,k) \circ case_{D,E}$$

Therefore, the collection is a lax natural transformation.

Suppose the `f` in the definition of g is polymorphic with type $\forall t.t \times [t] \rightarrow H(t)$ so that it is possible to assume that

$$f_R \circ (h^e \times List^e(h)) \sqsubseteq H^e(h) \circ f_D$$

where f is the semantics of `f`. Also, if we assume that $a_R \sqsubseteq H^e(h)(a_D)$, where a_D and a_R are the semantics of `a` in $H(D)$ and in $H(R)$ respectively, this statement together with the result that we just proved imply that

$$g_R \circ List^e(h) \sqsubseteq H^e(h) \circ g_D$$

That is, g is interpreted by a lax natural transformation. To see this, use $H(D)$, $H(R)$, and $H(h)$ instead of E, S, and k respectively in the proof above.

Coming back to the question of invariance :

- Suppose that g_D is tail-strict. Without loss of generality, assume D is the two-element lattice **2**. We want to show that g_R is tail-strict. Suppose ys is either an infinite or a partial list. We have to show that $g_R(ys) = \perp$. From the lax natural transformation condition we have

  $$g_R(ys) \sqsubseteq (H^e(h) \circ g_{\mathbf{2}} \circ List^c(h))(ys)$$

  Since $List^c(h)(ys)$ is also an infinite or a partial list, and $g_{\mathbf{2}}$ is tail-strict, it follows that

  $$g_R(ys) \sqsubseteq H^e(h)(\perp)$$

  This is true for all h. As in Chapter 5, if we assume that R has a top element then there always exists an embedding-closure pair h for which the embedding component h^e is strict, and moreover $H^e(h)$ is also strict. Hence, we have

  $$g_R(ys) = \perp$$

  Therefore, g_R is also tail-strict.

- On the other hand, suppose $g_{\mathbf{2}}$ is hyper-strict. To prove that g_R is also hyper-strict, let ys be any list that is not total. Now, it is not difficult to show that $List^c(h)(ys)$ is also not total. Thus, the inequality

  $$g_R(ys) \sqsubseteq (H^e(h) \circ g_{\mathbf{2}} \circ List^c(h))(ys)$$

implies that

$$g_R(ys) \sqsubseteq H^e(h)(\bot)$$

By an argument similar to the previous case, we have $g_{R,S}(ys) = \bot$, that is, g_R is hyper-strict.

What we have done so far is to prove a partial result. Roughly, what it says is that if any of the properties holds for smaller instances of a polymorphic function then that property must also hold for the higher instances as well. To complete the proof it is necessary to prove the other half. That is, if the higher (complex) instances have the property, so do the smaller (simpler) ones. However, as was shown in Chapter 5, one needs to use the category of domains and embedding-projection pairs to do this. The lax natural transformation condition for g between functors on this category implies that

$$H^p(h) \circ g_R \circ List^e(h) \sqsupseteq g_D$$

for all embedding-projection pairs $h : D \rightarrow^{ep} R$. Notice also that the functor *List* here is different from the one that was being use so far. From this inequality, the remaining half of the proof is obtained.

The statement about the invariance of hyper-strictness is perhaps vacuous. It appears that there are no examples of hyper-strict polymorphic functions in the language. In fact, that is what one would intuitively expect. We included the proof of its invariance because no extra effort was required.

Chapter 8

Further Applications

There are other analyses of programs that can be done by abstract interpretation in a way similar to strictness analysis. The inefficiency in computing fixed points by iteration is also present in these analyses, so we have problems similar to the ones we had before with polymorphic functions. Fortunately, techniques similar to those used in Chapter 6 can also be applied here. That is, an approximation to the abstract interpretation of any instance of a polymorphic function can be obtained from that of the simplest.

The two analyses which we study here are *binding-time analysis* and *termination analysis.*

8.1 Binding-Time Analysis

For the purpose of partial evaluation, expressions in a program may be classified as *static* or *dynamic.* Any expression that can be evaluated during partial evaluation would be classified as static, and all other expressions are classified as dynamic. Binding-time analysis is the analysis which determines these properties. One way of performing this is by abstract interpretation. As one would expect, there may be expressions which are static but this fact may not be detected by the abstract interpretation. We have a similar situation in strictness analysis; there are strict

functions whose strictness could not be detected by the abstract interpretation we were discussing in the previous chapters.

In the first-order case, the abstract interpretation for binding-time analysis is done on the two-element domain $\{S, D\}$ with $S \sqsubset D$, where S represents the property of being static and D represents the property of being dynamic. An abstract interpretation that uses this domain is given by Sestoft [46]. Strictly speaking, any expression for which we could not decide if it is static is classified as dynamic. There is some analogy between D here and 1 as used in strictness analysis.

8.1.1 Higher-Order Functions

An abstract interpretation for binding-time analysis of a simply-typed λ-calculus with constants was defined by Hunt and Sands [26]. Our study of polymorphic functions will make use of this abstract interpretation. Their method is very similar to the one used in the strictness analysis of higher-order functions by Burn et al [11].

The notation we shall follow will be similar to the one in Chapter 5. Assume that **int** and **bool** are the only basic types, and that Int and $Bool$ are their respective semantic domains. The abstract domains corresponding to these semantic domains (or types) are given by

$$Abs\ Int = Abs\ Bool = \{S, D\}$$

The abstract domains corresponding to products and function spaces are given by

$$\begin{aligned} Abs\ (D_1 \times D_2) &= (Abs\ D_1) \times (Abs\ D_2) \\ Abs\ (D_1 \to D_2) &= (Abs\ D_1) \to (Abs\ D_2) \end{aligned}$$

Hunt and Sands also deal with lists, however we will delay this until later.

8.1.2 Interpretation of Terms

If one identifies 0 with S and 1 with D, there is a similarity between the abstract interpretation for strictness analysis in [11] and the one used in binding-time analysis.

The only difference is that here

- integer and boolean constants are interpreted by S.
- the usual binary operators `+`,`-`,`*` and `/` are interpreted by $\sqcup$.
- the conditional `if` is interpreted by $if^{\mathcal{B}}$, where for a lattice A,
$$if^{\mathcal{B}}_A : \{S, D\} \to A \to A \to A$$
is given by
$$if^{\mathcal{B}}_A\ a\ b\ c = \begin{cases} \top_A & \text{if } a = D \\ b \sqcup c & \text{if } a = S \end{cases}$$
Obviously, we are assuming that A is the lattice that corresponds to the type of the appropriate instance of `if`.

The remaining expressions are interpreted in exactly the same way as for strictness analysis, and of course, subject to the identification of elements of the abstract domains which we mentioned above.

Notation. For any function `f`, we shall denote its abstract version by $f^{\mathcal{B}}$ ($\mathcal{B}$ refers to the fact that we are dealing with the abstract interpretation for binding-time analysis).

Examples

(1) Let

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

Now, $f^{\mathcal{B}}$ is given by
$$f^{\mathcal{B}}\ x\ y\ z = \begin{cases} \top & \text{if } x = D \\ y & \text{if } x = S \end{cases}$$
Therefore, if $\mathtt{E}_0$ and $\mathtt{E}_1$ are static, for example, then `f` $\mathtt{E}_0$ $\mathtt{E}_1$ $\mathtt{E}_2$ is also static even when $\mathtt{E}_2$ is dynamic.

(2) Let

```
g x = if True 0 x
h x = 0
```

Clearly, g and h have the same semantics. On the other hand, for every x

$$g^{\mathcal{B}}\ x\ =\ x \text{ and } h^{\mathcal{B}}\ x\ =\ S$$

From this we conclude that h E is static for every E, whereas this is not the case for g E. More precisely, the abstract interpretation detects that g E is static only when E is static.

8.1.3 Polymorphism

Because of the similarity with the abstract interpretation for strictness analysis, to conclude that the abstract interpretation (for binding-time analysis) of any polymorphic function is a lax natural transformation, we only need to look at the functions whose interpretation are defined differently. In fact, the only function that we need to check is the conditional.

We show that the collection $\{if_A^{\mathcal{B}}\}$ is a lax natural transformation between the functors F and G on the category of finite lattices and embedding-closure pairs, where

$$F(A) = \{S, D\} \text{ and } G(A) = A \to A \to A$$

The actions of F and G on morphisms are given in the usual way; in fact F is the constant functor always returning the identity morphism on $\{S, D\}$. Now, let $h : A \to^{ec} B$ be an embedding-closure pair. Since $F^e(h) = id$, we have

$$\begin{aligned}
(if_B^{\mathcal{B}} \circ F^e(h))\ S\ b\ c &= if_B^{\mathcal{B}}\ S\ b\ c \\
&= b \sqcup_B c \\
(if_B^{\mathcal{B}} \circ F^e(h))\ D\ b\ c &= if_B^{\mathcal{B}}\ D\ b\ c \\
&= \top_B
\end{aligned}$$

On the other hand,

$$\begin{aligned}
(G^e(h) \circ if_A^{\mathcal{B}})\ S\ b\ c &= h^e(if_A^{\mathcal{B}}\ S\ h^c(b)\ h^c(c)) \\
&= h^e(h^c(b) \sqcup_A h^c(c)) \\
&= h^e(h^c(b \sqcup_B c)) \\
&\sqsupseteq b \sqcup_B c \\
&= (if_B^{\mathcal{B}} \circ F^e(h))\ S\ b\ c
\end{aligned}$$

Also,

$$\begin{aligned}(G^e(h) \circ if_A^{\mathcal{B}})\ D\ b\ c &= h^e(if_A^{\mathcal{B}}\ D\ h^c(b)\ h^c(c)) \\ &= h^e(\top_A) \\ &= \top_B \\ &= (if_B^{\mathcal{B}} \circ F^e(h))\ D\ b\ c\end{aligned}$$

Combining all these we have

$$if_B^{\mathcal{B}} \circ F^e(h) \sqsubseteq G^e(h) \circ if_A^{\mathcal{B}}$$

That is, the collection is indeed a lax natural transformation form F to G.

The operation $\sqcup$ in the definition above should normally be indexed by the appropriate lattice. In fact, we have already shown, in Chapter 3, that the collection $\{\sqcup_A\}$ is a lax natural transformation between the functors $Id \times Id$ and Id.

Now in general, for any polymorphic function `f` of type $\forall t.F(t) \to G(t)$, we have

$$f_B^{\mathcal{B}} \circ F^e(h) \sqsubseteq G^e(h) \circ f_A^{\mathcal{B}}$$

and hence,

$$f_B^{\mathcal{B}} \sqsubseteq \bigsqcap G^e(h) \circ f_A^{\mathcal{B}} \circ F^c(h)$$

The greatest lower bound is taken over all possible embedding-closure pairs h. As in strictness analysis, if `f` is recursive and B is a big lattice then the computation of $f_B^{\mathcal{B}}$ by iterative means is normally very expensive. Instead, because of the above inequality, we let $A = \{S, D\}$ and first compute $f_A^{\mathcal{B}}$ by iteration, and then build an approximation to $f_B^{\mathcal{B}}$ from it.

One of the desirable features of the abstract interpretation described above is that it also works for higher-order functions. This is a noteworthy advance over the work of Launchbury [33] where he uses projections in his binding-time analysis of polymorphic first-order functions.

8.1.4 Lists

Hunt and Sands also deal with lists. The abstract domain they use in the abstract interpretation of, for example, lists of integers is the three-element domain

$$\{SPINE(S), SPINE(D), D\}$$

where

$$SPINE(S) \sqsubset SPINE(D) \sqsubset D$$

$SPINE(x)$ is the property of a list with static structure with all its elements having property x.

In general, for any finite lattice A, they define the domain

$$\{SPINE(a) \mid a \in A\} \cup \{D\}$$

where D is the top element, and $SPINE(a) \sqsubseteq SPINE(b)$ if and only if $a \sqsubseteq b$. From now on, we shall denote this domain by $K(A)$. The operator K is analogous to L in the previous chapter.

K as a functor on $\mathcal{A}^{ec}$

It is now possible to extend the definition of K so that it becomes a functor from the category of finite lattices and embedding-closure pairs to itself. To do this, let $h : A \rightarrow^{ec} B$ be an embedding-closure pair. Define the functions

$$K^e(h) : K(A) \rightarrow K(B) \text{ and } K^c(h) : K(B) \rightarrow K(A)$$

by

$$\begin{aligned} K^e(h)(D) &= D \\ K^e(h)(SPINE(a)) &= SPINE(h^e(a)) \\ K^c(h)(D) &= D \\ K^c(h)(SPINE(b)) &= SPINE(h^c(b)) \end{aligned}$$

Clearly both functions are continuous. Moreover, it is easy to show that $(K^e(h), K^c(h))$ is an embedding-closure pair and furthermore, K is a functor; we omit the verification.

Definition of $cons^{\mathcal{B}}$

Going back to some operations that are defined on lists, we first consider **cons**. Now, as given in [26] for the monomorphic case

$$cons^{\mathcal{B}}_A : A \times K(A) \rightarrow K(A)$$

is defined by

$$cons_A^{\mathcal{B}}(a,l) = \begin{cases} D & \text{if } l = D \\ SPINE(a \sqcup a') & \text{if } l = SPINE(a') \end{cases}$$

Also $nil_A^{\mathcal{B}} = SPINE(\perp_A)$. Clearly $cons_A^{\mathcal{B}}$ is continuous. Moreover, the collection $\{cons_A^{\mathcal{B}}\}$ is a lax natural transformation between the functors $Id \times K$ and K. To show this we note the following

$$\begin{aligned} (K^e(h) \circ cons_A^{\mathcal{B}})(a, D) &= K^e(h)(D) \\ &= D \\ (K^e(h) \circ cons_A^{\mathcal{B}})(a, SPINE(a')) &= K^e(h)(SPINE(a \sqcup a')) \\ &= SPINE(h^e(a \sqcup a')) \\ &\sqsupseteq SPINE(h^e(a) \sqcup h^e(a')) \end{aligned}$$

On the other hand,

$$\begin{aligned} (cons_B^{\mathcal{B}} \circ (h^e \times K^e(h)))(a, D) &= cons_B^{\mathcal{B}}(h^e(a), K^e(h)(D)) \\ &= cons_B^{\mathcal{B}}(h^e(a), D) \\ &= D \\ (cons_B^{\mathcal{B}} \circ (h^e \times K^e(h)))(a, SPINE(a')) &= cons_B^{\mathcal{B}}(h^e(a), SPINE(h^e(a'))) \\ &= SPINE(h^e(a) \sqcup h^e(a')) \end{aligned}$$

From these statements, we obtain the desired result. That is,

$$cons_B^{\mathcal{B}} \circ (h^e \times K^e(h)) \sqsubseteq K^e(h) \circ cons_A^{\mathcal{B}}$$

Notice also that

$$\begin{aligned} nil_B^{\mathcal{B}} &= SPINE(\perp_B) \\ &\sqsubseteq SPINE(h^e(\perp_A)) \\ &= K^e(h)(SPINE(\perp_A)) \\ &= K^e(h)(nil_A^{\mathcal{B}}) \end{aligned}$$

Therefore, $nil_B^{\mathcal{B}} \sqsubseteq K^e(h)(nil_A^{\mathcal{B}})$; this property will be used later.

Definition and properties of $case^{\mathcal{B}}$

As in Chapter 7, we concentrate on functions defined in terms of `case`. An important issue here is how to define

$$case^{\mathcal{B}}_{A,B} : B \times (A \times K(A) \to B) \times K(A) \to B$$

To motivate the definition which we will give, let $b \in B$, $f : A \times K(A) \to B$, and $as \in K(A)$. Now, $case^{\mathcal{B}}_{A,B}(b, f, as)$ should depend on the nature of as.

- If $as = D$ it is reasonable to let $case^{\mathcal{B}}_{A,B}(b, f, as) = D$
- If $as = SPINE(v)$ then the list whose abstraction is given to be as is either `nil` or `cons x xs`. In the latter case, its abstract version will be something of the form $cons^{\mathcal{B}}_{A}(a, l)$ for some a and l, and moreover l itself must be of the form $SPINE(a')$ for some a'. Hence, it is safe to assume that both a and a' are equal to v. Thus, we define

$$case^{\mathcal{B}}_{A,B}(b, f, as) \;=\; b \sqcup (f(v, SPINE(v)))$$

Statements analogous to those about the abstract interpretation for strictness can now be made. That is, $\{case^{\mathcal{B}}_{A,B}\}$ is a lax natural transformation between the functors F and G, where

$$F(A, B) = B \times (A \times K(A) \to B) \times K(A) \text{ and } G(A, B) = B$$

Again, their action on morphisms is defined in the usual way. Now, suppose we have finite lattices A,B,C and D, and embedding-closure pairs

$$h : A \to^{ec} C \text{ and } k : B \to^{ec} D$$

Now, we have

$$\begin{aligned} (k^e \circ case^{\mathcal{B}}_{A,B})(b, f, D) &= k^e(D) \\ &= D \qquad \text{(since } k^e \text{ } \top\text{-preserving)} \end{aligned}$$

Also,

$$\begin{aligned} (k^e \circ case^{\mathcal{B}}_{A,B})(b, f, SPINE(v)) &= k^e(b \sqcup (f(v, SPINE(v)))) \\ &\sqsupseteq k^e(b) \sqcup k^e(f(v, SPINE(v))) \end{aligned}$$

On the other hand, letting

$$M = case^{\mathcal{B}}_{C,D} \circ (k^e \times \lambda u.(k^e \circ u \circ (h^c \times K^c(h))) \times K^e(h))$$

we have

$$\begin{aligned} M(b,f,D) &= D \\ M(b,f,SPINE(v)) &= case^{\mathcal{B}}_{C,D}(k^e(b), k^e \circ f \circ (h^c \times K^c(h)), K^e(h)(SPINE(v))) \\ &= case^{\mathcal{B}}_{C,D}(k^e(b), k^e \circ f \circ (h^c \times K^c(h)), SPINE(h^e(v))) \\ &= k^e(b) \sqcup k^e(f(h^c(h^e(v)), K^c(h)(SPINE(h^e(v))))) \\ &= k^e(b) \sqcup k^e(f(v, SPINE(v))) \end{aligned}$$

It is now easy to see that the collection is indeed a lax natural transformation. That is,

$$case^{\mathcal{B}}_{C,D} \circ F^e(h,k) \sqsubseteq G^e(h,k) \circ case^{\mathcal{B}}_{A,B}$$

Recall the definition of the function **g** which was given in Chapter 7 by

```
g(xs) = case(a,f,xs)
```

Assuming that `f` is polymorphic with type $\forall t.t \times [t] \to H(t)$ and hence

$$f^{\mathcal{B}}_C \circ (h^e \times K^e(h)) \sqsubseteq H^e(h) \circ f^{\mathcal{B}}_A$$

and also that

$$a^{\mathcal{B}}_C \sqsubseteq H^e(h)(a^{\mathcal{B}}_A)$$

we can show that

$$g^{\mathcal{B}}_C \sqsubseteq H^e(h) \circ g^{\mathcal{B}}_A \circ K^c(h)$$

(Note that, strictly speaking, we should have H' here.) Recall that for any xs

$$g^{\mathcal{B}}_A(xs) = case^{\mathcal{B}}_{A,H(A)}(a^{\mathcal{B}}_A, f^{\mathcal{B}}_A, xs)$$

Taking greatest lower bound over all possible expressions of the form which appear in the right hand side of the above inequality, we obtain a good approximation. We normally would replace A by $\{S,D\}$ to use the simplest instance. Identifying $\{S,D\}$ with **2** , we have

$$g^{\mathcal{B}}_C \sqsubseteq \bigsqcap H^e(h_u) \circ g^{\mathcal{B}}_{\mathbf{2}} \circ K^c(h_u)$$

where u ranges over all non-top elements of C.

Example

The following table is that of $append_2^B$, where append is defined in the usual way. The table is obtained by the usual iterative means. As in strictness analysis, this may be used in the analysis of `foldr` which in turn is used in the analysis of `concat`.

xs/ys	$SPINE(S)$	$SPINE(D)$	D
$SPINE(S)$	$SPINE(S)$	$SPINE(D)$	D
$SPINE(D)$	$SPINE(D)$	$SPINE(D)$	D
D	D	D	D

8.2 Termination Analysis

The property of functions that one usually looks for when performing termination analysis is dual to strictness. More precisely, for a function f we check if for all $x \neq \bot$ it is the case that $f(x) \neq \bot$. The techniques used in both analysis are, therefore, similar.

Mycroft has shown how abstract interpretation can be used for termination analysis [37]. He deals with first-order functions defined on flat domains. The abstract domains used in this interpretation are exactly those used for strictness analysis. However, in this case 1 represents termination, whereas 0 represents no information. Constants and the conditional are given the following interpretations.

- integer and boolean constants are interpreted by 1.
- the binary operators `+`,`-`,`*` and `/` are interpreted by $\sqcap$.
- the conditional `if` is interpreted by $if^{\mathcal{T}}$, where for a lattice A,
$$if_A^{\mathcal{T}} : \mathbf{2} \to A \to A \to A$$
is given by
$$if_A^{\mathcal{T}}\ a\ b\ c = \begin{cases} \bot_A & \text{if } a = 0 \\ b \sqcap c & \text{otherwise} \end{cases}$$
Again, we are assuming that A is the lattice that corresponds to the type of the appropriate instance of `if`.

We need not be specific about the lattice A above. In his extension of this interpretation to higher-order functions, Abramsky [4] uses these definitions and the

interpretation of other expressions is defined in a way similar to that for strictness analysis given in [11].

Polymorphism

We show that the abstract interpretation of a polymorphic function is a lax natural transformation between functors on the category of finite lattices and embedding-projection pairs, $\mathcal{A}^{ep}$. This is fairly straightforward; except perhaps that of the conditional defined above. Thus, we must consider the collection $\{if_A^{\mathcal{T}}\}$. Before doing this, notice that normally the $\sqcap$ used in the definition above should be indexed by A. We have actually shown, in Chapter 3, that the collection $\{\sqcap_A\}$ is a lax natural transformation between functors on the category $\mathcal{C}^{ep} \times \mathcal{C}^{ep}$.

Suppose A and B are finite lattices, and $k : A \to^{ep} B$ is any embedding-projection pair. Notice that this is of the form (k^e, k^p) and recall that $k^e \circ k^p \sqsubseteq id$ and $k^p \circ k^e = id$. Now, letting F and G be the functors given by

$$F(A) = \mathbf{2} \text{ and } G(A) = A \to A \to A$$

where $F(k)$ and $G(k)$ are defined as in Chapter 3, we want to show that the $\{if_A^{\mathcal{T}}\}$ is a lax natural transformation between F and G. To see this consider the following equations

$$\begin{aligned}
(if_B^{\mathcal{T}} \circ F^e(k))\ 0\ b\ c &= if_B^{\mathcal{T}}\ 0\ b\ c \\
&= \bot_B \\
(if_B^{\mathcal{T}} \circ F^e(k))\ 1\ b\ c &= if_B^{\mathcal{T}}\ 1\ b\ c \\
&= b \sqcap_B c
\end{aligned}$$

On the other hand,

$$\begin{aligned}
(G^e(k) \circ if_A^{\mathcal{T}})\ 0\ b\ c &= k^e(if_A^{\mathcal{T}}\ 0\ k^p(b)\ k^p(c)) \\
&= k^e(\bot_A) \\
&= \bot_B \qquad \text{(since } k^e \text{ is strict)}
\end{aligned}$$

Also,

$$\begin{aligned}
(G^e(k) \circ if_A^{\mathcal{T}})\ 1\ b\ c &= k^e(if_A^{\mathcal{T}}\ 1\ k^p(b)\ k^p(c)) \\
&= k^e(k^p(b) \sqcap_B k^p(c))
\end{aligned}$$

$$\begin{aligned} &= k^e(k^p(b \sqcap_A c)) \\ &\sqsubseteq b \sqcap_A c \end{aligned}$$

It is now trivial to observe that

$$G^e(k) \circ if_A^{\mathcal{T}} \sqsubseteq if_B^{\mathcal{T}} \circ F^e(k)$$

In general, the abstract interpretation of any polymorphic function is a lax natural transformation. Furthermore, if f is a lax natural transformation from some functor F to G then the function

$$\bigsqcup G^e(k) \circ f_2^{\mathcal{T}} \circ F^p(k)$$

may be used in place of $f_B^{\mathcal{T}}$ because if $x \neq \bot$ and $(\bigsqcup G^e(k) \circ f_2^{\mathcal{T}} \circ F^p(k))(x) \neq \bot$ then $f_B^{\mathcal{T}}(x) \neq \bot$.This why embedding-projection pairs are chosen for termination analysis.

Chapter 9

Conclusion

It has been known for a long time that instances of polymorphic functions are semantically related to each other. In this thesis we have seen two particular examples of relationships that are useful for program analysis. The choice of relationship depends on the semantic properties under investigation. In the case of strictness, for example, the relations that arise out of embedding-closure pairs are appropriate. One consequence of using these relations is a proof of a partial result about the semantic polymorphic invariance of strictness. Combining this with the results obtained from the use of embedding-projection pairs, we showed how the full result of polymorphic invariance can be proved. We have also stated and proved the appropriate invariance results for functions defined on lists.

The abstract interpretation of any polymorphic function was regarded as the standard interpretation of some polymorphic function in a language of abstract terms, where the semantic domains are finite lattices. This enabled us to use results from our study of the original language and its semantics; in particular, that the abstract interpretation of any polymorphic function is a lax natural transformation. It is interesting to note that the semantic polymorphic invariance of strictness in the language of abstract terms implies that the strictness analysis of the original language by abstract interpretation is polymorphically invariant.

We studied lax natural transformations on the category of finite lattices and embedding-closure pairs in some detail. This effort was rewarded by our ability

to build approximate values to the abstract interpretation of any instance of a polymorphic function from that of the smallest. This is the solution proposed to the central problem which was described at the beginning. It is the main contribution of this thesis. Seward has already built an efficient strictness analyser based on this method.

The lax natural transformation condition also holds in the presence of lists. However, the abstract domains used in the abstract interpretation of functions defined on lists are those of Nielson and Nielson. The disadvantage of this over the use of Wadler's abstract domains is that the domains are bigger. Consequently, in some cases the number of functions involved in building an approximation are by that much bigger, and hence making the process less efficient. On the other hand, there are several examples that arise in practice where the two domains are of the same size. In particular, this is so when the types being considered do not involve products.

Binding-time analysis was given less space in the thesis. However, every result that was proved about strictness analysis has a counterpart in binding-time analysis. This does not apply to functions defined on lists, because the abstract domains used in the analyses of lists are different. We have also shown that termination analysis can be treated in much the same way by using embedding-projection pairs.

It is worth recalling that the method of finding approximations in each analysis should only be applied in the case of recursive functions. In other cases, the cost of building approximations is likely to be more expensive than evaluating the actual value directly.

Because of their size, we would have liked to use Wadler's abstract domains in the strictness analysis of lists. But then it would be necessary to prove that $\{case^{\mathcal{S}}\}$ (Wadler's version) is a lax natural transformation. We have not managed to do this. More experiments are required to assess how the use of the domains by Nielson and Nielson affects the performance of strictness analysers.

Results about binding-time analysis must be comparable to those on strictness analysis. But, if the language is to include lists, separate experiments must be performed. However, it is not difficult to show that, binding-time analysis of lists is cheaper than

strictness analysis because of the size of the abstract domains used.

Appendix A

The Representation Theorem

Theorem 5.2.1

Let S_1 and S_2 be domain assignments. Let Γ be a type assignment. Let η_1 and η_2 be two environments such that $\eta_i(x) \in \mathcal{T}[\![\Gamma(x)]\!]\ S_i$ for all $x \in dom(\Gamma)$ and $i = 1, 2$. Suppose for each type variable t, we have an embedding-closure pair $\rho(t) : S_1(t) \rightarrow^{ec} S_2(t)$. Let h_τ stand for $\mathcal{T}[\![\tau]\!]\ \rho$.

If η_1 and η_2 are related, *i.e.*, for all $x \in dom(\Gamma)$

$$h^e_{\Gamma(x)}(\eta_1(x)) \ \sqsupseteq\ \eta_2(x),$$

then for each formula $\Gamma \vdash E : \tau$ we have

$$h^e_\tau(\mu_{\Gamma,\tau}[\![E]\!]\ S_1\ \eta_1) \ \sqsupseteq\ \mu_{\Gamma,\tau}[\![E]\!]\ S_2\ \eta_2$$

Proof

It is important to remember that both the condition and the statement of the theorem could equivalently be expressed in terms of the h^c's instead of the h^e's. That is, the statement is the same as

$$\mu_{\Gamma,\tau}[\![E]\!]\ S_1\ \eta_1 \ \sqsupseteq\ h^c_\tau(\mu_{\Gamma,\tau}[\![E]\!]\ S_2\ \eta_2)$$

The proof is now given by structural induction on terms.

(1) If k is a constant of type b then

$$\mu_{\Gamma,b}[\![k]\!]\ S_1\ \eta_1 \ = \ \mu_{\Gamma,b}[\![k]\!]\ S_2\ \eta_2$$

and since h^e_b is an identity we have the desired inequality.

(2) Since $\mu_{\Gamma,\Gamma(x)}[\![x]\!]\ S_i\ \eta_i = \eta_i(x)$ for $i = 1, 2$, the statement follows from the fact that $\eta_1(x)$ is related to $\eta_2(x)$ by hypothesis.

(3) Suppose $\Gamma \vdash E_1 : \tau \to \tau'$ and $\Gamma \vdash E_2 : \tau$

Now,

$$\begin{aligned}
& h^e_{\tau'}(\mu_{\Gamma,\tau'}[\![(E_1 E_2)]\!]\ S_1\ \eta_1) \\
& \qquad \text{[defn. of } \mu\text{]} \\
&= h^e_{\tau'}((\mu_{\Gamma,\tau\to\tau'}[\![E_1]\!]\ S_1\ \eta_1)(\mu_{\Gamma,\tau}[\![E_2]\!]\ S_1\ \eta_1)) \\
& \qquad \text{[inductive hypothesis]} \\
&\sqsupseteq h^e_{\tau'}((h^c_{\tau\to\tau'}(\mu_{\Gamma,\tau\to\tau'}[\![E_1]\!]\ S_2\ \eta_2))(h^c_\tau(\mu_{\Gamma,\tau}[\![E_2]\!]\ S_2\ \eta_2))) \\
& \qquad \text{[defn. of } h^c_{\tau\to\tau'}\text{]} \\
&= h^e_{\tau'}((h^c_{\tau'} \circ (\mu_{\Gamma,\tau\to\tau'}[\![E_1]\!]\ S_2\ \eta_2) \circ h^e_\tau)(h^c_\tau(\mu_{\Gamma,\tau}[\![E_2]\!]\ S_2\ \eta_2))) \\
&= h^e_{\tau'}(h^c_{\tau'}((\mu_{\Gamma,\tau\to\tau'}[\![E_1]\!]\ S_2\ \eta_2)(h^e_\tau(h^c_\tau(\mu_{\Gamma,\tau}[\![E_2]\!]\ S_2\ \eta_2))))) \\
& \qquad \text{[inductive hypothesis]} \\
&\sqsupseteq (\mu_{\Gamma,\tau\to\tau'}[\![E_1]\!]\ S_2\ \eta_2)(\mu_{\Gamma,\tau}[\![E_2]\!]\ S_2\ \eta_2) \\
&= \mu_{\Gamma,\tau'}[\![(E_1 E_2)]\!]\ S_2\ \eta_2
\end{aligned}$$

(4) Suppose $\Gamma, x : \tau \vdash E : \tau'$. Since η_1 and η_2 are related, for each $d' \in \mathcal{T}[\![\tau]\!]\ S_2$, $\eta_1[h^c_\tau(d')/x]$ and $\eta_2[d'/x]$ are related. This is because $h^e_\tau(h^c_\tau(d')) \sqsupseteq d'$. By induction assumption,

$$h^e_{\tau'}(\mu_{\Gamma,\tau'}[\![E]\!]\ S_1\ \eta_1[h^c_\tau(d')/x]) \ \sqsupseteq \ \mu_{\Gamma,\tau'}[\![E]\!]\ S_2\ \eta_2[d'/x]$$

Now,

$$h^e_{\tau\to\tau'}(\mu_{\Gamma,\tau\to\tau'}[\![\lambda x : \tau.E]\!]\ S_1\ \eta_1)$$

$$
\begin{aligned}
& \text{[defn. of } h^e_{\tau\to\tau'}\text{]} \\
= \;& h^e_{\tau'} \circ (\mu_{\Gamma,\tau\to\tau'}[\![\lambda x:\tau.E]\!]\ S_1\ \eta_1) \circ h^c_\tau \\
= \;& \lambda d' \in (\mathcal{T}[\![\tau]\!]\ S_2).h^e_{\tau'}(\mu_{\Gamma,\tau'}[\![E]\!]\ S_1\ \eta_1[h^c_\tau(d')/x]) \\
& \text{[inductive hypothesis]} \\
\sqsupseteq \;& \lambda d' \in (\mathcal{T}[\![\tau]\!]\ S_2).\mu_{\Gamma,\tau'}[\![E]\!]\ S_2\ \eta_2[d'/x] \\
= \;& \mu_{\Gamma,\tau\to\tau'}[\![\lambda x:\tau.E]\!]\ S_2\ \eta_2
\end{aligned}
$$

(5) Suppose $\Gamma \vdash < E_1, E_2 >: \tau \times \tau'$
By induction assumption,

$$
\begin{aligned}
h^e_\tau(\mu_{\Gamma,\tau}[\![E_1]\!]\ S_1\ \eta_1) &\sqsupseteq \mu_{\Gamma,\tau}[\![E_1]\!]\ S_2\ \eta_2 \quad \text{and} \\
h^e_\tau(\mu_{\Gamma,\tau'}[\![E_2]\!]\ S_1\ \eta_1) &\sqsupseteq \mu_{\Gamma,\tau'}[\![E_2]\!]\ S_2\ \eta_2
\end{aligned}
$$

Now,

$$
\begin{aligned}
& h^e_{\tau\times\tau'}(\mu_{\Gamma,\tau\times\tau'}[\![< E_1, E_2 >]\!]\ S_1\ \eta_1) \\
= \;& (h^e_\tau \times h^e_{\tau'})(\mu_{\Gamma,\tau}[\![E_1]\!]\ S_1\ \eta_1, \mu_{\Gamma,\tau'}[\![E_2]\!]\ S_1\ \eta_1) \\
= \;& (h^e_\tau(\mu_{\Gamma,\tau}[\![E_1]\!]\ S_1\ \eta_1), h^e_{\tau'}(\mu_{\Gamma,\tau'}[\![E_2]\!]\ S_1\ \eta_1)) \\
\sqsupseteq \;& ((\mu_{\Gamma,\tau}[\![E_1]\!]\ S_2\ \eta_2), (\mu_{\Gamma,\tau'}[\![E_2]\!]\ S_2\ \eta_2)) \\
= \;& \mu_{\Gamma,\tau\times\tau'}[\![< E_1, E_2 >]\!]\ S_2\ \eta_2
\end{aligned}
$$

(6) Suppose $\Gamma \vdash E : \tau \times \tau'$. We want to show that

$$
h^e_\tau(\mu_{\Gamma,\tau}[\![\mathtt{fst}\ \tau \times \tau'\ E]\!]\ S_1\ \eta_1) \sqsupseteq \mu_{\Gamma,\tau}[\![\mathtt{fst}\ \tau \times \tau'\ E]\!]\ S_2\ \eta_2
$$

By induction assumption we have :

$$
\mu_{\Gamma,\tau\times\tau'}[\![E]\!]\ S_1\ \eta_1 \sqsupseteq h^c_{\tau\times\tau'}(\mu_{\Gamma,\tau\times\tau'}[\![E]\!]\ S_2\ \eta_2)
$$

Now,

$$
\begin{aligned}
& h^e_\tau(\mu_{\Gamma,\tau}[\![\mathtt{fst}\ \tau \times \tau'\ E]\!]\ S_1\ \eta_1) \\
= \;& h^e_\tau(fst(\mu_{\Gamma,\tau\times\tau'}[\![E]\!]\ S_1\ \eta_1)) \\
\sqsupseteq \;& h^e_\tau(fst(h^c_{\tau\times\tau'}(\mu_{\Gamma,\tau\times\tau'}[\![E]\!]\ S_2\ \eta_2)))
\end{aligned}
$$

$$
\begin{aligned}
&= h^e_\tau(fst((h^c_\tau \times h^c_{\tau'})(\mu_{\Gamma,\tau\times\tau'}[\![E]\!]\ S_2\ \eta_2))) \\
&= h^e_\tau(fst(h^c_\tau(fst(\mu_{\Gamma,\tau\times\tau'}[\![E]\!]\ S_2\ \eta_2)), h^c_{\tau'}(snd(\mu_{\Gamma,\tau\times\tau'}[\![E]\!]\ S_2\ \eta_2)))) \\
&= h^e_\tau(h^c_\tau(fst(\mu_{\Gamma,\tau\times\tau'}[\![E]\!]\ S_2\ \eta_2))) \\
&\sqsupseteq fst(\mu_{\Gamma,\tau\times\tau'}[\![E]\!]\ S_2\ \eta_2) \\
&= \mu_{\Gamma,\tau}[\![\mathtt{fst}\ \tau\times\tau'\ E]\!]\ S_2\ \eta_2
\end{aligned}
$$

Similarly, we can show that a similar proposition holds for the term $\mathtt{snd}\ \tau \times \tau'\ E$.

(7) Suppose $\Gamma \vdash \mathtt{if}\ \tau\ E_0\ E_1\ E_2 : \tau$, and for $i = 1, 2$ assume that

$$h^e_\tau(\mu_{\Gamma,\tau}[\![E_i]\!]\ S_1\ \eta_1) \sqsupseteq \mu_{\Gamma,\tau}[\![E_i]\!]\ S_2\ \eta_2$$

and

$$h^e_{\mathbf{bool}}(\mu_{\Gamma,\mathbf{bool}}[\![E_0]\!]\ S_1\ \eta_1) \sqsupseteq \mu_{\Gamma,\mathbf{bool}}[\![E_0]\!]\ S_2\ \eta_2$$

Since $h^e_{\mathbf{bool}} = id$, at least one of the following holds :

$$
\begin{aligned}
\mu_{\Gamma,\mathbf{bool}}[\![E_0]\!]\ S_2\ \eta_2 &= \bot, \quad \text{or} \\
\mu_{\Gamma,\mathbf{bool}}[\![E_0]\!]\ S_1\ \eta_1 &= \mu_{\Gamma,\mathbf{bool}}[\![E_0]\!]\ S_2\ \eta_2
\end{aligned}
$$

In the former case,

$$\mu_{\Gamma,\tau}[\![\mathtt{if}\ \tau\ E_0\ E_1\ E_2]\!]\ S_2\ \eta_2 = \bot$$

and hence

$$h^e_\tau(\mu_{\Gamma,\tau}[\![\mathtt{if}\ \tau\ E_0\ E_1\ E_2]\!]\ S_1\ \eta_1) \sqsupseteq \mu_{\Gamma,\tau}[\![\mathtt{if}\ \tau\ E_0\ E_1\ E_2]\!]\ S_2\ \eta_2$$

The second case together with the induction assumption also yields the desired result.

(8)

$$h^e_\tau(\mu_{\Gamma,\tau}[\![\mathtt{fix}\ \tau\ E]\!]\ S_1\ \eta_1) = h^e_\tau(\bigsqcup_{n=0}^{\infty}(\mu_{\Gamma,\tau\to\tau}[\![E]\!]\ S_1\ \eta_1)^n(\bot))$$

$$
\begin{aligned}
& \text{[continuity of } h^e_\tau\text{]} \\
\sqsupseteq\ & \bigsqcup_{n=0}^{\infty} h^e_\tau((\mu_{\Gamma,\tau\to\tau}[\![E]\!]\ S_1\ \eta_1)^n(\bot)) \\
& \text{[inductive hypothesis]} \\
\sqsupseteq\ & \bigsqcup_{n=0}^{\infty} h^e_\tau((h^c_{\tau\to\tau}(\mu_{\Gamma,\tau\to\tau}[\![E]\!]\ S_2\ \eta_2))^n(\bot)) \\
& \text{[defn. of } h^c_{\tau\to\tau}\text{]} \\
=\ & \bigsqcup_{n=0}^{\infty} h^e_\tau((h^c_\tau \circ (\mu_{\Gamma,\tau\to\tau}[\![E]\!]\ S_2\ \eta_2) \circ h^e_\tau)^n(\bot)) \\
& [h^e_\tau \circ h^c_\tau \sqsupseteq id] \\
\sqsupseteq\ & \bigsqcup_{n=0}^{\infty} h^e_\tau((h^c_\tau \circ (\mu_{\Gamma,\tau\to\tau}[\![E]\!]\ S_2\ \eta_2)^n \circ h^e_\tau)(\bot)) \\
\sqsupseteq\ & \bigsqcup_{n=0}^{\infty} (\mu_{\Gamma,\tau\to\tau}[\![E]\!]\ S_2\ \eta_2)^n(\bot) \\
=\ & \mu_{\Gamma,\tau}[\![\mathtt{fix}\,\tau\ E]\!]\ S_2\ \eta_2
\end{aligned}
$$

(9) Suppose $\Gamma \vdash \mathtt{let}\ x : \sigma = E_1\ \mathtt{in}\ E_2 : \tau$. Since the semantics of this is defined in terms of that of the term $E_2[E_1/x]$, for the purposes of this proof we may assume that we do not have `let`-expressions. We make similar assumptions about instantiations.

Appendix B

The Safety Condition

Proposition 6.4.1

Suppose we have the formula $\Gamma \vdash E : \tau$ and that S is a domain assignment which assigns type variables domains from $\mathcal{D}$. Let η be an environment. Let S' be the lattice assignment such that for any type variable t, $S'(t) = Abs\ S(t)$.

If η' is any environment such that for any variable $x \in dom(\Gamma), abs_{\Gamma(x)}(\eta(x)) \sqsubseteq \eta'(x)$, then

$$abs_\tau(\mu_{\Gamma,\tau}[\![E]\!]\ S\ \eta) \quad \sqsubseteq \quad \nu_{\Gamma',\ Abs\ \tau}[\![E']\!]\ S'\ \eta'$$

where $\Gamma' \vdash E' : \tau'$ is the translation of the above formula.

Proof

The proof is given by induction on the structure of terms.

(1) For every basic type b, by definition of $\nu_{\Gamma',Abs\ b}$ the two expressions are actually equal.

(2) If $x \in dom(\Gamma)$ then

$$abs_{\Gamma(x)}(\mu_{\Gamma,\Gamma(x)}[\![x]\!]\ S\ \eta) \quad = \quad abs_{\Gamma(x)}(\eta(x))$$

By assumption on the environments, the statement follows.

(3) If $E_1 \in A_{\Gamma,\tau\to\tau'}$ and $E_2 \in A_{\Gamma,\tau}$ then

$$\begin{aligned}
&abs_{\tau'}(\mu_{\Gamma,\tau'}[\![(E_1E_2)]\!]\ S\ \eta)\\
&\quad= abs_{\tau'}((\mu_{\Gamma,\tau\to\tau'}[\![E_1]\!]\ S\ \eta)(\mu_{\Gamma,\tau}[\![E_2]\!]\ S\ \eta))\\
&\qquad\text{[property of } abs\text{]}\\
&\quad\sqsubseteq (abs_{\tau\to\tau'}(\mu_{\Gamma,\tau'}[\![E_1E_2]\!]\ S\ \eta))(abs_{\tau}(\mu_{\Gamma,\tau}[\![E_2]\!]\ S\ \eta))\\
&\qquad\text{inductive hypothesis}\\
&\quad\sqsubseteq (\nu_{\Gamma',\ Abs\ (\tau\to\tau')}[\![E_1']\!]\ S'\ \eta')(\nu_{\Gamma',\ Abs\ \tau}[\![E_2']\!]\ S'\ \eta')\\
&\quad= \nu_{\Gamma',\ Abs\ \tau'}[\![(E_1'E_2')]\!]\ S'\ \eta'
\end{aligned}$$

(4) Suppose $E \in A_{(\Gamma,x:\tau),\tau'}$. We want to show that

$$\begin{aligned}
&abs_{\tau\to\tau'}(\mu_{\Gamma,\tau\to\tau'}[\![\lambda x:\tau.E]\!]\ S\ \eta)\\
&\quad\sqsubseteq\ \nu_{\Gamma',\ (Abs\ \tau)\to(Abs\ \tau')}[\![\lambda x:(Abs\ \tau).E']\!]\ S'\ \eta'
\end{aligned}$$

That is, for any value b from the appropriate lattice

$$\begin{aligned}
&(abs_{\tau\to\tau'}(\mu_{\Gamma,\tau\to\tau'}[\![\lambda x:\tau.E]\!]\ S\ \eta))(b)\\
&\quad\sqsubseteq\ (\nu_{\Gamma',\ (Abs\ \tau)\to(Abs\ \tau')}[\![\lambda x:(Abs\ \tau).E']\!]\ S'\ \eta')(b)
\end{aligned}$$

But

$$\begin{aligned}
&(abs_{\tau\to\tau'}(\mu_{\Gamma,\tau\to\tau'}[\![\lambda x:\tau.E]\!]\ S\ \eta))(b)\\
&\quad= \bigsqcup\{abs_{\tau'}((\mu_{\Gamma,\tau\to\tau'}[\![\lambda x:\tau.E]\!]\ S\ \eta)(a))\mid abs_\tau(a)\sqsubseteq b\}
\end{aligned}$$

Thus, it is sufficient to show that for an arbitrary element a with $abs_\tau(a)\sqsubseteq b$

$$\begin{aligned}
&abs_{\tau'}((\mu_{\Gamma,\tau\to\tau'}[\![\lambda x:\tau.E]\!]\ S\ \eta)(a))\\
&\quad\sqsubseteq\ (\nu_{\Gamma',\ (Abs\ \tau)\to(Abs\ \tau')}[\![\lambda x:(Abs\ \tau).E']\!]\ S'\ \eta')(b)
\end{aligned}$$

But

$$\begin{aligned}
&abs_{\tau'}((\mu_{\Gamma,\tau\to\tau'}[\![\lambda x:\tau.E]\!]\ S\ \eta)(a))\\
&\quad= abs_{\tau'}(\mu_{(\Gamma,x:\tau),\ \tau'}[\![E]\!]\ S\ \eta[a/x])\\
&\qquad\text{[inductive hypothesis]}
\end{aligned}$$

$$
\begin{aligned}
&\sqsubseteq \nu_{(\Gamma', x:(Abs\ \tau)), Abs\ \tau'}[\![E']\!]\ S'\ (\eta'[abs_\tau(a)/x]) \\
&\quad \text{[monotonicity]} \\
&\sqsubseteq \nu_{(\Gamma', x:(Abs\ \tau)), Abs\ \tau'}[\![E']\!]\ S'\ (\eta'[b/x]) \\
&= (\nu_{\Gamma', (Abs\ \tau)\to(Abs\ \tau')}[\![\lambda x : (Abs\ \tau).E']\!]\ S'\ \eta')(b)
\end{aligned}
$$

Therefore, we have the desired result.

(5) If $E_1 \in A_{\Gamma,\tau}$ and $E_2 \in A_{\Gamma,\tau'}$ then

$$
\begin{aligned}
&abs_{\tau\times\tau'}(\mu_{\Gamma,\tau\times\tau'}[\![< E_1, E_2 >]\!]\ S\ \eta) \\
&\quad \text{[defn. of } \mu\text{]} \\
&= abs_{\tau\times\tau'}(\mu_{\Gamma,\tau}[\![E_1]\!]\ S\ \eta, \mu_{\Gamma,\tau'}[\![E_2]\!]\ S\ \eta) \\
&\quad \text{[defn. of } abs\text{]} \\
&= (abs_\tau(\mu_{\Gamma,\tau}[\![E_1]\!]\ S\ \eta), abs_{\tau'}(\mu_{\Gamma,\tau'}[\![E_2]\!]\ S\ \eta)) \\
&\quad \text{[inductive hypothesis]} \\
&\sqsubseteq (\nu_{\Gamma', Abs\ \tau}[\![E_1']\!]\ S'\ \eta', \nu_{\Gamma, Abs\ \tau'}[\![E_2']\!]\ S'\ \eta') \\
&= \nu_{\Gamma', (Abs\ \tau\times\tau')}[\![< E_1', E_2' >]\!]\ S'\ \eta'
\end{aligned}
$$

(6) If $E \in A_{\Gamma,\tau\times\tau'}$ then

$$
\begin{aligned}
abs_\tau(\mu_{\Gamma,\tau}[\![\mathtt{fst}\ \tau\times\tau'\ E]\!]\ S\ \eta) &= abs_\tau(fst(\mu_{\Gamma,\tau\times\tau'}[\![E]\!]\ S\ \eta)) \\
&= fst(abs_{\tau\times\tau'}(\mu_{\Gamma,\tau'}[\![E]\!]\ S\ \eta)) \\
&\sqsubseteq fst(\nu_{\Gamma', (Abs\ \tau)\times(Abs\ \tau')}[\![E']\!]\ S'\ \eta') \\
&= \nu_{\Gamma', Abs\ \tau}[\![\mathtt{fst}\ \tau\times\tau'\ E']\!]\ S'\ \eta'
\end{aligned}
$$

The case of $\mathtt{snd}\ \tau\times\tau'\ E$ can be proved in a similar way.

(7) Suppose we have the formula $\Gamma \vdash \mathtt{if}\ \tau\ E_0\ E_1\ E_2 : \tau$. From this we have $\Gamma' \vdash \mathtt{cond}\ Abs\ \tau\ (\mathtt{or}\ Abs\ \tau\ E_1'\ E_2') : Abs\ \tau$. There are now two cases to consider.

case(i)

$$\mu_{\Gamma,\,\mathbf{bool}}[\![E_0]\!]\ S\ \eta\ =\ \bot$$

In this case

$$\mu_{\Gamma,\tau}[\![\texttt{if}\ \tau\ E_0\ E_1\ E_2]\!]\ S\ \eta\ =\ \bot$$

Since abs_τ is strict,

$$abs_\tau(\mu_{\Gamma,\tau}[\![\texttt{if}\ \tau\ E_0\ E_1\ E_2]\!]\ S\ \eta)\ =\ \bot$$

Therefore, we have the inequality.

case(ii)

$$\mu_{\Gamma,\,\mathbf{bool}}[\![E_0]\!]\ S\ \eta\ \neq\ \bot$$

Since $abs_{\mathbf{bool}}$ is $\bot$-reflecting

$$abs_{\mathbf{bool}}(\mu_{\Gamma,\,\mathbf{bool}}[\![E_0]\!]\ S\ \eta)\ \neq\ 0$$

By induction assumption,

$$abs_{\mathbf{bool}}(\mu_{\Gamma,\,\mathbf{bool}}[\![E_0]\!]\ S\ \eta)\ \sqsubseteq\ \nu_{\Gamma',\,\mathbf{2}}[\![E_0']\!]\ S'\ \eta'$$

Hence,

$$\nu_{\Gamma',\,\mathbf{2}}[\![E_0']\!]\ S'\ \eta'\ =\ 1$$

Therefore,

$$\nu_{\Gamma',\,Abs\,\tau}[\![\texttt{cond}\ Abs\ \tau\ E_0'\ (\texttt{or}\ E_1'\ E_2')]\!]\ S'\ \eta' = \sqcup_A(v_1, v_2)$$

where for $i = 1, 2$

$$\begin{aligned} v_i\ &=\ \nu_{\Gamma',\,Abs\,\tau}[\![E_i']\!]\ S'\ \eta', \quad \textit{and} \\ A\ &=\ \mathcal{T}'[\![Abs\ \tau]\!]\ S' \end{aligned}$$

Again by induction, for $i = 1, 2$ we have

$$abs_\tau(\mu_{\Gamma,\,\tau}[\![E_i]\!]\ S\ \eta)\ \sqsubseteq\ \nu_{\Gamma',\,Abs\,\tau}[\![E_i']\!]\ S'\ \eta'$$

Clearly this implies that for $i = 1, 2$

$$abs_\tau(\mu_{\Gamma,\tau}[\![E_i]\!]\ S\ \eta)\ \sqsubseteq\ \sqcup_A(v_1, v_2)$$

where v_1, v_2 and A are as defined above. But in the case we are considering,

$$\begin{aligned}\mu_{\Gamma,\tau}[\![\texttt{if}\ \tau\ E_0\ E_1\ E_2]\!]\ S\ \eta &= \mu_{\Gamma,\tau}[\![E_1]\!]\ S\ \eta \quad \text{or}\\ \mu_{\Gamma,\tau}[\![\texttt{if}\ \tau\ E_0\ E_1\ E_2]\!]\ S\ \eta &= \mu_{\Gamma,\tau}[\![E_2]\!]\ S\ \eta\end{aligned}$$

It is now obvious from this that the required inequality holds.

(8)

$$\begin{aligned}&abs_\tau(\mu_{\Gamma,\tau}[\![\texttt{fix}\ \tau\ E]\!]\ S\ \eta)\\ &= abs_\tau(\bigsqcup_{n=0}^{\infty}(\mu_{\Gamma,\tau\to\tau}[\![E]\!]\ S\ \eta)^n(\bot))\\ &\quad [\text{continuity of } abs]\\ &= \bigsqcup_{n=0}^{\infty} abs_\tau((\mu_{\Gamma,\tau\to\tau}[\![E]\!]\ S\ \eta)^n(\bot))\\ &\quad [\text{property of } abs]\\ &\sqsubseteq \bigsqcup_{n=0}^{\infty} abs_{\tau\to\tau}((\mu_{\Gamma,\tau\to\tau}[\![E]\!]\ S\ \eta)^n)(abs_\tau(\bot))\\ &\quad [\text{property of } abs]\\ &\sqsubseteq \bigsqcup_{n=0}^{\infty}(abs_{\tau\to\tau}(\mu_{\Gamma,\tau\to\tau}[\![E]\!]\ S\ \eta))^n(abs_\tau(\bot))\\ &\quad [abs \text{ is strict}]\\ &= \bigsqcup_{n=0}^{\infty}(abs_{\tau\to\tau}(\mu_{\Gamma,\tau\to\tau}[\![E]\!]\ S\ \eta))^n(\bot)\\ &\quad [\text{inductive hypothesis}]\\ &\sqsubseteq \bigsqcup_{n=0}^{\infty}(\nu_{\Gamma',Abs\ \tau\to Abs\ \tau}[\![E']\!]\ S'\ \eta')^n(\bot)\\ &= \nu_{\Gamma',Abs\ \tau}[\![\texttt{fix}\ Abs\ \tau\ E']\!]\ S'\ \eta'\end{aligned}$$

(9) Again, we may assume that we do not have `let`-expressions and expressions of the form $E[\tau']$.

Bibliography

[1] S. Abramsky. Strictness Analysis and Polymorphic Invariance. In *Programs as Data Objects*, LNCS 217 (1985), 1-23.

[2] S. Abramsky. Notes on Strictness Analysis for Polymorphic Functions. Draft paper (1988).

[3] S. Abramsky and C.L. Hankin (eds). *Abstract Interpretation of Declarative Languages.* Ellis-Horwood (1987).

[4] S. Abramsky. Abstract Interpretation, Logical Relations and Kan Extensions. *Journal of Logic and Computation*, 1(1) (1990), 5-39.

[5] S. Abramsky and T. Jensen. A Relational Approach to Strictness Analysis for Higher-Order Polymorphic Functions. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, (1991), 49-54.

[6] G. Baraki. A Note on Abstract Interpretation of Polymorphic Functions. In *Functional Programming and Computer Architecture, 5th Conference*, LNCS 523 (1991), 367-378.

[7] P.N. Benton. Strictness Logic and Polymorphic Invariance. In *Logical Foundations of Computer Science, Second International Symposium*, LNCS 620 (1992), 32-44.

[8] G.L. Burn. An Issue to do with Strictness Analysis of Polymorphically Typed Programs. Draft paper (1985).

[9] G.L. Burn. A Relationship Between Abstract Interpretation and Projection Analysis (Extended Abstract). In *POPL 90, Conference on Principles of Programming Languages*, ACM (1990), 151-156.

[10] G.L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation.* Pitman (1991).

[11] G.L. Burn, C.L. Hankin and S. Abramsky. Strictness Analysis for Higher-Order Functions. *Science of Computer Programming*, 7 (1986), 249-278.

[12] C. Clack and S. Peyton-Jones. Strictness Analysis - a practical approach. In *Functional Programming and Computer Architecture*, LNCS 201 (1985), 35-49.

[13] C. Clack and S. Peyton-Jones. Generating Parallelism from Strictness Analysis. In *Proceedings of the Workshop on Implementation of Functional Languages*, Programming Methodology Group, University of Goteborg and Chalmers University of Technology, Report 17 (1985), 92-131.

[14] T. Coquand. An Introduction to Type Theory. Draft paper (1989).

[15] T. Coquand, C. Gunter, and G. Winskel. Domain Theoretic Models of Polymorphism. Technical Report 116, University of Cambridge (1987).

[16] Y. Fuh and P. Mishra. Type Inference with Subtypes. In *ESOP 88*, LNCS 300 (1988), 94-114.

[17] J.-Y. Girard. *Interprétation Fonctionelle et Elimination des Coupures dans l'Arithmétique d'Ordre Supérieure.* PhD thesis, Universite Paris VII, 1972.

[18] P. Hudak et al. *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language*, March 1992. Version 1.2. Also in Sigplan Notices, May 1992.

[19] R.J.M. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2) (1989), 98-107.

[20] R.J.M. Hughes. Strictness Analysis in Non-flat Domains. In *Programs as Data Objects*, LNCS 217 (1985), 112-135.

[21] R.J.M. Hughes. Abstract Interpretation of First-Order Polymorphic Functions. In *Proceedings of the 1988 Glasgow Workshop on Functional Programming*, Research Report 89/R4, University of Glasgow (1989).

[22] R.J.M. Hughes. Backwards Analysis of Functional Programs. Research Report 87/R3, University of Glasgow (1987).

[23] R.J.M. Hughes and J. Launchbury. Projections for Polymorphic Strictness Analysis. Research Report 90/R33, University of Glasgow (1990).

[24] S. Hunt. Frontiers and Open Sets in Abstract Interpretation. In *FPCA'89 Conference Proceedings, London*, (1989), 1-11.

[25] S. Hunt and C. Hankin. Fixed Points and Frontiers: A New Perspective. *Journal of Functional Programming*, 1(1) (1991), 91-120.

[26] S. Hunt and D. Sands. Binding Time Analysis: A New PERspective. *SIGPLAN Notices*, 26(9) (1991), 154-165.

[27] S. Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice.* Ph.D. Thesis, Imperial College (1991).

[28] T.P. Jensen. Strictness Analysis in Logical Form. In *Functional Programming and Computer Architecture, 5th Conference*, LNCS 523 (1991), 352-365.

[29] T.P. Jensen. Disjunctive Strictness. In *Logic in Computer Science, Proceedings of the 7th Symposium*, Computer Society Press of the IEEE (1992), 174-185.

[30] T.P. Jensen. *Abstract Interpretation in Logical Form.* Ph.D. Thesis, Imperial College (1992).

[31] S. Kamin. Head-strictness is not a Monotonic Abstract Property. *Information Processing Letters*, 41 (1992), 195-198.

[32] T. Kuo and P. Mishra. Strictness Analysis : A New Perspective based on Type Inference. In *FPCA'89 Conference Proceedings, London*, (1989), 260-272.

[33] J. Launchbury. *Projection Factorisations in Partial Evaluation.* Distinguished Dissertation in Computer Science, Vol 1 (1991), CUP.

[34] C. Martin and C.L. Hankin. Finding Fixed Points in Finite Lattices. In *Functional Programming and Computer Architecture*, LNCS 274 (1987), 426-445.

[35] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Science*, 17 (1978), 348-375.

[36] J.C. Mitchell and R. Harper. The Essence of ML. In *POPL'88*, (1988), 28-46.

[37] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* Ph.D. Thesis, University of Edinburgh (1981).

[38] F. Nielson and H.R. Nielson. The Tensor Product in Wadler's Analysis of Lists. In *4th Symposium on Programming, Rennes, France. Proceedings.* LNCS 582 (1992), 351-370.

[39] A. Ohori. A Simple Semantics for ML Polymorphism. In *FPCA'89 Conference Proceedings, London*, (1989), 281-292.

[40] G.D. Plotkin. Lambda-definability in the Full Type Hierarchy. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press (1980), 363-373.

[41] G.D. Plotkin. *Lecture Notes in Domain Theory.* University of Edinburgh (1981).

[42] J.C. Reynolds. Towards a Theory of Type Structure. In B. Robinet, editor, *Proc. Colloque sur la Programmation*, LNCS 19 (1974), 408-425.

[43] J.C. Reynolds. Types, Abstraction and Parametric Polymorphism. In *Information Processing 83.* North Holland (1983), 513-523.

[44] J.C. Reynolds. Polymorphism is not Set-theoretic. In Kahn, MacQueen, and Plotkin, editors, *Semantics of Data Types, Sophia-Antipolis, France*, LNCS 173 (1984), 145-156.

[45] D. Scott. Data Types as Lattices. *SIAM Journal of Computing*, 5(3) (1976), 522-587.

[46] P. Sestoft. The Structure of a Self-Applicable Partial Evaluator. In *Programs as Data Objects*, LNCS 217 (1985), 236-256.

[47] J. Seward. *Personal Communication.*

[48] J. Seward. Polymorphic Strictness Analysis. Draft paper (1992).

[49] A. Tarski. A Lattice-Theoretic Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, 5 (1955), 285-309.

[50] D.A. Turner. Miranda: A Non-Strict Functional Language with Polymorphic Types. In *FPCA'85 , Nancy, France*, LNCS 201 (1985), 1-16.

[51] P. Wadler and R.J.M. Hughes. Projections for Strictness Analysis. In *FPCA'87, Portland, Oregon*, LNCS 274 (1987), 385-407.

[52] P. Wadler. Strictness Analysis on Non-Flat Domains by Abstract Interpretation. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages.* Ellis-Horwood (1987), 266-275.

[53] P. Wadler. Theorems for Free! In *FPCA'89 Conference Proceedings, London*, (1989), 347-359.