Static Analysis of Functional Languages

Submitted in fulfilment of the requirements
for the Degree of Master of Science

Rhodes University

Jon-Dean Mountjoy

January 1994

Abstract

Static analysis is the name given to a number of compile time analysis techniques used to automatically generate information which can lead to improvements in the execution performance of function languages. This thesis provides an introduction to these techniques and their implementation.

The abstract interpretation framework is an example of a technique used to extract information from a program by providing the program with an alternate semantics and evaluating this program over a non-standard domain. The elements of this domain represent certain properties of interest. This framework is examined in detail, as well as various extensions and variants of it. The use of binary logical relations and program logics as alternative formulations of the framework, and partial equivalence relations as an extension to it, are also looked at.

The projection analysis framework determines how much of a sub-expression can be evaluated by examining the context in which the expression is to be evaluated, and provides an elegant method for finding particular types of information from data structures. This is also examined.

The most costly operation in implementing an analysis is the computation of fixed points. Methods developed to make this process more efficient are looked at.

This leads to the final chapter which highlights the dependencies and relationships between the different frameworks and their mathematical disciplines.

Acknowledgements

I would like to thank Pete Wentworth for supervising this project and providing much guidance. I am also grateful to John Ebden for reading and commenting on a draft of this thesis, and Jim Chadwick for answering many of my questions. Finally, I thank Nicola for everything else.

Contents

Chapter 1

Introduction

Since the conception of abstract interpretation twelve years ago, a considerable amount of research has been taking place in the field of analysis techniques for functional languages. Almost all implementations of lazy functional languages are termed "naïve" if they do not embody some form of analysis; and in addition, they almost certainly suffer from severe efficiency problems.

The simple and clean semantics of functional languages has prompted the rigorous development of powerful and complex techniques for the extraction of information from a program. This development has necessitated the construction of rigorous mathematical models, drawing from many branches of mathematics. Domain theory, category theory, logic and equivalence relations are only a few of the tools used in the construction of analysis frameworks. Strictness analysis, binding time analysis, evaluation order analysis and update analysis are only a handful of the applications developed using these frameworks.

We begin this chapter by first examining why program analysis techniques are needed, motivating this by claiming that abstraction is both the cause of, and solution to performance problems. We then introduce analysis techniques and frameworks, concluding with the objectives and an overview of the thesis.

1.1 Program Analysis Matters[1]

The purpose of this section is to motivate the need for analysis techniques for functional languages. Many needs motivate such analysis, the most common of which is the need to improve performance. Most functional languages have inferior performance in comparison with their imperative counterparts[2], and so analysis techniques need to be developed to counter this.

We believe that this performance issue is not the core reason for the need of analysis techniques. There is something more fundamental than this need, and that is the cost of abstraction.

1.1.1 The Cost of Abstraction

John Backus writes on the benefits of using a functional paradigm over an imperative one: "Unlike von Neumann languages, these systems have semantics loosely coupled to states – only one state transition occurs per major computation." [Bac78] A severe price has to be paid for this abstraction. Functional languages abstract away many features of a von Neumann architecture, including memory management and explicit execution order, enabling us to write programs at a higher level. As Vegdahl [Veg84] writes, this ability "may be a blessing during programming, but a curse during execution". The languages are so far removed from the architectures on which they are ultimately implemented that special architectures have been proposed to execute functional languages [Veg84]. These architectures include physical architectures and abstract machines [Lan63, Tur79, BPR88].

A key feature of a functional language, which results in much of its abstraction, is that of *referential transparency*, which Stoy [Sto77] defines as:

> "The only thing that matters about an expression is its value, and any subexpression can be replaced by any other equal in value. Moreover, the value of an expression is, within certain limits, the same wherever it occurs."

The consequences of this are manifold.

[1] This is the title of Gomard's thesis [GS91].

[2] The introduction of static analysis has now increased the performance dramatically.

- An expression has the same value in any context, implying that "the value of an expression is independent of the history of the computation." [Tri90] Thus the order of evaluation of expressions is no longer fixed, and we have freed the language of any specific execution order.

- Parallelism and laziness are now viable execution options. Freedom of execution order allows us to find much parallelism [Bur87b, Gol88, HKL91] in a functional language, and this is often exploited and hailed as yet another benefit of using functional languages. It also allows us to choose the (less efficient) call-by-need reduction order which is needed for laziness. Indeed, many of the analyses are used to determine when the lazy evaluation order of a lazy language can 'safely' be converted to an eager evaluation order.

- There are no side effects. Allowing an expression to have a side effect could destroy our referential transparency. A variable is a simple expression, and allowing assignment would allow an expression to take on different values, destroying referential transparency. A consequence of this is that the hitherto ubiquitous assignment is banished from our language.

An implication of the last point is that of non-destructive updates. If we have an array and wish to change a particular element in the array, we have to copy the entire array and give this new object a different name, preserving referential transparency. Abstracting away memory leads to the need of time-consuming garbage collection, and lazy evaluation leads to a great overhead in the storing of closures.

All of these issues obviously impact on run-time efficiency.

1.1.2 The Benefits of Staying Pure

However, abstraction also aids us in finding the solution to our problems.

The functional language paradigm provides mechanisms not found in conventional languages which allow us to reap the benefits of abstraction at a much lower cost. Referential transparency itself, the very source of implementation inefficiency, provides features which aid us in the optimization of the language.

In mathematics, we view functions as being referentially transparent; any function denotes a value, and we can replace it with any expression equal in value. Since functional languages share this property, this forms the basis for using mathematical reasoning in proving properties about programs. We are now able to construct a simple algebra of programs, enabling an optimizing compiler to transform and reduce a program to a more efficient version, in a provably correct manner.

This property also filters down to the denotational description of the language, which is simpler than that of an imperative language. Although many optimizations are performed at the language level itself, many of the analyses described in this thesis mostly focus on the denotational description. Using denotational semantics [Sco76, Sco82, GS80] to model the semantics of the functional language, we obtain a firm mathematical basis. Analyses can now be built at this denotational level, and proved correct by using the mathematics of denotational semantics, namely domain theory.

1.1.3 Utilizing the Formal Basis

The primary goal of static analysis is to "obtain as much information as possible about a program's possible run time behavior without actually having to run it on all input data; and to do this automatically." [JN90]. In their seminal works, the Cousots [CC79, CC77] developed a method of abstract interpretation to analyse programs, and although their techniques applied to data-flow languages, they provided the basis upon which Mycroft [Myc81] constructed a rigorous framework for the abstract interpretation of *functional* languages.

We will now consider an example in which some uses of analysis are introduced.

Consider the following fragment of code:

```
fact x = 1                  if x==0,
         x * fact (x-1)     otherwise
```

In a lazy functional language[3], the actual parameter of the function `fact` would be passed as a suspension every time that it is called. This closure mechanism allows for

[3]Most of this thesis addresses analysis techniques for *lazy* functional languages. ie. languages which form suspensions (closures, thunks) of the code and environment, so as to implement the call-by-need mechanism. *Eager* languages implement the call-by-value mechanism.

the implementation of laziness, yet produces a lot of run-time overhead. An obvious step in the optimization of such a language, would be to try and eliminate the creation of the closures, and perform the computation in a manner like that of an eager language. A question we then ask is: "When can the evaluation of a function be changed from a lazy manner, to an eager manner, without changing the meaning of the function". Obviously, if we changed the evaluation order to an eager method and an expression did not terminate, we might have changed the run-time behavior, as a lazy language might not have executed that non-terminating expression at all.

A motivating analysis which will be used throughout this thesis is strictness analysis, which allows us to perform just those optimizations that are hinted at above.

Strictness analysis is one of the most heavily researched analysis techniques. It is this analysis which detects whether an expression is going to be needed in a computation. If we know that an expression is going to be needed, we are sure that it is going to be evaluated at some time using a lazy strategy, and so evaluating it eagerly will not change the operational semantics of the program. To rephrase this, if we know that an expression is going to be needed in a computation, and if in computing this expression eagerly we introduce a non-termination, we have not changed the operational semantics as the lazy evaluation scheme would not have terminated either.

In the abstract interpretation framework, we would perform a strictness analysis by manipulating the denotational semantics meaning of a program and proving that any transformations suggested would not change the meaning of our program.

1.2 Analysis Frameworks and Disciplines

Abramsky [Abr90] writes of program analysis:

> "The task of a theoretical framework for this technique is to provide appropriate concepts to formalize the notion of *correctness* of an analysis, and mathematical tools and theories to support *proofs* of correctness of particular analyses."

To highlight some of the differences between analyses and the mathematics that implements and supports them, we will look at some of the key concepts behind an analysis

and also at the disciplines used in their description. We call the former analysis frameworks, and the latter analysis disciplines.

1.2.1 Analysis Disciplines

As expressed above, there are various mathematical underpinnings to each analysis framework. Initially only the mathematics behind denotational semantics was used, namely domain theory.

As the analyses developed, it was found that the underlying mathematics had to be changed in order to capture different properties of a program. As an example, an initial change to Mycroft's theory was to use different powerdomain constructs, using the Hoare powerdomain instead of the Plotkin powerdomain. This enabled the theory to capture the property of 'definitely will not terminate' as opposed to 'definitely will terminate.'

The mathematics describing an analysis was also 'rephrased' in other formalisms, such as relations and logic. As researchers tried to capture even more properties, the mathematical frameworks were developed, some using other features of domain theory such as projections, others being described in a completely different manner using partial equivalence relations. Because analyses also address typed functional languages and there is a tight relationship between type theory and category theory, this was also used.

Finding fixed points in lattices is essential if an analysis is to operate on recursive function definitions. On an implementation side, finding fixed points in lattices is computationally very expensive, and so research has been devoted to lattice theory to find more efficient ways of computing these fixed points.

We will use the words 'analysis discipline' to mean the mathematical model which is used in the construction of an analysis.

The various theories developed to describe analyses have found roots in many mathematical disciplines, and in this thesis we hope to expose some of the different mathematics needed behind the analysis frameworks.

1.2.2 Analysis Frameworks

Various analysis frameworks have been developed utilizing different mathematical disciplines. A framework essentially provides the tools for proving the correctness of a particular analysis, and is often specialized to the analysis. A more formal definition of correctness is given in a later chapter, but an analysis can be thought of as 'correct' if the information that the analysis yields is not in contradiction to the actual run-time behavior. The most heavily researched analysis frameworks, namely abstract interpretation and projection analysis, are presented in this thesis.

1.3 The Objectives of this Thesis

This thesis endeavours to provide an overview and taxonomy of the static analysis techniques that are currently available for functional languages. Many papers have been written introducing new analysis frameworks, and many more on their development and extensions. We hope to provide the reader with at least an insight into these.

Those frameworks that have received the greatest attention, namely abstract interpretation and projection analysis, are examined; abstract interpretation especially since research in this field has been taking place for many more years than in any other.

Another objective is to expose the different mathematical disciplines that have been used in the description of the various frameworks, providing pointers to material that should be read if a particular topic is to be examined in depth. Rather than concentrating on the mathematical proof of all of the results, the concepts and intuitions behind the material have been emphasized.

A secondary goal is to explore several relationships that have been found between abstract interpretation and projection analysis, and also to look briefly at developments that have been made to improve the efficiency of various frameworks by improving fixed point techniques.

Many of the results presented in this thesis are formal by nature, thus some prerequisite mathematics is needed. Elementary domain theory and denotational semantics (as found in [Sto77, Sch86, GS80]) is assumed, together with various other mathematical constructs, a summary of which appears in the following section.

1.4 Notation and Terminology

Definition 1.1 (Partial Ordered Set (Poset)) *A* partially ordered *set is a pair* $(P, \sqsubseteq_P)$*, with* P *a set (the carrier) and* $\sqsubseteq_P$ *a relation on* P *satisfying:*

$$\begin{aligned} &d \sqsubseteq_P d && (reflexivity)\\ &d_1 \sqsubseteq_P d_2 \text{ and } d_2 \sqsubseteq_P d_3 \Rightarrow d_1 \sqsubseteq_P d_3 && (transitivity)\\ &d_1 \sqsubseteq_P d_2 \text{ and } d_2 \sqsubseteq_P d_1 \Rightarrow d_1 = d_2 && (anti-symmetry) \end{aligned}$$

We usually write P *to denote both the poset and the carrier, the context making it clear which is intended.*

Definition 1.2 (Chain) *An* ω*-chain in* P *is a countably infinite family* $\{x_n\}_{n\in\omega}$ *of elements of* P *such that* $x_n \sqsubseteq x_{n+1}$ *for all* $n \in \omega$*.*

Definition 1.3 (Directed Set) *A subset* $D' \subseteq D$ *of a partially ordered set* D *is* directed *iff every finite subset of* D' *has an upper bound in* D'*.*

Definition 1.4 (Least Element) *Given a partially ordered set* $(P, \sqsubseteq_P)$*, if there exists an element* $d \in P$ *such that for all* $c \in P$*,* $d \sqsubseteq_P c$*, then* d *is the least element in* P *and is denoted by the symbol* $\bot$*, pronounced "bottom".*

Definition 1.5 (Bounds) *We write the least upper bound (join) of two elements* x *and* y *as* $x \sqcup y$*.*

We write the greatest lower bound (meet) of two elements x *and* y *as* $x \sqcap y$*.*

Definition 1.6 (Lattices) *A join semi-lattice is a poset in which every pair of elements has a join. A meet semi-lattice is defined dually.*

A lattice is a poset in which every pair of elements has both a meet and a join.

A complete lattice has joins and meets for all subsets of the poset.

Definition 1.7 (Monotone Functions) *Let* A, B *be posets. A map* $f : A \to B$ *is monotone if for elements* $a, a' \in A$*, when* $a \sqsubseteq a'$ *then* $f(a) \sqsubseteq f(a')$*.*

A pointwise ordering on monotone maps is a partial order defined by
$f \sqsubseteq g \iff \forall a \in A, f(a) \sqsubseteq g(a)$*.*

A poset of all monotone maps under the pointwise ordering is written $[A \to_m B]$*.*

Definition 1.8 (Continuity) *A map $f : D \to E$ defined on domains D, E is continuous if for all ω-chains $\{x_n\}, f(\bigsqcup\{x_n\}) = \bigsqcup\{f(x_n)\}$.*

The collection of all continuous maps from D to E under the pointwise ordering forms a domain, written $[\mathrm{D} \to \mathrm{E}]$.

Definition 1.9 (Products and Sums) *Let A, and B be posets. A cartesian product of A and B, written $A \times B$, is ordered by $(a, b) \sqsubseteq (a', b') \iff a \sqsubseteq a'$ and $b \sqsubseteq b'$*

The separated sum is written $A + B$ and has as carrier set $\{\bot\} \cup \{in_1(a) \mid a \in A\} \cup \{in_2(b) \mid b \in B\}$ and is ordered by $\bot \sqsubseteq x, \forall x \in A + B$ and $in_i(a) \sqsubseteq in_j(b) \iff i = j$ and $a \sqsubseteq b$.

The left and right projections on a product are denoted by π_1 and π_2 respectively, thus $\pi_1(a, b) = a$.

Definition 1.10 (CPO) *A partially ordered set P is a* complete partial order *(cpo) iff every ω-chain in P has a least upper bound in P.*

Definition 1.11 (Lifting) *Given a poset A, $A_\bot$ has as carrier set $\{\bot\} \cup \{lift(a) \mid a \in A\}$ ordered by $\bot \sqsubseteq x$ for all x and $lift(a) \sqsubseteq lift(a') \iff a \sqsubseteq a'$.*

Definition 1.12 (Domain) *A domain is a bounded-complete ω-algebraic cpo with least element. See [Sch86] for a more formal introduction. We will not be considering these complex domains as those found in this thesis are more often finite lattices.*

Thus, any finite lattice is a domain. If D, E are domains, then so are $D \times E, D + E$ and $D_\bot$.

Definition 1.13 (Flat Domains) *A domain D is said to be flat if $d \sqsubseteq d' \iff d = \bot$ or $d = d'$. A flat domain that is often used in this thesis is the domain $\mathbf{2}$, with carrier set $\{\mathbf{0}, \mathbf{1}\}$ and $\mathbf{0} \sqsubseteq \mathbf{1}$.*

1.5 Overview of the Thesis

The following is a brief introduction to each chapter:

- Chapter 2 introduces the concepts behind the abstract interpretation framework, typical of a forward analysis that propagates information from the leaves of the expression tree to the root. A denotational description of the simply typed lambda calculus is given, and it is shown that by providing alternate domains, and abstract functions modeling the concrete ones, properties can be extracted from a program. Denotational semantics and domain theory are used as the main tools in the development of the framework, and an application of abstract interpretation to strictness analysis is shown. The notions of properties, abstraction and concretisation maps, best interpretations and correctness are also introduced.

- Chapter 3 introduces some extensions to the abstract interpretation framework. Issues such as combining properties, non-standard types systems, the use of logical relations and the analysis of non-flat domains are introduced. The use of various other mathematical disciplines is emphasized by exploring instantiations of the abstract interpretation framework using logic, logical relations and partial equivalence relations.

- Chapter 4 describes the projection analysis framework. This technique is an example of a backwards analysis in which properties about an expression are propagated through the expression to sub-expressions. Projections are shown to be able to very easily capture a property that we are unable to find using abstract interpretation, namely head strictness. The framework in which we construct the analysis technique uses projection functions found in domain theory, and denotational semantics.

- Chapter 5 describes various implementation issues related to the finding of fixed points in lattices. The pending analysis and frontier analysis techniques are introduced, as well as a method of finding approximate fixed point information.

- The final chapter reviews the material covered in the thesis.

1.6 Conclusion

In this chapter we have addressed some of the issues relating to why program analysis techniques are needed in lazy functional languages. It was argued that abstraction was

the fundamental cause of the efficiency problems, though it also contributes towards the finding of a solution.

Backus [Bac78] and Hughes [Hug89b] argue convincingly on 'why functional programming matters'. In addition Hughes addresses the many benefits gained by using a *lazy* functional language.

Unfortunately we pay a severe price for this laziness, and this thesis examines aspects of static analysis techniques which are geared towards optimizing implementations of these types of languages.

Chapter 2

Abstract Interpretation

In this chapter we examine the abstract interpretation framework. To introduce the concepts of the framework, an informal analogy is drawn between abstract interpretation and the familiar 'rule of signs' calculation which is used to verify the sign of an answer in a mathematical calculation. The fundamental goals of abstract interpretation are then put forward, and we proceed to give a more formal view of the framework, using denotational semantics and domain theory as the basic underlying disciplines. Abstraction and concretisation maps are introduced as ways of relating various interpretations. Scott-closed sets, best interpretations and a notion of correctness are also introduced.

Finally, the history and development of the theory is given.

2.1 Informal Abstract Interpretation

In this section we give an informal introduction to the key ideas behind the abstract interpretation framework. These notions are relatively simple to express and we will do this by drawing an analogy with the familiar 'rule of signs' calculation which is used to verify the sign of an arithmetic result. We conclude this section by making some important observations which should be kept in mind as the more formal development of the theory takes place in the following sections.

Consider the following language of (somewhat restricted) arithmetic expressions,

$$\begin{aligned} Exp ::= & c_n \\ & \mid Exp_1 \textbf{ times } Exp_2 \\ & \mid Exp_1 \textbf{ plus } Exp_2 \end{aligned}$$

which has the following denotational semantics:

$$\begin{aligned} & E^{st} : Exp \to \mathbf{D}^{st} \\ & E^{st}[\![c_n]\!] = n \\ & E^{st}[\![e_1 \textbf{ times } e_2]\!] = E^{st}[\![e_1]\!] \times E^{st}[\![e_2]\!] \\ & E^{st}[\![e_1 \textbf{ plus } e_2]\!] \ = E^{st}[\![e_1]\!] + E^{st}[\![e_2]\!] \end{aligned}$$

We use the superscript *st* to denote that this is the standard semantics. Such denotational descriptions are typically like those found in standard texts on denotational semantics [Sto77, Sch86]. The domain over which the functions are expressed is the flat domain of integers $\mathbf{D}^{st}$, where the bottom element of the domain represents an undefined integer. We also choose our functions **times** and **plus** to be strict in this value, meaning that if either of their arguments are $\perp$, then so is the result. Using this language we can create expressions such as c_{233} **times** c_{-10} and c_{234} **plus** c_{-546} whose standard denotational meaning reflects our standard notions of multiplication and addition.

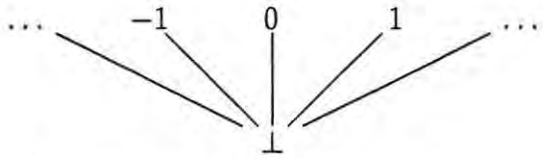
Now let us suppose that we need the sign of an evaluated expression to perform some optimization. One approach to calculating this is to evaluate the entire expression and check the sign of the result. For the example expressions above, this reduces to c_{233} **times** $c_{-10} = 233 \times -10 = -2330$. The answer is clearly negative. A similar calculation can be performed to find out whether c_{234} **plus** c_{-546} is positive, negative or zero.

However, there is a simpler method which yields these results much more quickly. If the set of integers is replaced with an alternative set, say $\{\perp, positive, negative, zero\}$, and the addition function is modified to work over this new set using the familiar 'rule of signs'[1], a less computationally expensive solution can be found. We will parenthesize the multiplication and addition symbol to remind us that it is the 'rule of signs' operation, and not the usual operator over the integers.

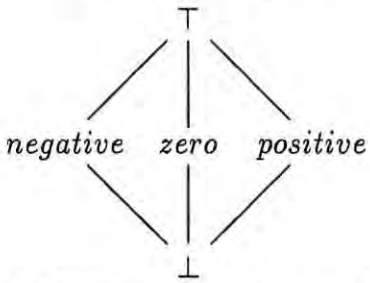
[1]This rule says that a negative number multiplied by a negative number, yields a positive number etc.

As an example, c_{233} **times** c_{-10} reduces to *positive* $(\times)$ *negative* which on application of the rule of signs, reduces to *negative*. Although this method looks very promising, a problem has been introduced by abstracting away the values of the numbers. Specifically, what is the sign of c_{234} **plus** c_{-546}? This reduces to *positive* **plus** *negative* for which we have no answer. To cater for this occurrence, we extend the alternative domain with the value $\top$ (pronounced 'top') which represents the 'unknown' quantity.

To formulate the above analysis, we have done two things. Firstly, the base type over which the language operated has been changed from the flat domain of integers,



to a lattice of elements which denote properties of interest:



We will call this new domain $\mathbf{D}^{ab}$, the superscript emphasizing that it is an *abstract* domain, as opposed to the *concrete* (standard) domain $\mathbf{D}^{st}$.

The semantics for the changed language of expressions is then:

$$
\begin{aligned}
&E^{ab} : Exp \to \mathbf{D}^{ab} \\
&E^{ab}[\![c_n]\!] = sign(E^{st}[\![c_n]\!]) \\
&E^{ab}[\![e_1 \textbf{ times } e_2]\!] = E^{ab}[\![e_1]\!]\,(\times)\,E^{ab}[\![e_2]\!] \\
&E^{ab}[\![e_1 \textbf{ plus } e_2]\!] \;= E^{ab}[\![e_1]\!]\,(+)\,E^{ab}[\![e_2]\!]
\end{aligned}
$$

where the rules for $(+)$ and $(\times)$ are those for the 'rule of signs', and the function *sign* is defined as follows:

$$
sign(x) = \begin{cases} positive & \text{if } x > 0 \\ negative & \text{if } x < 0 \\ zero & \text{if } x = 0 \\ \bot & \text{if } x = \bot \end{cases}
$$

Finding the sign for c_{234} **times** c_{546} now reduces to calculating *positive* (+) *positive*, which by the 'rule of signs' reduces to *positive*. A calculation of the sign for the expression c_{234} **plus** c_{-546} reduces to *positive* (+) *negative* which is $\top$.

One reason for using abstract values instead of the concrete ones is for computability, as we want to ensure that the analysis can be performed in a finite time. By insisting that the abstract domain is finite, we have ensured the effective computation of fixed points over this domain. Something which has not been illustrated in the above example is that we want the results of the analysis to describe all possible program executions [JN90]. Having abstract values denote sets of possible inputs allows us to do this.

As pointed out in [JN90], no 'interesting' analysis over a sufficiently powerful language that always terminates can be exact (due to the undecidability of the halting problem). There are three suggested alternatives to this:

- Restrict the language to allow only finite behaviour and decidable properties.
- Have the system interactively ask for help.
- Find approximate but safe information.

The first point would obviously impair the programmer, and so we discard this idea. We do the same for the second suggestion as we want an automatic system. We thus settle for the third alternative. By 'approximate but safe' we mean that the analysis is allowed to be too conservative in the answer produced, but it must be ensured that the answer is always correct. For the language *Exp*, this means that it is safe to use $\top$ as an answer to c_{234} **times** c_{546}[2] , but not $\bot$, *negative* or *zero*.

Some important observations can be made about the above example, and the reader should keep these in mind when we explore the framework more formally.

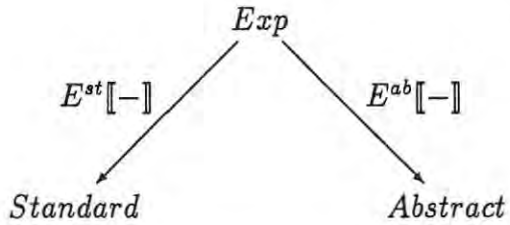
- The *property* of interest in the above example was the sign of an evaluated expression.

[2]Note that *positive* is also safe, and is a 'better' description. This leads us to a notion of a *best* abstraction, which we examine in section 2.4.6

- The expression could be fully evaluated with the standard semantics to find the sign. Alternatively, the expression language could be given a different semantics, and the expression could *then* be evaluated, giving either the identical answer or a 'do not know' (ie. the answer is expressed as *top*).
- The analysis is safe.
- Giving the language an alternate semantics meant finding a new domain and a modification of the semantics of the language to work over this new domain.
- The elements of the new domain somehow capture subsets of the original domain. *Positive* represents all the numbers greater than zero, *negative* all those less than zero and *zero* the number zero in the domain of integers of the original semantics. $\top$ could be interpreted to represent all of them.
- In abstracting away the *value* of a number, some information has been lost which prevents the analysis from determining the sign of certain expressions.

As illustrated in figure 2.1 there are now two interpretations for the language of arithmetic expressions. The first expresses the standard semantics of the language, and the second a semantics which allows for the extraction of certain properties.

Figure 2.1 Two interpretations for the language of expressions.



2.1.1 Abstraction and Concretisation

Having two interpretations for a language, we need to relate them if we are going to show that one is safe with respect to the other. We observed in the previous section that the elements of the abstract domain somehow modeled subsets of the domain of

integers (which we called the *concrete* domain). It is thus natural to define a mapping from the abstract domain to all possible subsets of the concrete domain, with the intention that this map will map abstract elements to the subsets that they represent.

We can represent all possible subsets with a powerdomain, and so we define the following *concretisation* map:

$$Conc : \mathbf{D}^{ab} \rightarrow \mathcal{P}(\mathbf{D}^{st})$$

Instead of defining this map directly, we will define it in terms of other maps. A map that will be useful in its definition is the abstraction map, which maps values of our concrete domain $\mathbf{D}^{st}$ to the abstract domain $\mathbf{D}^{ab}$:

$$abs : \mathbf{D}^{st} \rightarrow \mathbf{D}^{ab}$$

An example of an abstraction map is the *sign* function defined in the abstract semantics for the arithmetic expression language. It takes as arguments elements of the type $\mathbf{D}^{st}$ and returns elements in the abstract domain.

Trying to build up to a natural dual of the $Conc$ map, we now define an abstraction of sets of elements:

$$Abs : \mathcal{P}(\mathbf{D}^{st}) \rightarrow \mathbf{D}^{ab}$$

If the set of elements was {-5,-4,-1} then we could apply our *abs* function to each element and obtain the set $\{negative, negative, negative\}$. The obvious element in the abstract domain which represents this is the element *negative*. If however the set of elements was {-1, 1} and we apply the *abs* function getting $\{negative, positive\}$ we will have to use the element $\top$ to represent the fact that we do not know what the sign will be. The abstraction map *abs* then maps elements of the standard domain to elements in the abstract domain, while *Abs* maps sets of elements of the standard domain to elements in the abstract domain.

Keeping in mind that the elements in our abstract domain are ordered, we can write the *Abs* function as follows:

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

Using this definition the abstractions of the sets {-5,-4,-1} and {-1, 1} yield *negative* and $\top$ respectively.

The concretisation map can now be defined. Recall that we wanted $Conc : \mathbf{D}^{ab} \rightarrow \mathcal{P}(\mathbf{D}^{st})$ to map elements in the abstract domain to the sets of elements that they represent. Thus,

$$Conc(a) = \bigcup\{S \mid Abs(S) \sqsubseteq a\} \qquad \text{where } a \in \mathbf{D}^{ab}$$

which tells us that the concretisation of a value in the abstract domain is the union of the set of elements which when abstracted, are as defined as that value. As an example, we will concretise all of the elements in our abstract domain,

$$\begin{array}{lcll} Conc(\bot) & = & \bigcup\{S \mid Abs(S) \sqsubseteq \bot\} & = \{\bot\} \\ Conc(negative) & = & \bigcup\{S \mid Abs(S) \sqsubseteq negative\} & = \{n \mid n < 0\} \cup \{\bot\} \\ Conc(positive) & = & \bigcup\{S \mid Abs(S) \sqsubseteq positive\} & = \{n \mid n > 0\} \cup \{\bot\} \\ Conc(zero) & = & \bigcup\{S \mid Abs(S) \sqsubseteq zero\} & = \{0\} \cup \{\bot\} \\ Conc(\top) & = & \bigcup\{S \mid Abs(S) \sqsubseteq \top\} & = \mathbf{D}^{st} \end{array}$$

and obtain just what we expected. We have now found a concretisation map $Conc$ from the abstract domain to subsets of the concrete domain. We have also found an abstraction map, Abs, from sets of concrete values to elements of the abstract domain.

2.1.2 Correctness

We can now state what it means for our abstract semantics to *correctly* model the concrete semantics. In the abstract semantics, the $(+)$ function takes arguments of type $\mathbf{D}^{ab} \times \mathbf{D}^{ab}$ to arguments of type $\mathbf{D}^{ab}$. That is:

$$\mathbf{D}^{ab} \times \mathbf{D}^{ab} \xrightarrow{(+)} \mathbf{D}^{ab}$$

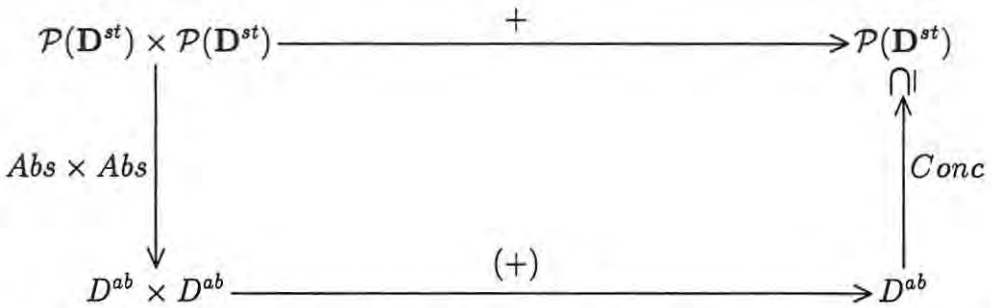
In addition to this, there is also the standard addition function:

$$\mathbf{D}^{st} \times \mathbf{D}^{st} \xrightarrow{+} \mathbf{D}^{st}$$

In the previous section the maps Abs and $Conc$ were defined, but these operated over the powerdomain of the sets. To account for this, we raise the addition function pointwise to obtain:

$$\mathcal{P}(\mathbf{D}^{st}) \times \mathcal{P}(\mathbf{D}^{st}) \xrightarrow{+} \mathcal{P}(\mathbf{D}^{st})$$

Putting all of this together, utilizing the *Abs* and *Conc* maps in the previous section, we can get:



An intuitive reading of the above is as follows: If we calculated c_{233} **plus** c_{343} (which when raised to the powerdomain is of the form $\mathcal{P}(\mathbf{D}^{st}) \times \mathcal{P}(\mathbf{D}^{st})$) we would get an answer lying in $\mathcal{P}(\mathbf{D}^{st})$, namely $\{576\}$.

The *Abs* function allows us to abstract the c_{233} and c_{343} to *positive* and *positive* $(D^{ab} \times D^{ab})$, which on application of the 'rule of signs' version of the addition function leads to an element in the abstract domain, *positive* (D^{ab}). We can now concretise this element, and we define our analysis to be *correct* if this concretisation results in a superset of the values which would have resulted if we had done the calculation in the concrete domain $\mathcal{P}(\mathbf{D}^{st})$. Note that it has to be a superset as we have lost the value of the number in the abstraction, and so we cannot concretise back to the correct number.

2.2 The Goal of Abstract Interpretation

Given a program in some language which has a particular standard semantics, the goal of abstract interpretation is to give this language an alternative semantics (an *abstract* semantics) and to execute the program within this semantics. By doing so, properties about the program can be found. In addition to this, the process must be automatic.

Assigning an abstract interpretation to the language entails the choice of the correct abstract domain which somehow captures the properties of interest. In addition to this, an alternative semantics has to be given to some of the statements in the language.

A notion of *correctness* has to be defined, which informally means that if the abstract interpretation yields a property, then when the program is executed it will also yield

that property under all possible inputs. Abstract interpretation must never falsely infer a property, that is, it must be safe. As the previous sections have shown, an abstract semantics is proved correct by relating it to the standard semantics of the language.

When analyzing a language, the abstract interpretation must terminate. To ensure this, all infinite domains (such as the natural numbers) are replaced with finite ones[3] allowing for the solution of fixed point equations.

2.3 The language $\Lambda_{\mathcal{T}}$

This section introduces a language that is to be used throughout this thesis, the monomorphically typed lambda calculus with constants, which we denote by $\Lambda_{\mathcal{T}}$.

It is assumed that there exists a finite set of base types

$$\imath, \jmath \in \mathcal{T}_0$$

which includes *bool* and *int* representing the booleans and integers. The type system $\mathcal{T}$ is then:

$$\sigma, \tau \in \mathcal{T} ::= \imath \mid \sigma \times \tau \mid \sigma \to \tau$$

For each type we assume the sets Var_σ and Con_σ from which the variables and constants are drawn. The sets of all variables and constants are then:

$$x, y, \ldots \in Var = \bigcup_{\sigma \in \mathcal{T}} Var_\sigma$$

$$c \in Con = \bigcup_{\sigma \in \mathcal{T}} Con_\sigma$$

It is further assumed that when σ and τ are distinct types, then Var_σ and Var_τ are disjoint, as are Con_σ and Con_τ. Thus every variable and constant can be uniquely decorated with its type. This may sometimes be omitted to simplify the notation.

The syntax of terms in $\Lambda_{\mathcal{T}}$ is,

$$e \in \Lambda_{\mathcal{T}} ::= x \mid c \mid \lambda x \,.\, e \mid e_1 e_2 \mid (e_1, e_2) \mid \mathbf{fst}(e) \mid \mathbf{snd}(e)$$

[3]This is not absolutely necessary. The Cousots have developed a method which does not use this restriction.[CC92]

with x denoting the variables, c the constants, $\lambda x\,.\,e$ lambda abstractions, $e_1 e_2$ application, (e_1, e_2) tuppling and $\mathbf{fst}(e), \mathbf{snd}(e)$ decomposition of tuples.

All terms are well-formed in accordance with the type-checking schemata in figure 2.2.

Figure 2.2 Typing schemata for $\Lambda_{\mathcal{T}}$

(1) $x^{\sigma} : \sigma$ if $x \in Var_{\sigma}$

(2) $c^{\sigma} : \sigma$ if $c \in Con_{\sigma}$

(3) $$\frac{e_1 : \sigma \to \tau \quad e_2 : \sigma}{(e_1 e_2) : \tau}$$

(4) $$\frac{e : \tau}{(\lambda x^{\sigma}\,.\,e) : \sigma \to \tau}$$

(5) $$\frac{e_1 : \sigma \quad e_2 : \tau}{(e_1, e_2) : \sigma \times \tau}$$

(6) $$\frac{e : \sigma \times \tau}{\mathbf{fst}(e) : \sigma}$$

(7) $$\frac{e : \sigma \times \tau}{\mathbf{snd}(e) : \tau}$$

The constants of the language include the following:

- **plus, minus** : $int \to int \to int$
- **true, false** : $bool$
- **iszero** : $int \to bool$
- $\mathbf{if}_{\sigma} : bool \to \sigma \to \sigma \to \sigma$, for each type $\sigma \in \mathcal{T}$
- $\mathbf{Y}_{\sigma} : (\sigma \to \sigma) \to \sigma$, for each type $\sigma \in \mathcal{T}$

2.4 Formal Abstract Interpretation

This section explores the abstract interpretation technique proper. We begin by introducing the notion of an *interpretation*, following this by giving the standard semantics and interpretation of the languagé introduced in the previous section. Scott-closed sets are then investigated, and an abstract interpretation to find strictness properties is given to $\Lambda_{\mathcal{T}}$. The *Abs* and *Conc* maps are reviewed, and a more formal notion of correctness is given.

2.4.1 Interpretations

As hinted at previously, when forming an abstract interpretation, only some of the constructs of the language will be given a different interpretation. To formalize the notion of an *abstract* interpretation, we will now look at the definition of an interpretation.

An *interpretation* I, is a tuple

$$(\{D_\iota^I\}_{\iota \in \mathcal{T}_0}, \{K_\sigma^I\}_{\sigma \in \mathcal{T}})$$

where:

- D_ι^I are domains, giving interpretations for the base types.
- K_σ^I are maps from $Con_\sigma \rightarrow D_\sigma^I$ which are the interpretations of the constants of type σ.

The interpretations for the base types are extended to interpretations for the tuple and function types by,

$$\begin{aligned} D^I_{\sigma \times \tau} &= D^I_\sigma \times D^I_\tau \\ D^I_{\sigma \rightarrow \tau} &= [D^I_\sigma \rightarrow D^I_\tau] \end{aligned}$$

where $[D^I_\sigma \rightarrow D^I_\tau]$ represents the set of all continuous functions between D^I_σ and D^I_τ.

The set of all partial maps ρ from Var to $\bigcup_{\sigma \in \mathcal{T}} D^I_\sigma$ is written Env^I.

I then induces a valuation function $[\![-]\!]^I : \Lambda_\mathcal{T} \rightarrow Env^I \rightarrow \bigcup_{\sigma \in \mathcal{T}} D^I_\sigma$. The complete function is given in figure 2.3.

A standard interpretation $\mathbf{S}$ can now be given to $\Lambda_\mathcal{T}$. *bool* and *int* are interpreted as flat domains of booleans and integers. figure 2.4 gives the standard interpretation for the language. An intuitive reading is as follows:

- The meaning of an integer **n** is just the value of the integer in the flat domain of *int*. This domain is written $D^{\mathbf{S}}_{int}$.
- The meaning of **true** and **false** are just their denoted values in the flat domain of *bool*. This domain is written $D^{\mathbf{S}}_{bool}$ and its carrier set is written $\{tt, ff, \perp\}$.

- If **iszero**'s argument is undefined, then so is the result. Otherwise, the value returned is either *tt* or *ff* depending on the value of the argument. The same principle holds for **plus** and **minus**.

- The **if** is interpreted as a conditional, and there exists a separate **if** for each type (because our language is monomorphic).

- Each **Y** is interpreted as a least fixed point operator, and again there is a separate **Y** for each type.

Figure 2.3 The Valuation function induced by I

$$
\begin{array}{ll}
[\![-]\!]^I : \Lambda_T \to Env^I \to \bigcup_{\sigma \in T} D^I_\sigma & \\
[\![c]\!]^I\rho & = K^I_\sigma[\![c]\!] \\
[\![x]\!]^I\rho & = \rho(x) \\
[\![\lambda x\,.\,e]\!]^I\rho & = \lambda d \in D^I_\sigma\,.\,[\![e]\!]^I\rho[x \to d] \text{ if } x \in Var_\sigma \\
[\![e_1 e_2]\!]^I\rho & = ([\![e_1]\!]^I\rho)([\![e_2]\!]^I\rho) \\
[\![(e_1, e_2)]\!]^I\rho & = ([\![e_1]\!]^I\rho, [\![e_2]\!]^I\rho) \\
[\![\mathbf{fst}(e)]\!]^I\rho & = \pi_1([\![e]\!]^I\rho) \\
[\![\mathbf{snd}(e)]\!]^I\rho & = \pi_2([\![e]\!]^I\rho)
\end{array}
$$

It is now possible to give an abstract interpretation to the language. By the definition of an interpretation above, a new set of base domains has to be supplied, as well as a new set of constants over these domains. We proceed by defining a useful analysis that will be used throughout the thesis, *strictness analysis*, and we will denote this abstract interpretation by the symbol **B** (expressing the origin of the framework, due to Burn, Hankin and Abramsky [BHA86]).

Informally a function is strict if, when given a non-terminating argument, the function itself does not terminate. Formally:

Definition 2.1 (Strictness) *A function $f : D \to E$ is said to be* strict *if $f(\bot_D) = \bot_E$.*

Figure 2.4 The Standard Interpretation of $\Lambda_{\mathcal{T}}$

$D^{\mathbf{S}}_{int}$ and $D^{\mathbf{S}}_{bool}$ are interpreted as flat domains.

$$
\begin{array}{lcl}
\mathbf{n}^{\mathbf{S}} & = & n \\
\mathbf{true}^{\mathbf{S}} & = & tt \\
\mathbf{false}^{\mathbf{S}} & = & ff \\
\mathbf{iszero}^{\mathbf{S}} x & = & \begin{cases} \bot & \text{if } x = \bot \\ tt & \text{if } x = 0 \\ ff & \text{otherwise} \end{cases} \\
\mathbf{plus}^{\mathbf{S}}\ x\ y & = & \begin{cases} \bot & \text{if } x = \bot \text{ or } y = \bot \\ x + y & \text{otherwise} \end{cases} \\
\mathbf{minus}^{\mathbf{S}}\ x\ y & = & \begin{cases} \bot & \text{if } x = \bot \text{ or } y = \bot \\ x - y & \text{otherwise} \end{cases} \\
\mathbf{if}^{\mathbf{S}}_{\sigma}\ b\ x\ y & = & \begin{cases} \bot_{\sigma} & \text{if } b = \bot \\ x & \text{if } b = tt \\ y & \text{if } b = ff \end{cases} \\
\mathbf{Y}^{\mathbf{S}}_{\sigma} f & = & \bigsqcup_{i \in \omega} f^i \bot^{\mathbf{S}}_{\sigma}
\end{array}
$$

By giving a very simple alternate interpretation, we can define a strictness analysis over the terms of our language which will determine whether a function is strict or not. To calculate the information of interest, we need to answer the question "Does this function, when applied to a non-terminating argument, terminate?". We choose a two point domain to answer this question, named **2**, with points **0** and **1** with $\mathbf{0} \sqsubseteq \mathbf{1}$. We intend the abstraction of x to be **1** if x *may* terminate, and the abstraction of x to be **0** if x *definitely* fails to terminate.

We would expect the evaluation of any integer or boolean to terminate, and so would naturally set $\mathbf{n}^{\mathbf{B}} = \mathbf{true}^{\mathbf{B}} = \mathbf{false}^{\mathbf{B}} = \mathbf{1}$.

For **iszero** we would expect this function to terminate if its argument terminates, and so we set $\mathbf{iszero}^{\mathbf{B}} x = x$. For the arithmetic function, we expect something similar. If either of the two arguments fail to terminate, then so does the expression. This is conveniently expressed using the meet operator as our domain is ordered. We thus set $\mathbf{plus}^{\mathbf{B}}\ x\ y = \mathbf{minus}^{\mathbf{B}}\ x\ y = x \sqcap y$.

We would expect that $\mathbf{if}^{\mathbf{B}}_{\sigma}\ b\ x\ y$ would not terminate if b did not terminate or if both x and y did not terminate. We thus set this to $x \sqcup y$ if b terminates, $\mathbf{0}$ otherwise.

The full interpretation for strictness analysis is shown in figure 2.5.

Figure 2.5 The Abstract Interpretation for Strictness analysis

$D^{\mathbf{B}}_{int}$ and $D^{\mathbf{B}}_{bool}$ are interpreted as the two point domain, $\mathbf{2}$

$\mathbf{n}^{\mathbf{B}}$	$=$	$\mathbf{1}$
$\mathbf{true}^{\mathbf{B}}$	$=$	$\mathbf{1}$
$\mathbf{false}^{\mathbf{B}}$	$=$	$\mathbf{1}$
$\mathbf{iszero}^{\mathbf{B}}\ x$	$=$	x
$\mathbf{plus}^{\mathbf{B}}\ x\ y$	$=$	$x \sqcap y$
$\mathbf{minus}^{\mathbf{B}}\ x\ y$	$=$	$x \sqcap y$
$\mathbf{if}^{\mathbf{B}}_{\sigma}\ b\ x\ y$	$=$	$b \sqcap (x \sqcup y)$
$\mathbf{Y}^{\mathbf{B}}_{\sigma}\ f$	$=$	$\bigsqcup_{i \in \omega} f^i\ \bot^{\mathbf{B}}_{\sigma}$

2.4.2 An Example

To demonstrate the application of a strictness interpretation, we will consider determining the strictness of the function,

$$f\ x\ y\ z = \mathbf{if}\ (\mathbf{iszero}\ y)\ (f\ 2\ 3\ x)\ x$$

which when expressed in our language, involves finding the strictness of:

$$\mathbf{Y}(\lambda f.\lambda x.\lambda y.\lambda z.\mathbf{if}\ (\mathbf{iszero}\ y)\ (f\ 2\ 3\ x)\ x)$$

As indicated in figure 2.5, the constant numbers take on the value $\mathbf{1}$ in the abstract interpretation, and so the above expression reduces to

$$\mathbf{Y}(\lambda f'.\lambda x'.\lambda y'.\lambda z'.\mathbf{if}\ (y')\ (f'\ \mathbf{1}\ \mathbf{1}\ x')\ x')$$

where we have primed the variables to indicate that we are now in the abstract domain. Using the rule for the conditional, we get:

$$\mathbf{Y}(\lambda f'.\lambda x'.\lambda y'.\lambda z'.y' \sqcap ((f'\ \mathbf{1}\ \mathbf{1}\ x') \sqcup x'))$$

To find the fixpoint of the above expression, we turn to the the ascending Kleene chain, $\bigsqcup_{i\in\omega} f^i \perp_\sigma^{\mathbf{B}}$. Calculating the fixed point then involves iterating the formula until it converges (at which point the least upper bound has been found). See the introduction of chapter 5 for more information on finding fixed points. We then have:

$$\begin{aligned} f'^1\, x'\, y'\, z' &= y' \sqcap ((f'^0\, \mathbf{1}\, \mathbf{1}\, x') \sqcup x') \\ &= y' \sqcap (\mathbf{0} \sqcup x') \\ &= y' \sqcap x' \end{aligned}$$

The next element in the chain will be:

$$\begin{aligned} f'^2\, x'\, y'\, z' &= y' \sqcap ((f'^1\, \mathbf{1}\, \mathbf{1}\, x') \sqcup x') \\ &= y' \sqcap (\mathbf{1} \sqcup x') \\ &= y' \sqcap \mathbf{1} \\ &= y' \end{aligned}$$

Continuing, we get:

$$\begin{aligned} f'^3\, x'\, y'\, z' &= y' \sqcap ((f'^2\, \mathbf{1}\, \mathbf{1}\, x') \sqcup x') \\ &= y' \sqcap (\mathbf{1} \sqcup x') \\ &= y' \end{aligned}$$

and so we have indeed found the fixed point, and we denote this function by f''.

To test for strictness, need only set the variable of interest to $\mathbf{0}$ and the others to $\mathbf{1}$, and calculate the result. This provides an effective test for $f\ \perp = \perp$. Thus to test for strictness of x, we calculate $f''\ \mathbf{0}\ \mathbf{1}\ \mathbf{1}$ which is $\mathbf{1}$. Thus the function is not strict in x. This also holds for z. For y however, we have that $f''\ \mathbf{1}\ \mathbf{0}\ \mathbf{1} = \mathbf{0}$. Thus the function f is strict in y.

The strictness analysis presented here is an example of a *safe* analysis. Any overestimation of information means that we fail to infer that a function is strict when it is. Termination analysis [Myc81, Abr90] is a *live* analysis. In termination analysis, $\mathbf{1}$ represents definite termination, and $\mathbf{0}$ possible non-termination. Any underestimation then means that we fail to infer that a function terminates when it actually does.

2.4.3 Forward Analyses

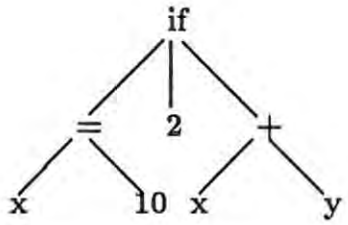
Following Hughes [Hug90], abstract interpretation can be motivated as providing a framework which conveys information in a 'forwards' manner. To illustrate this idea, we turn back to the strictness analysis example.

Let us examine the syntax tree for the expression:

$$f\ x\ y = \textbf{if}\ x = 10\ \textbf{then}\ 2\ \textbf{else}\ x + y$$

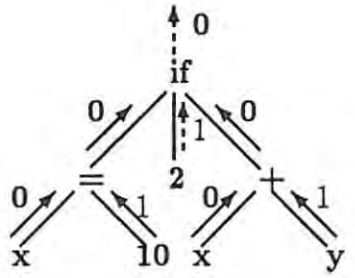
shown in figure 2.6.

Figure 2.6 Syntax tree for an **if** statement.



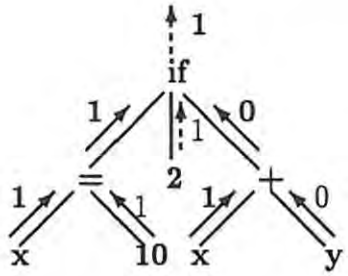
In a forward analysis we propagate information from the leaves of the tree to the root. We want to produce some information about the arguments of a function, and deduce information about the expression itself. Thus in an analysis where we tried to show that the above function is strict in x, we would set x to **0** and the constants and y to **1** and then propagate this information through the tree as in figure 2.7.

Figure 2.7 Forward analysis showing strictness in x



This should be compared to the analysis in figure 2.8 where we show that the function is not strict in y.

Figure 2.8 Forward analysis showing no strictness in y



2.4.4 Scott-Closed Sets

Previously, we observed that the elements of the abstract domain somehow modeled subsets of the concrete domain. Here, we build on this, showing that the elements represent Scott-closed subsets of the concrete domain.

Definition 2.2 (Down-closed) *Given a domain D, a subset S is said to be down-closed if whenever $s \in S$ and there exists an $s' \in D$ such that $s' \sqsubseteq s$, then $s' \in S$.*

The down-closed sets of the domain $\mathbf{D}^{ab}$ defined in section 2.1 are $\{\bot\}$, $\{\bot, positive\}$, $\{\bot, negative\}$,$\{\bot, negative, positive, zero\}$,...

Definition 2.3 (Scott-closed) *A set S is Scott-closed if:*

- *it is down-closed,*
- *when X is a directed subset of S, then $\bigsqcup X \in S$.*

The abstract interpretation framework of [BHA86] uses, as we have thus far, each element of the abstract domain to model Scott-closed sets of the concrete domain. All of the sets on the right hand side of the equations defining *Conc* on page 23 are Scott-closed sets. We thus say that a *property* is a Scott-closed set. In later sections we will

see that properties can be represented by other objects, specifically partial equivalence relations.

Some important attributes of Scott-closed sets are that finite unions and intersections of Scott-closed sets are Scott-closed. This implies that our theory can also represent finite *conjunctions* and *disjunctions* [Bur91] as they can be represented as properties. Note that the complement of a Scott-closed set is not necessarily Scott-closed and so we cannot represent the negation of a property.

The set of all non-empty Scott-closed subsets of a domain D is called the *Hoare power domain* and forms a complete meet semi-lattice when ordered by subset inclusion. We write this power domain as $\mathcal{P}_H(D)$.

The abstract interpretation framework of [BHA86] uses this as the powerdomain operator. A consequence of this is that if the property of interest is not representable by Scott-closed sets, then this particular implementation of abstract interpretation will be unable to find it.

2.4.5 Abs, Conc and Correctness

Although this section does not prove the abstract interpretation framework, we rephrase some of the material presented in section 2.1.2 using terms in $\Lambda_\mathcal{T}$ and the Hoare power domain. The ideas about providing abstract interpretations is that we can use them to make assertions about computations in the standard interpretation. Thus for a function, say $f : \sigma \to \tau$, with standard interpretation:

$$D_\sigma^{\mathbf{S}} \xrightarrow{[\![f]\!]^{\mathbf{S}}\rho^{\mathbf{S}}} D_\tau^{\mathbf{S}}$$

we wish to deduce certain properties from the abstract interpretation:

$$D_\sigma^{\mathbf{B}} \xrightarrow{[\![f]\!]^{\mathbf{B}}\rho^{\mathbf{B}}} D_\tau^{\mathbf{B}}$$

Our notion of safety says that an abstract interpretation is correct if whenever:

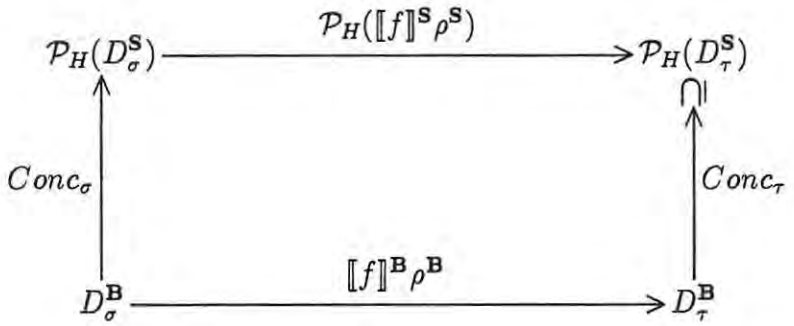
$$([\![f]\!]^{\mathbf{B}}\rho^{\mathbf{B}})\, s = t$$

then for all s' represented by s, that is $s' \in Conc_\sigma(s)$,

$$([\![f]\!]^{\mathbf{S}}\rho^{\mathbf{S}})\, s' \in Conc_\tau(t)$$

holds. The diagram in figure 2.9 says just this.

Figure 2.9 The Safety Diagram for Abstract Interpretation



Another way of looking at this is to say that with every function $f : D^{\mathbf{S}} \to D^{\mathbf{S}}$ we will associate an *abstract* function $Abs\, f : D^{\mathbf{B}} \to D^{\mathbf{B}}$. We want $Abs\, f$ to predict every possible result that f might produce, and when this happens we say that $Abs\, f$ is *safe*. We thus say:

$$\text{for } s \in D^{\mathbf{B}} \text{ and for each } x \in Conc(s),\ \ f\, x \in Conc\,((Abs\, f)s)$$

A result used in the proof of the above, is the following:

Theorem 2.1 *Suppose that* $[\![c_\sigma]\!]^{\mathbf{B}}\rho^{\mathbf{B}} \sqsupseteq abs_\sigma([\![c_\sigma]\!]^{\mathbf{S}}\rho^{\mathbf{S}})$ *for all constants (each constant is safely abstracted) then for all environments* $\rho^{\mathbf{B}}, \rho^{\mathbf{S}}$ *in which all variables are safely abstracted, that is,* $\rho^{\mathbf{B}}(x^\tau) \sqsupseteq abs_\tau(\rho^{\mathbf{S}}(x^\tau))$*, we will have for all terms* $e : \sigma$*:*

$$[\![e]\!]^{\mathbf{B}}\rho^{\mathbf{B}} \sqsupseteq abs_\sigma([\![e]\!]^{\mathbf{S}}\rho^{\mathbf{S}})$$

This theorem tells us that we can prove a strong relationship between the abstract and concrete semantics of the language *by only* proving that the above conditions hold for the constants of the language. This theorem is used to prove the main result of the abstract interpretation framework, which is the correctness theorem for abstract interpretation:

Theorem 2.2 (The Correctness Theorem for Abstract Interpretation) *Given a function $f : \sigma \to \tau$ and environments and interpretations of constants satisfying the conditions of the previous theorem, we have that if:*

$$s' \in D_\sigma^{\mathbf{B}} \text{ and } (\llbracket f \rrbracket^{\mathbf{B}} \rho^{\mathbf{B}})(s') = t'$$

then for all $s \in Conc_\sigma(s')$,

$$(\llbracket f \rrbracket^{\mathbf{S}} \rho^{\mathbf{S}})(s) \in Conc_\tau(t')$$

Proofs of the above theorems can be found in [BHA86, BPR88]. In the framework, finite lattices are used for the abstract domains (to ensure effectiveness of the analysis) and the maps *Abs* and *Conc* are shown to be monotonic. The final proof shows that figure 2.9 is indeed true for the strictness interpretation **B** presented in figure 2.5.

2.4.6 Best Interpretations

In section 2.1.1 abstraction and concretisation maps were introduced as mechanisms to allow us to relate a standard and an abstract interpretation. We will now look at a notion of the *best* interpretation.

If we consider the abstract lattice $\mathbf{D}^{ab}$ representing the signs of an expression, the top element was looked at as a representation of sets of conflicting information, such as $\{-1, 2\}$. Being the top element, it is also capable of representing a set such as $\{-1, -3\}$, and although there is nothing wrong with this representation, it is not the *best* in the sense that we could do better if *negative* was to represent the set instead.

More precisely, a set C is *safely* described by an element of the abstract domain a, if

$$C \subseteq Conc(a)$$

Thus for the example above, both *negative* and $\top$ are safe, but *negative* should be preferred because if we have two elements such that $a \sqsubseteq b$ then $Conc(a)$ will be a subset of the $Conc(b)$, (by monotonicity of $Conc$). The larger the abstract value, the larger the set of values it represents and thus the less precise information is conveyed.

To show that the abstraction map produce a best description (relative to the concretisation maps), we need to have some relation between the two maps, and this is captured by the notion of *adjoinedness* [CC79].

Definition 2.4 (Adjoined) *Abs and Conc are said to be* adjoined *if*

$$\begin{aligned} Conc(Abs(a)) &\sqsupseteq a \\ Abs(Conc(a)) &\sqsubseteq a \end{aligned}$$

The abstraction and concretisation maps then, in a sense, model each other. The above definition of adjoinedness can be used to show the following equivalence [CC79]:

Definition 2.5 (Adjoined) *If Abs and Conc are adjoined then for each* $X \in \mathcal{P}_H(D^{\mathbf{S}})$, $a \in D^{\mathbf{B}}$:

$$X \subseteq Conc(a) \iff Abs(X) \sqsubseteq a$$

This is exactly the requirement that we need to ensure best interpretations. As an example, consider the above definition with a instantiated to *negative* (a point in the abstract domain), and X to the subset $\{-2, \perp\}$. The definition above then says that if $X \subseteq Conc(negative)$, then the abstraction of the set X will be as defined as *negative*. This precludes *top* from being a description of the set and this is what we want if we are looking for a best description. In category theory, we say that *Abs* and *Conc* form a Galois connection.

Abramsky and Hunt [Abr90, Hun91] show that *abs* can be defined inductively by:

$$\begin{aligned} abs_\iota(c) &= \begin{cases} 0 & \text{if } c = \perp \\ 1 & \text{otherwise} \end{cases} \\ abs_{\sigma\to\tau}(f) &= \lambda a. \bigsqcup\{abs_\tau(f\ d) \mid abs_\sigma(d) \sqsubseteq a\} \\ abs_{\sigma\times\tau}(x,y) &= (abs_\sigma(x), abs_\tau(y)) \end{aligned}$$

to produce best interpretations.

2.5 Related Work

The Cousots [CC79] were pioneers in the formalization of analysis techniques for imperative data flow languages, and in essence developed abstract interpretation as we know it. (The basic ideas of substituting abstract values can actually be traced back to Peter Naur in 1963). Mycroft [Myc81] put the Cousots' work in a functional language

framework, though his initial formulation was only capable of analyzing first order functions and flat data structures and operated on a type-less language.

Burn, Hankin and Abramsky [BHA85, BHA86] founded the "BHA-style" of analysis which is essentially an extension of Mycroft's work to operate over a monomorphically typed higher order functional language. The framework presented in this thesis is a typical BHA-style abstract interpretation. Burn has since generalized this work in his thesis [Bur87a].

Abstraction and concretisation maps can be found in the Cousots' work where they were used to relate successive approximate interpretations to a collecting interpretation. Nielson [NN91] investigates these maps extensively, and also discusses adjoinedness and best interpretations in detail. These issues are also addressed in most presentations of the abstract interpretation framework.

Scott-closed sets were used implicitly in the BHA presentation, and also in Burn's thesis. The notions of associating properties with Scott-closed sets became more evident when researchers attempted to explore the limitations of the framework and its relationship with other frameworks [Bur92, Bur91, Bur90a].

The distinction between forward and backward analyses became more apparent once the backwards projection analyses evolved [Hug87]. These notions are perhaps a bit misleading since recently Hughes and Launchbury have shown how to reverse abstract interpretation [HL91, HL92a, HL92b].

A description of an implementation of the BHA framework can be found in Seward's thesis [Sew91]. Formal results indicating how strictness information can be used to change the evaluation order can be found in [Bur87b, Bur92].

2.6 Conclusion

In this chapter we have looked at the basic abstract interpretation framework as found in [BHA86, Bur87a] and all other standard texts on the subject. Emphasis has not been placed on proving the framework correct, (see the above references for details on this), but rather on the important concepts behind the framework.

These include:

Computing with Abstract Values By drawing an analogy with the rule of signs calculation, we have shown that properties can be found for a program by executing this program over an abstract domain of values. This is the essence of abstract interpretation. Having a finite set of abstract values also allows for an *effective* analysis as we can now compute fixed points. We also observed that the abstract values were representative of the concrete standard domain, thus introducing the notion of a property.

Properties In the formal setting Scott-closed sets were shown to model properties, and analyses could only be developed in this particular framework if the properties of interest could be modeled by Scott-closed sets.

Abs **and** *Conc* In striving to prove that properties found using abstract interpretation are indeed correct, abstraction maps mapping values from the concrete domain to abstract values in the abstract domain, and concretisation maps mapping values from the abstract domain to subsets of values in the concrete domain were introduced. It was shown that if these maps form adjoints then best interpretations could be made.

Correctness The notion of correctness was introduced. A value computed by an abstract function was said to be correct if it described every possible output of the concrete function.

Chapter 3

Abstract Interpretation: Extended

The previous chapter introduced abstract interpretation from a purely domain theoretic vantage point, with the Hoare powerdomain playing an important role. In this chapter we look at other formulations such as non-standard type systems, logical relations and partial equivalence relations.

We will introduce non-standard type systems by following Jensen and viewing them as program logics. Logical relations can be seen as a very clean way of specifying the abstract interpretation framework, and our presentation will follow Abramsky and Hunt in the use of Plotkin's binary logical relations theorem. We will also look at a limitation of the framework due to the use of Scott-closed sets as models of the properties and introduce Hunt's partial equivalence relations as an alternate representation.

Wadler's extension to non-flat data types are also introduced, as well as issues concerning the combination of properties using the cartesian and tensor product.

3.1 Other Disciplines

In this section we examine three different analysis disciplines which embody abstract interpretation. One property of functional languages that is routinely derived automatically is its type correctness. We would obviously benefit if these inference algorithms could be put to use to calculate strictness properties as well. Non-standard type systems are a means of calculating strictness properties using specialized types. These

concepts are examined in section 3.1.1.

Logical relations provide a clean mathematical reformulation of the abstract interpretation framework presented in the previous chapter. Its primary contribution is in the elimination of the need for powerdomains in the theory. We look at this in section 3.1.2.

Partial equivalence can be seen as an extension to the logical relations approach. A restriction of the abstract interpretation presented thus far is that the property of interest must be describable by the Scott-closed sets. The partial equivalence relation (per) discipline allows for the capturing of different properties than the Scott-closed set discipline by making the elements of the abstract domain represent pers over the domain rather than Scott-closed sets of the domain. This is examined in section 3.1.3.

3.1.1 Non-Standard Type Systems

By using non-standard type inference rules, Kuo and Mishra [KM87, KM89] have shown that it is possible to extract properties from a program. Jensen [Jen91, Jen92] has extended this work and introduced the use of a program logic. We will introduce some of Jensen's work on abstract interpretation in a logical form. This will be done by first introducing the basic concepts behind the framework. Ideals and filters of lattices are then looked at with the purpose of presenting an axiomatic system for deducing properties. We then introduce a strictness logic and provide an example of its use. It may be beneficial to first read the example section directly after this one to get a feel for the application.

The Basic Intuitions

In chapter 2 correctness theorems were presented which related the non-standard systems to the standard system. Here, we take a slightly different approach and relate a non-standard type system to an abstract interpretation system. Recall that the standard type system for $\Lambda_{\mathcal{T}}$ was presented in figure 2.2.

If we create an analysis framework and prove that it is correct with respect to abstract interpretation, we can rely on existing correctness proofs for abstract interpretation to show that our system is indeed correct. Specifically, we will create a strictness logic for

the lambda calculus, and show that it is complete and sound with respect to a strictness analysis performed via abstract interpretation. This ensures that it is also correct with respect to the standard semantics of the language. Jensen originally created his framework as a way of relating the relative powers of the abstract interpretation and type inference analysis techniques.

Lattices and Properties

To relate the domain-theoretic abstract interpretation to logic, we start by constructing a map between the domains used in the abstract interpretation and a collection of properties. Because we are hoping to show that the logic is complete with respect to abstract interpretation, we also construct a mapping which does the reverse.

In a set-based logic we identify a property by the subset of elements satisfying that property. Likewise, any subset will determine a property. We say that property P entails property Q if the sets of elements that satisfy P is a subset of the set of elements satisfying Q. We will henceforth write P for 'property P' and use the letters P and Q to denote properties.

As an example, a domain that was used frequently in the abstract interpretation framework of the previous chapter, is the domain **2** where we interpret the bottom element as denoting non-termination, and the top denoting termination. (Note that the domains used in abstract interpretation are join-semilattices.) Here we have properties $\{\mathbf{0}\}, \{\mathbf{1}\}$ and $\{\mathbf{0}, \mathbf{1}\}$. Note that the latter element denotes the vacuous property satisfied by all elements. If we are only interested in non-termination, we need only look at properties $\{\mathbf{0}\}$ and $\{\mathbf{0}, \mathbf{1}\}$. A key idea in [Jen91] is to realize that if the set is ordered (just as $\mathbf{0} \sqsubseteq \mathbf{1}$), we can require that our properties embody this ordering. As an example, we can require that the properties must be down closed sets. This will give us just those two points mentioned previously for non-termination. This concept is formalized by insisting that each property in an analysis must be an ideal of a join semi lattice.

Definition 3.1 *An ideal of a join-semilattice A is a non-empty subset $X \subseteq A$ where:*

1. *X is down closed*
2. *X is closed under binary joins. $(x, y \in X \Rightarrow x \sqcup y \in X)$*

If all of the ideals of A are ordered by subset inclusion we will form a meet-semilattice, with the meet of two ideals being represented by the set intersection of the ideals. (Note that the union of two ideals is generally not an ideal.)

If we denote the class of all finite join-semilattices by **JSL**, and that of finite meet-semilattices by **MSL**, we have just created a mapping $Idl : \mathbf{JSL} \rightarrow \mathbf{MSL}$ which maps a join-semilattice to its meet-semilattice of ideals.

Now that we have constructed a collection of properties from a domain, we want to reverse the process and construct a domain from the set of properties. We will do this by constructing a new domain, whose elements are properties using the notion that an element should be fully described by the properties it satisfies. This cannot be done *ad hoc*. If P_1 holds for some element p, and we have that P_1 implies P_2, then P_2 must hold for p as well. To accomplish this, we ensure that only properties closed under implication and conjunction are considered. This is formalized by the concept of a *filter* of a meet-semilattice.

Definition 3.2 *A* filter *of a meet-semilattice B is a non-empty subset $X \subseteq B$ where:*

1. *X is upward closed.*
2. *If $P_1, P_2 \in X$, then $P_1 \wedge P_2 \in X$.*

Our domain will then be the filters of the properties ordered by reverse inclusion, and the least upperbound operator being set intersection. We thus have a map Fil : **MSL** $\rightarrow$ **JSL**

Jensen shows that this resulting **JSL** is isomorphic to the original **JSL**, implying that the abstract domains found in abstract interpretation can equally well be represented by a join-semilattice of elements or as a meet-semilattice of properties as these two structures determine each other up to an isomorphism.

Axiomatisation

We are now in a position to present a formal system for reasoning about properties. As in section 2.4.1 we will consider the base domain **2**. We will however, restrict this presentation by only considering the base types and the function type. See [Jen92]

for an analysis which also caters for pairs of elements. To each type in the abstract interpretation we can associate a join-semilattice by $D : \mathcal{T} \to \mathbf{JSL}$ defined as:

$$\begin{array}{lcl} D(\mathbf{2}) & = & \{\mathbf{0},\mathbf{1}\} \text{ with } \mathbf{0} \sqsubseteq \mathbf{1} \\ D(\sigma \to \tau) & = & D(\sigma) \to_m D(\tau) \end{array}$$

The results of the previous section guarantee that these join-semilattices can be represented by the meet-semilattice of their ideals. We now give an axiomatic system for *defining* these meet-semilattices. Thus we axiomize the description of the meet-semilattice $Idl(A)$ where A is a join-semilattice created above.

For every type σ we need to define a logical theory:

$$\mathcal{L}(\sigma) = (L(\sigma), \wedge, \leq, =)$$

where $L(\sigma)$ is a set of formulae denoting the properties, $\wedge$ is a binary conjunction operator, $\leq$ is our entailment relation and $=$ is logical equivalence between formulae.

We distinguish two properties (ideals), $\mathbf{t}$ denoting the empty meet (ie. it is the top element, the greatest ideal), and $\mathbf{f}$ the least ideal (ie. the least element). The set of formulae are then defined inductively by the following set of rules:

- $\mathbf{t}, \mathbf{f} \in L(\sigma)$
- $\dfrac{\phi, \psi \in L(\sigma)}{\phi \wedge \psi \in L(\sigma)}$
- $\dfrac{\phi \in L(\sigma), \psi \in L(\tau)}{\phi \to \psi \in L(\sigma \to \tau)}$

We now need to define the axioms and rules which impose a meet-semilattice structure

on $L(\sigma)$. Note that ordering is by entailment.

- $\phi \leq \phi$
- $\dfrac{\phi \leq \psi, \psi \leq \chi}{\phi \leq \chi}$
- $\phi \wedge \psi \leq \phi$
- $\dfrac{\phi \leq \psi_1, \phi \leq \psi_2}{\phi \leq \psi_1 \wedge \psi_2}$
- $\phi \wedge \psi \leq \psi$
- $\dfrac{\phi \leq \psi, \psi \leq \phi}{\phi = \psi}$
- $\dfrac{\phi = \psi}{\phi \leq \psi, \psi \leq \phi}$
- $(\phi \to \psi_1 \wedge \phi \to \psi_2) \leq \phi \to \psi_1 \wedge \psi_2$
- $\dfrac{\phi_2 \leq \phi_1, \psi_1 \leq \psi_2}{\phi_1 \to \psi_1 \leq \phi_2 \to \psi_2}$
- $\phi \leq \mathbf{t}_\sigma$
- $\mathbf{t}_{\sigma \to \tau} = \mathbf{t}_\sigma \to \mathbf{t}_\tau$
- $\mathbf{f}_\sigma \leq \phi$
- $\mathbf{t}_\sigma \to \mathbf{f}_\tau = \mathbf{f}_{\sigma \to \tau}$

To establish a connection between the logical system defined above, and the collection of ideals of $D(\sigma)$, we establish a semantic function $[\![\_]\!]_\sigma : \mathcal{L}(\sigma) \to Idl(D(\sigma))$ defined by:

$$\begin{array}{lcl} [\![\mathbf{t}]\!]_\sigma & = & D(\sigma) \\ [\![\mathbf{f}]\!]_\sigma & = & \bot_\sigma \\ [\![\phi_1 \wedge \phi_2]\!]_\sigma & = & [\![\phi_1]\!]_\sigma \cap [\![\phi_2]\!]_\sigma \\ [\![\phi \to \psi]\!]_{\sigma \to \tau} & = & \{f \mid \forall x \in [\![\phi]\!]_\sigma . f\ x \in [\![\psi]\!]_\tau\} \end{array}$$

Thus the logical system $\mathcal{L}(\sigma)$ provided an axiomatic description for the meet-semilattice formed by the ideals of the join-semilattice of the abstract domains. The semantic function $[\![\_]\!]_\sigma : \mathcal{L}(\sigma) \to Idl(D(\sigma))$ then provides a way of relating this logical system to the meet-semilattice.

Readers familiar with logic will recognize that the ideals are a model for the logic.

Jensen [Jen91] proves the completeness and soundness theorems:

Definition 3.3 (Soundness) *For all elements $\phi, \psi \in \mathcal{L}(\sigma)$:*

$$\phi \leq \psi \Rightarrow [\![\phi]\!]_\sigma \subseteq [\![\psi]\!]_\sigma$$

and

Definition 3.4 (Completeness) *For all elements* $\phi, \psi \in \mathcal{L}(\sigma)$*:*

$$[\![\phi]\!]_\sigma \subseteq [\![\psi]\!]_\sigma \Rightarrow \phi \leq \psi$$

Finally, it is also shown that all ideals can be denoted by a formula from the formal system, that is:

Definition 3.5 (Definability) *For all elements* $a : D(\sigma)$ *there exists a formula* $\psi_a \in \mathcal{L}(\sigma)$ *such that:*

$$down\ closure(a) = [\![\psi_a]\!]_\sigma$$

Note that this theorem might generate different formulae denoting the same ideal, but the completeness theorem guarantees us that these formulae are all logically equivalent. If we could group together all of the equivalent formulae (that is, quotient the set of formulae by logical equivalence) we will arrive at a structure that is isomorphic to the ideals of the domain $D(\sigma)$. (This quotient structure is known as the *Lindenbaum algebra* $\mathcal{LA}(\sigma)$ of the logical structure $\mathcal{L}(\sigma)$). We have thus shown that:

$$D(\sigma) \cong Fil(Idl(D(\sigma)) \cong Fil(\mathcal{LA}(\sigma))$$

A Strictness Logic

The work of the previous section constructs a very general framework for relating non-standard type systems to abstract interpretation. Jensen applies these developments to strictness analysis and constructs a strictness logic which is proved to be sound and complete with respect to abstract interpretation. That is, he proves that the denotation of any expression given by a strictness analysis is determined by, and determines, the set of formulae provable of e in the strictness logic.

We will present the strictness logic and give an example of its use:

$$\frac{\Gamma \vdash E : \psi_1, \Gamma \vdash E : \psi_2}{\Gamma \vdash E : \psi_1 \wedge \psi_2}\ \mathbf{Conj} \qquad \frac{\Gamma \leq \Delta, \Delta \vdash E : \phi,\ \phi \leq \psi}{\Gamma \vdash E : \psi}\ \mathbf{Weak}$$

$$\Gamma \vdash E : \mathbf{t}\ \mathbf{Taut} \qquad \Gamma[x \mapsto \phi] \vdash x : \phi\ \mathbf{Var}$$

$$\frac{\Gamma[x \mapsto \phi] \vdash E : \psi}{\Gamma \vdash \lambda x.E : (\phi \to \psi)}\ \mathbf{Abs} \qquad \frac{\Gamma \vdash E_1 : (\phi \to \psi),\quad \Gamma \vdash E_2 : \phi}{\Gamma \vdash E_1 E_2 : \psi}\ \mathbf{App}$$

$$\frac{\Gamma \vdash E : \phi \to \phi}{\Gamma \vdash \mathbf{Y} : \phi}\ \mathbf{Fix}$$

$$\frac{\Gamma \vdash E_1 : \mathbf{f}}{\Gamma \vdash \mathbf{if}(E_1, E_2, E_3) : \mathbf{f}}\ \mathbf{Cond-1} \qquad \frac{\Gamma \vdash E_2 : \phi\ \Gamma \vdash E_3 : \phi}{\Gamma \vdash \mathbf{if}(E_1, E_2, E_3) : \phi}\ \mathbf{Cond-2}$$

Other rules can added for other constants, so for $+$ which is strict in both arguments, we will add the rule:

$$\vdash + : \mathbf{f} \to \mathbf{t} \to \mathbf{f} \wedge \mathbf{t} \to \mathbf{f} \to \mathbf{f}$$

The relationship between abstract interpretation and the logic should now be apparent. In abstract interpretation, establishing strictness of some function $f'\ x\ y$ in each argument separately requires showing that $f'\ \mathbf{0}\ \mathbf{1} = \mathbf{0}$ and $f'\ \mathbf{1}\ \mathbf{0} = \mathbf{0}$. The two logical formulae $\mathbf{f} \to \mathbf{t} \to \mathbf{f}$ and $\mathbf{t} \to \mathbf{f} \to \mathbf{f}$ correspond directly to this.

An Example

As an example of an application of the above theory, consider proving the strictness of an expression:

$$\mathbf{Y}(\lambda f.\lambda x.\lambda y.\lambda z.\mathbf{if}(z = 0, x + y, f\ y\ x\ (x - 1)))$$

If we want to prove that this function is strict in x and y separately, then we want to prove the formula:

$$\mathbf{f} \to \mathbf{t} \to \mathbf{t} \to \mathbf{f} \wedge \mathbf{t} \to \mathbf{f} \to \mathbf{t} \to \mathbf{f}$$

We will abbreviate this formula with ψ, and let ψ_1 and ψ_2 denote either side of the

conjunction. Also, let E denote the term $\lambda x.\lambda y.\lambda z.\mathbf{if}(z = 0, x + y, f\ y\ x\ (x - 1))$. Let Γ denote the environment $[f : \psi, x : \mathbf{t}, y : \mathbf{f}, z : \mathbf{t}]$.

Then:

$$\dfrac{\dfrac{\Gamma \vdash x : \mathbf{t}\ \ \Gamma \vdash y : \mathbf{f}}{\Gamma \vdash x + y : \mathbf{f}}\quad \dfrac{\dfrac{\Gamma \vdash f : \psi}{\Gamma \vdash f : \psi_1}\quad \Gamma \vdash y : \mathbf{f}\ \ \Gamma \vdash x : \mathbf{t}\ \ \Gamma \vdash (x - 1) : \mathbf{t}}{\Gamma \vdash f\ y\ x\ (x - 1) : \mathbf{f}}}{\dfrac{\Gamma \vdash \mathbf{if}(z = 0, x + y, f\ y\ x\ (x - 1))\mathbf{f}}{[f : \psi] \vdash E : \psi_2}\ \mathbf{Abs, Abs, Abs}}\ \mathbf{Cond - 2}$$

Similarly, with Γ as $[f : \psi, x : \mathbf{f}, y : \mathbf{t}, z : \mathbf{t}]$,

$$\dfrac{\dfrac{\Gamma \vdash x : \mathbf{f}\ \ \Gamma \vdash y : \mathbf{t}}{\Gamma \vdash x + y : \mathbf{f}}\quad \dfrac{\dfrac{\Gamma \vdash f : \psi}{\Gamma \vdash f : \psi_2}\quad \Gamma \vdash y : \mathbf{t}\ \ \Gamma \vdash x : \mathbf{f}\ \ \Gamma \vdash (x - 1) : \mathbf{t}}{\Gamma \vdash f\ y\ x\ (x - 1) : \mathbf{f}}}{\dfrac{\Gamma \vdash \mathbf{if}(z = 0, x + y, f\ y\ x\ (x - 1))\mathbf{f}}{[f : \psi] \vdash E : \psi_1}\ \mathbf{Abs, Abs, Abs}}\ \mathbf{Cond - 2}$$

Combining these and proceeding further:

$$\dfrac{\dfrac{\dfrac{[f : \psi] \vdash E : \psi_1\quad [f : \psi] \vdash E : \psi_2}{[f : \psi] \vdash E : \psi}\ \mathbf{Conj}}{\vdash \lambda f.\lambda x.\lambda y.\lambda z.\mathbf{if}(z = 0, x + y, f\ y\ x\ (x - 1)) : \psi \to \psi}\ \mathbf{Abs}}{\mathbf{Y}(\lambda f.\lambda x.\lambda y.\lambda z.\mathbf{if}(z = 0, x + y, f\ y\ x\ (x - 1)))}\ \mathbf{Fix}$$

3.1.2 Relations

Previously, we have used concretisation and abstraction maps to relate the different interpretations given to a language. We will now introduce the use of logical relations to relate the interpretations and prove them correct. What follows is a synthesis of [Abr90] and [Hun91]. In order to present the results on relations, it is necessary to have a mathematical look at them first.

Mathematical Preliminaries

We will write $R : A \leftrightarrow B$ to mean that R is a relation between the sets A and B. Given $a \in A$ and $b \in B$, we write $a\ R\ b$ to mean that a and b are related with the relation R. $\mathcal{R}(A, B)$ denotes the set of all relations between the sets A and B.

If relations $P, Q \in \mathcal{R}(\mathcal{A}, \mathcal{B})$, and $a \in A, b \in B$, then we say that $P = Q$ when,

$$a\, P\, b \iff a\, Q\, b$$

and we order relations by implication, that is,

$$P \leq Q \iff a\, P\, b \Rightarrow a\, Q\, b$$

The product of two relations $P_1 : A_1 \leftrightarrow B_1$ and $P_2 : A_2 \leftrightarrow B_2$ is a relation $P_1 \times P_2 : A_1 \times A_2 \leftrightarrow B_1 \times B_2$ defined by

$$(a_1, a_2)\, P_1 \times P_2\, (b_1, b_2) \iff a_1\, P_1\, b_1 \text{ and } a_2\, P_2\, b_2$$

We are now in a position to define a logical relation and how it can be used to prove an analysis correct.

The Logical Relation and Its Use

In relating the abstract and standard interpretations in chapter 2, concretisation maps were constructed at each type. The approach that we present here is to create a special type of *relation* at each type, such that two elements are correct with respect to each other if they are related by the relation. With this in mind, we say that a relation between two interpretations I and J, $R : I \leftrightarrow J$, is a family $\{R_\sigma\}$ where $R_\sigma \subseteq D^I_\sigma \times D^J_\sigma$. The type of relations used are *logical* relations.

Definition 3.6 (Binary Logical Relation) *Given interpretations I and J, $R : I \leftrightarrow J$ is a family of binary relations indexed by types, $\{R_\sigma\}_{\sigma \in \mathcal{T}}$ with $R_\sigma : D^I_\sigma \leftrightarrow D^J_\sigma$. We call R logical if for all $\sigma, \tau \in \mathcal{T}$:*

- $f\, R_{\sigma \to \tau}\, g \iff$ *for each* $c \in D^I_\sigma, a \in D^J_\sigma, c\, R_\sigma\, a \Rightarrow f\, c\, R_\tau\, g\, a$
- $(c_1, c_2)\, R_{\sigma \times \tau}\, (a_1, a_2) \iff c_1\, R_\sigma\, a_1$ *and* $c_2\, R_\tau\, a_2$

The first point ensure that a logical relation relates functions which, when they are applied to related arguments, yield related results. The second point ensures that the products are related elementwise.

Our notion of what an *abstract interpretation* has now changed from the definition given in section 2.4. An abstract interpretation now consists of some finite interpretation $\mathbf{B}$, and a *logical* relation $R^{\mathbf{B}} : \mathbf{B} \leftrightarrow \mathbf{S}$.

As pointed out in Hunt [Hun91], there is an isomorphism

$$\mathcal{R}(A, B) \cong A \to \mathcal{P}(B)$$

which implies that there is an explicit relationship between the framework utilising powerdomains and concretisation maps, and that using logical relations. To see this, given a logical relation $R^{\mathbf{B}} : \mathbf{B} \leftrightarrow \mathbf{S}$ we can define a family of concretisation maps $Conc$, $\{\gamma_\sigma\}_{\sigma \in \mathcal{T}}$, by

$$\gamma_\sigma\ a = \{b \in D^{\mathbf{S}}_\sigma \mid b\ R_\sigma\ a\}$$

which are all of the form seen in the previous chapter, γ_σ: $D^{\mathbf{B}}_\sigma \to \mathcal{P}(D^{\mathbf{S}}_\sigma)$

Using the strictness analysis as a sample analysis, we can define logical relations for the base types as follows:

- $d\ R^{\mathbf{B}}_\iota\ 0 \iff d = \bot^S_\iota$
- $d\ R^{\mathbf{B}}_\iota\ 1 \iff d \in D^S_\iota$

If these definitions are expanded using the above equation for the isomorphism, we will get $\gamma_\iota\ 0 = \{b \in D^{\mathbf{S}}_\sigma \mid b\ R\ 0\} = \{\bot_\iota\}$. Likewise, $\gamma_\iota\ 1 = D^{\mathbf{S}}_\iota$ which is exactly what we had in the initial definition of concretisation maps in chapter 2. Of course, both $\{\bot^{\mathbf{S}}_\iota\}$ and $D^{\mathbf{S}}_\iota$ are Scott-closed sets.

The Binary Logical Relation Theorem

Proving correctness of $\mathbf{B}$ means showing that for every term e in $\Lambda_{\mathcal{T}}$, the standard interpretation of the term c and the abstract interpretation a of the term are always such that $c\ R^B_\sigma\ a$. This is done in [Abr90] by using the binary logical relation theorem due to Plotkin [Plo80]:

Definition 3.7 (The Binary Logical Relation Theorem) *Let $R : I \leftrightarrow J$ be a logical relation between two interpretations I and J. Suppose that $c^I\ R_\tau\ c^J$ for all τ and for all constants $c : \tau$, then for all σ, for all $e : \sigma$, for all $\rho \in Env^I, \rho' \in Env^J$:*

$$\rho\ R\ \rho' \Rightarrow [\![e]\!]^I \rho\ R_\sigma\ [\![e]\!]^J \rho'$$

The correctness of an interpretation $\mathbf{B}$, is then that whenever c and a are the standard and abstract interpretations of some term, then $c\, R^{\mathbf{B}}_{\sigma}\, a$ always holds. More formally, we will say that an abstract interpretation $\mathbf{B}$ is correct with respect to the standard interpretation, if for all types σ, and all terms of that type $e : \sigma$, and all $\rho \in Env^{\mathbf{S}}, \rho' \in Env^{\mathbf{B}}$:

$$\rho\, R^{\mathbf{B}}\, \rho' \Rightarrow [\![e]\!]^{\mathbf{S}} \rho\, R^{\mathbf{B}}_{\sigma}\, [\![e]\!]^{\mathbf{B}} \rho'$$

Since the relation is logical, the Binary Logical Relation Theorem implies that to prove this correctness condition it suffices that we prove all the constants correct only. Note how closely this resembles the correctness theorem of abstract interpretation presented in theorem 2.2, where we also needed to just prove the constants (shown in figure 2.4) correct.

To illustrate the correctness condition, we will show that this holds for our strictness example. Consider a function $f : \sigma \to \tau$ in the standard interpretation, with the abstract interpretation of the function being f'.
Now if f' is strict, that is $f'\, 0 = 0$, and we have a logical relation such that $f\, R^{\mathbf{B}}_{\sigma\to\tau}\, f'$, then if $\perp^{\mathbf{S}}_{\sigma}\, R^{\mathbf{B}}_{\sigma}\, 0$ we will have that $f\, \perp^{\mathbf{S}}_{\sigma}\, R^{\mathbf{B}}_{\tau}\, f'\, 0$ because $R^{\mathbf{B}}$ is logical. Now the only $c \in D^{\mathbf{S}}_{\tau}$ such that $c R^{\mathbf{B}}_{\tau} 0$ is $\perp^{\mathbf{S}}_{\tau}$, which allows us to conclude that $f \perp^{\mathbf{S}}_{\sigma} = \perp^{\mathbf{S}}_{\tau}$. So we have proved that a strict function in the abstract interpretation implies that the function will be strict in the standard interpretation. This is precisely our notion of correctness.

3.1.3 Partial Equivalence Relations

Partial equivalence relations are most easily introduced by example, and so we will follow [Hun91, HS91] and introduce them from a binding time analysis 'perspective'. A binding time analysis usually takes place before partial evaluation and involves determining which parts of a program are dependent solely on some given parameters. It can also be viewed as a dependency analysis. Given a set of functions and a description of *some* of the function's inputs that are to be fixed, we then compute those parts of the results that are determined by this input. Obviously, if the analysis is to be effective, we can only compute an approximation to the result. We call the parts of the input that are fixed, *static*; otherwise they are known as *dynamic*.

Consider a function $c : A \rightarrow B$ defined by

$$c(x) = b$$

where $b \in B$. That is, $c(x)$ is a constant function mapping its input to the same constant value b. If x is static then $c(x)$ will be static. If x is dynamic, $c(x)$ will still be static as it is a constant function. Thus this function is always static independent of its argument. We can express this by:

$$\forall x, x' \in A, c(x) = c(x')$$

Consider another function $fst : A \times B \rightarrow A$ defined by:

$$fst(x, y) = x$$

.

If x is static, then this function becomes static, and this is true whether or not y is fixed. An important thing to observe is that this is true no matter what value x actually takes. This can be succinctly expressed by:

$$\forall x \in A, \forall y, y' \in B. fst(x, y) = fst(x, y')$$

A more challenging function is $swap : A \times B \rightarrow B \times A$ defined by:

$$swap(x, y) = (y, x)$$

When the first element of the pair is static, so is the second element of the result, and vice versa. Thus, $\forall x, x' \in A, \forall y, y' \in B$

$$\begin{aligned} \pi_1(swap(x, y)) &= \pi_1(swap(x', y)) \\ \pi_2(swap(x, y)) &= \pi_2(swap(x, y')) \end{aligned}$$

Recall that the projection functions π_1 and π_2 are defined in section 1.4. We will now introduce equivalence relations which succinctly capture the above notions of constancy.

Equivalence Relations

For each set D we define *equivalence relations* $All_D, Id_D \subseteq D^2$ where for all $x, x' \in D$:

$$x \; All_D \; x'$$
$$x \; Id_D \; x' \iff x = x'$$

For a function $f : A \to B$ and relations $P \subseteq A^2$ and $Q \subseteq B^2$ we write $f : P \twoheadrightarrow Q$ iff

$$\forall x, x' \in A.\; x \; P \; x' \iff (f\,x)\; Q\; (f\,x')$$

Thus $f : P \twoheadrightarrow Q$ implies that whenever two elements are related by P, then their results (after application of f) are related by Q and vice versa. As before, we define the relation $P \times Q$ elementwise.

This allows us to express the constancy of the c function by:

$$c : All_A \twoheadrightarrow Id_B$$

and of *fst* by

$$fst : Id_A \times All_B \twoheadrightarrow Id_A$$

and of *swap* by:

$$swap : All_A \times Id_B \twoheadrightarrow Id_B \times All_A$$

and

$$swap : Id_A \times All_B \twoheadrightarrow All_B \times Id_A$$

In [HS91] it is proposed that the term *static* be associated with *Id*, and *dynamic* with *All*. The above equations then read quite easily. $swap : Id_D \times All_E \twoheadrightarrow All_E \times Id_D$ then reads as saying that given a static first argument, and dynamic second argument, *swap* then returns a pair whose first element is dynamic and second element is static.

An informal look at the per framework

We will now present an informal summary of the partial equivalence relation framework presented in [Hun91, HS91].

A *partial equivalence relation* is a binary relation which is symmetric and transitive. (It is thus a less restricted class of relations than the equivalence relations above as we have dropped reflexivity.) A *complete partial equivalence relation* P is *strict* if $\bot\ P\ \bot$, and *inductive* if for all chains $\{x_i\}_{i\in\omega}$, $\{y_i\}_{i\in\omega}$ such that if for all $i \in \omega$, $x_i\ P\ y_i$ then $(\bigsqcup_{i\in\omega} x_i)\ P\ (\bigsqcup_{i\in\omega} y_i)$.

We denote the class of all complete pers over a domain D by $\textsc{cper}(D)$. Hunt [Hun91] shows that for any domain D, $\textsc{cper}(D)$ is closed under intersection and forms a complete meet semi-lattice. The abstract interpretation framework shares many traits with that presented in the previous chapter. The purpose of the framework is to provide an effective test for statements of the form:

$$[\![e]\!] : P \twoheadrightarrow Q$$

We will denote the abstract interpretation using pers by the symbol $\mathbf{H}$ (for Hunt). As in the formulation for abstract interpretation in chapter 2, we will create an abstract domain $D_\sigma^{\mathbf{H}}$ as a finite lattice. For the purposes of the constancy analysis, the base domain is chosen as:

$$D_\iota^{\mathbf{H}} = \{\mathrm{D}, \mathrm{S}\} \text{ with } \mathrm{S} \sqsubset \mathrm{D}$$

Instead of associating with each element of the abstract domain a concretisation map which maps to subsets of the power domain of the concrete domain, we associate a 'concretising' per $\gamma_\sigma\ (a) \in \textsc{cper}(D_\sigma^{\mathbf{S}})$. We then choose the interpretations of the constants in such a manner so that for any term $e : \sigma \to \tau$ and $\forall a \in D_\sigma^{\mathbf{H}}, b \in D_\tau^{\mathbf{H}}$:

$$[\![e]\!]^{\mathbf{H}} a = b \Rightarrow [\![e]\!] :\gamma_\sigma\ (a) \twoheadrightarrow \gamma_\tau\ (b)$$

Thus if we perform the abstract interpretation and determine that $[\![e]\!]^{\mathbf{H}} a = b$, we can be sure that $[\![e]\!] :\gamma_\sigma\ (a) \twoheadrightarrow \gamma_\tau\ (b)$.

At the base types, Hunt defines the concretisation maps as: $\gamma_\iota\ (a) = \begin{cases} All_\iota & \text{if } a = \mathrm{D} \\ Id_\iota & \text{if } a = \mathrm{S} \end{cases}$

Thus, if the abstract interpretation framework determined that $[\![e]\!]^{\mathbf{H}} \mathrm{D} = \mathrm{S}$ then we can conclude that $[\![e]\!] : All \twoheadrightarrow Id$. That is, that if e is given a dynamic argument, it will return a static result.

3.2 Extensions to the Framework

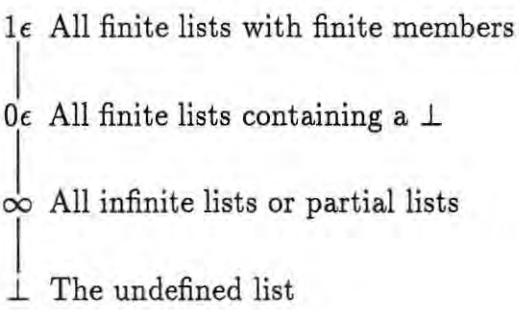
In the following sections we will investigate a technique developed to analyse functions over non-flat domains, and various aspects of the cartesian and tensor product.

3.2.1 Lists

The abstract interpretation presented in the previous chapter only allowed flat domains as the base types. We will now examine a method developed by Wadler [Wad87] which is used to analyse non-flat domains, in particular, lists. As is well known [Sch86], a list has a potentially infinite domain. An important aspect of Wadler's work is that it allows the analysis of lists using a finite abstract domain.

The domain proposed by Wadler captures the definedness of the list structure and also of all of the elements within the list. Represented by four points, Wadler proposes the abstract domain presented in figure 3.1 to describe the non-flat domain of lists of base types.

Figure 3.1 An abstract domain for lists of base types.



Here, a partial list is a list ending in bottom and a finite list is a list ending in Nil (We will write 'Nil' as opposed to []). We will make use of the function *cons* instead of the usual infix notation of ':'. As an example of the above description, the list *cons* 2 (*cons* ⊥ [])) would be represented by the value 0ϵ, and the list *cons* 2 (*cons* 2 ⊥) by the value ∞.

We will write the abstract version of the *cons* function as *cons'*. The following table is

a representation of the abstracted *cons* function with abstracted arguments:

	x'	
xs''	$\top$	$\bot$
1ϵ	1ϵ	0ϵ
0ϵ	0ϵ	0ϵ
∞	∞	∞
$\bot$	∞	∞

We would expect that the $Cons$ of a terminating element ($\top$) and a finite list containing bottom (0ϵ) will be a finite list containing bottom (0ϵ). From the table, we see that $Cons'\ \top\ 0\epsilon = 0\epsilon$ as expected. Similarly, $Cons'\ \top\ \infty = \infty$. That is, the $Cons'$ of a terminating element to an infinite list will yield an infinite list. The abstract form for the Nil constructor is just 1ϵ. It cannot be 0ϵ as this element reflects lists with a bottom element, and the Nil list obviously will not have one.

If our language has lists it will also need a way to decompose these lists, and Wadler presents a method to analyse functions presented in a case analysis style. We will thus write our functions in the following manner:

```
h l = case l of
        Nil        -> a
        Cons x xs  -> f x xs
      end
```

Suppose that we have an expression $length\ l$ and we know that the argument to *length* is a finite list which may contain $\bot$ as elements. That is, the definedness of l is 0ϵ. In order to determine the abstract value for the application, $length\ l$, we need to determine what x' and xs' could be if $Cons'x'xs'$ is 0ϵ. Looking up the values in the *cons'* table, we find that:

- x can be $\top$, xs can be 0ϵ, or
- x can be $\bot$, xs can be 0ϵ, or
- x can be $\bot$, xs can be 1ϵ

For the analysis to be safe, we must return the highest possible level of definedness for $h'\ 0\epsilon$, thus:

$$h'\ 0\epsilon = (f'\ \top\ 0\epsilon) \sqcup (f'\ \bot\ 0\epsilon) \sqcup (f'\ \bot\ 1\epsilon)$$

Because the abstract interpretation framework ensures that all of the functions are monotonic, we can reduce the above equation to:

$$h'\ 0\epsilon = (f'\ \top\ 0\epsilon) \sqcup (f'\ \bot\ 1\epsilon)$$

Repeating this analysis three more times, we arrive at the complete definition of the abstract form of h:

$$\begin{aligned} h'\ 1\epsilon &= a' \sqcup (f'\ \top\ 1\epsilon) \\ h'\ 0\epsilon &= (f'\ \top\ 0\epsilon) \sqcup (f'\ \bot\ 1\epsilon) \\ h'\ \infty &= f'\ \top\ \infty \\ h'\ \bot &= \bot \end{aligned}$$

As an application of this, we will consider analysing the *sum* function:

```
sum l = case l of
          Nil       -> 0
          Cons x xs -> x + sum xs
        end
```

This is abstracted as:

$$\begin{aligned} sum'\ 1\epsilon &= \top \sqcup (\top \sqcap (sum'\ 1\epsilon)) \\ &= \top \\ sum'\ 0\epsilon &= (\bot \sqcap (sum'\ 1\epsilon)) \sqcup (\top \sqcap (sum'\ 0\epsilon)) \\ &= sum'\ 0\epsilon \\ sum'\ \infty &= (\top \sqcap (sum'\ \infty)) \\ &= sum'\ \infty \\ sum'\ \bot &= \bot \end{aligned}$$

After fixedpoint iteration, we will find that *sum* has an abstract function:

$$\begin{aligned} sum'\,1\epsilon &= \top \\ sum'\,0\epsilon &= \bot \\ sum'\,\infty &= \bot \\ sum'\,\bot &= \bot \end{aligned}$$

which says that the *sum* function will only terminate if executed over a finite list with finite members (1ϵ).

3.2.2 Combining Properties

In chapter 2 we represented the tuppling in figure 2.3 as a cartesian product. This Section will illustrate that this is not necessarily the best representation. We begin by providing an example in which the cartesian product gives poor results. We then introduce some issues relating to faithfully modeling the properties of a constructed data type from the representation of its constituent parts, and examine the tensor product as an alternative representation. Finally, we will look at some applications of this technique.

To illustrate the poor results obtained when using a cartesian product, let us consider finding the strictness properties of the function:

$$\lambda b.\lambda x.\mathbf{if}\ b\,(x,2)\,(2,x)$$

If we interpret the tuppling as a product, then a pair $(0,1)$ in the abstract domain representing pairs of elements taken from some base type in our concrete domain would represent the set $\{(\bot_{D_t^{\mathbf{S}}}, x) \mid x \in D_t^{\mathbf{S}}\}$. That is, if this set is abstracted, it will be represented by the pair $(0,1)$. Likewise the pair $(1,0)$ represents the set $\{(x, \bot_{D_t^{\mathbf{S}}}) \mid x \in D_t^{\mathbf{S}}\}$.

A problem surfaces if we were to determine whether the function above is strict in x, as this would require us to represent the property that either the first value of the tuple may be bottom, or that the second value of the tuple may be bottom. Recall the rule for **if**.

That is, the set $\{(\bot_{D_t^{\mathbf{S}}}, x) \mid x \in D_t^{\mathbf{S}}\} \bigcup \{(x, \bot_{D_t^{\mathbf{S}}}) \mid x \in D_t^{\mathbf{S}}\}$. Because we are using the product, the point in the abstract domain which we use to represent this is $(0,1) \sqcup$

$(1,0) = (1,1)$, which represents $D_\iota^{\mathbf{S}} \times D_\iota^{\mathbf{S}}$, thus denying us the ability to state whether the above function is strict in x, (which of course it is). The property $(1,1)$ is a safe approximation, but we can hope for something better.

If we use a *tensor* product[1] instead, which we write as $\otimes$, then there is an additional point in the abstract domain, namely $0 \otimes 1 \sqcup 1 \otimes 0$ different from $1 \otimes 1$ which can represent the needed property.

Jones and Muchnick [JM81] were the first to distinguish between analyses able to derive information using different products. Analyses which use the cartesian product are said to use the independent attribute method, while those using the tensor product use a relational method.

3.2.3 Properties

In the framework derived in section 2.4, we chose to represent properties by Scott-closed sets. It was seen in section 3.1.3 that properties could also be represented by partial equivalence relations. We expand on this a little, pointing to literature where other forms of properties have been used.

In [MZ92] a different approach is taken in forming abstract domains. Instead of the user of the framework creating them, the approach generates the abstract domain from the concrete domain itself, with sufficient structure so as to capture useful properties. The properties are denoted by ideals, and the paper restricts abstract domains to being weak powerdomains.

In [Abr90], Abramsky develops a termination analysis (it can infer that certain programs definitely do terminate) which uses upper closed sets as representations of properties. This analysis is developed in an abstract interpretation framework with logical relations.

[1]The actual definition of the tensor product is rather complicated, and the reader is directed to [NN92].

3.3 Related Work

Wadler [Wad87] extended the framework of [BHA86] to analyse non-flat domains. Nielson [NN92] gives a more methodical approach to the analysis.

The framework was also extended to analyse polymorphic functions by Abramsky [Abr85]. The underlying mathematical framework has also developed considerably. Initially, denotational semantics, domain theory and power domains were the constructs used to build the framework [BHA85]. The theory has now been reformulated using logic [BL93], relations [MJ85], two-level semantics [Nie89] and partial equivalence relations [Hun91]. In almost all of these cases, the abstract interpretation framework itself has remained relatively constant. What has changed much are the various mathematical formulations of the framework, and their relative powers in deducing properties.

3.4 Conclusion

In this chapter we have looked at some of the extensions to the abstract interpretation framework. We began by looking at the framework embedded in various other disciplines, such as non-standard type systems, relations and partial equivalence relations.

In the non-standard type system, the type system was seen to be a program logic, defined over properties. The domains used in abstract interpretation were seen to form join-semilattices. The ideals of this lattice formed a meet-semilattice, and a logic was introduced to axiomatise the construction of this lattice. It was also shown that filters of this lattice would create a structure isomorphic to the original join-semilattice. This provided a framework for relating the non-standard type system to the domains used in abstract interpretation.

In the relational discipline, binary logical relations were introduced as a simple means of relating the abstract and standard interpretations. An isomorphism was shown to exist between relations and the powerdomain function approach demonstrated previously, implying directly that this is merely just a different formalism demonstrating the same thing. The crux of the system was the use of the binary logical relations theorem which allowed us to show correctness by just demonstrating the correctness of the constants.

The partial equivalence relations were introduced due to the inability of the foregoing systems to capture certain properties such as constancy for binding time analysis. Kamin [Kam92] has shown that the abstract interpretation framework presented in the previous chapter is unable to capture head strictness. (Informally, a list is head strict if the elements of the list are required to be defined. See the introductory paragraphs of the following chapter for a more thorough explanation.) Hunt's partial equivalence relation framework, however, *can* determine head strictness. The results of the previous sections shown that the relational, logical and powerdomain abstract interpretations are all equivalent in power, and so are not able to determine head strictness. It remains an open question as to exactly how powerful the per framework is.

Various other issues were also discussed, namely lists and properties. An analysis over a non-flat data structure was demonstrated by introducing a four point domain which represented different definedness levels on a list. Some important notions concerning properties were introduced, especially those relating to the tensor and cartesian product.

Chapter 4

Projection analysis

One of the weaknesses of the strictness analysis in the previous chapter is that it is unable to determine a particular type of strictness on list structures [Kam92], namely head strictness. Loosely, a function is said to be head strict if it examines the head elements of a list. As pointed out by Wadler [WH87], many functional programs operate with functions which read some of the input list and produce some of the output. These kind of functions are typically head strict, and not being able to detect this discards many optimization opportunities. A projection analysis however, can determine this kind of strictness. In investigating this framework, we will look deeper at the concept of head strictness and how domain projection functions can capture the definedness levels of a function. A notion of safety is also introduced, and finally the rules for performing a projection analysis are given.

We begin by informally introducing projection functions, building up to their formal description.

4.1 Projections, informally

We can, very loosely, think of a projection as a function which maps its argument to either itself, or something smaller (the function is less than the identity function), and that repeated applications of this function will yield the same value (the function is 'idempotent').

Consider two functions over pairs defined as follows for all u and v.

$$\begin{aligned} F(u,v) &= (u,\perp) \\ S(u,v) &= (\perp,v) \end{aligned}$$

An application of F or S to a pair will always yield something as defined, or less defined than the argument. Also, if the same function was re-applied to the result, the same result would be generated. Thus we can consider F and S as projections over pairs.

Let us consider another function, H, which is applied to a list. We define H as a function which accepts a list as an argument, and which replaces all occurrences of the list constructor ':' with a head strict version ':$_H$'. The list constructor $:_H$ is identical to :, except that it is strict in the head field. That is, it expects its first argument to be defined. If this argument is not defined, then the result is also not defined. Operationally, the $:_H$ evaluates its first argument at the time of building the list cell.

As an example,

$$H(1:2:\perp:4:[]) = 1:_H 2:_H \perp:_H 4:_H [] = 1:2:\perp$$

Obviously, if we apply H again to this result, we will get the same result. We now give the formal definition of head strictness.

Definition 4.1 (Head Strictness) *A function f is said to be* head strict *if $f = f \circ H$.*

The *before* function defined in figure 4.1 is an example of a head strict function. *before* examines the head of each cons cell to see if it is equal to zero, and this demands the evaluation of the head. Thus *before* is head strict. To illustrate this, let us consider two cases, one in which the zero occurs before the bottom, and one in which the zero occurs after the bottom. In the first case, we have

$$\begin{aligned} \mathit{before}\ (1:\perp:0:[]) &= 1:\perp \\ \mathit{before}\ (H(1:\perp:0:[])) &= \mathit{before}\ (1:\perp) = 1:\perp \end{aligned}$$

and in the second,

$$\begin{aligned} \mathit{before}\ (0:\perp:3:[]) &= [] \\ \mathit{before}\ (H(0:\perp:3:[])) &= \mathit{before}\ (0:\perp) = [] \end{aligned}$$

and each case satisfies definition 4.1.

Recall that strictness analysis could be used to eliminate the building of closures, leading to a (more efficient) eager evaluation strategy. If we are able to detect that a function is head strict, we can then replace the (lazy) cons operator with one which is eager in its first argument, leading to optimization.

Figure 4.1 Example programs *before* and *doubles* .

```
before xs  =case xs of
                []      ⇒ []
                y : ys  ⇒ if y == 0
                             then []
                             else y : before ys
doubles xs =case xs of
                []      ⇒ []
                y : ys  ⇒ (2 × y) : doubles ys
```

As another example, consider the function *doubles* shown in figure 4.1. This function is *not* head strict, because

$$doubles\ (1 : \bot : []) = 2 : \bot : []$$

which is not equivalent to

$$doubles\ (H(1 : \bot : [])) = doubles\ (1 : \bot\ ) = 2 : \bot$$

So for *doubles* we have that $doubles \neq doubles \circ H$. The essential difference between *doubles* and *before* , is that *before had* to evaluate y to test its value. Because we are dealing with a lazy functional language, *doubles* does not actually calculate the value of $2 \times y$ but inserts a closure for this expression which will only be evaluated if the value is demanded.

This begs the question, "Does the behaviour of *doubles* change when *its* result is demanded? That is, can the *context* in which a function appears, modify its behaviour?".

4.2 Contexts

The identity function, which we name ID , defined by

$$ID\ u = u$$

is also a projection, and is in fact the *greatest* projection (due to the fact that ID maps its argument to itself). The fact that it is the greatest projection hints at an ordering between the projection functions, and we will later encounter a complete lattice of projections which will aid in the finding of fixed points over domains of projections.

Projections can be used to show how much of an argument is required. In the words of Davis and Wadler [DW90],

> "Projections can be used to specify a degree of sufficient definedness of their arguments by regarding those parts of their arguments which are mapped to $\perp$ as definitely not needed, and those parts left unchanged as possibly needed."

It is this important property of projections that allow us to use them in an analysis. Let us consider the functions F and S defined in the previous section, and a function g (which we wish to analyse) defined by

$$g(u, v) = (v, u)$$

Suppose, in an expression, that we are only interested in the first component of the result of the application of g. That is, we do not look at the second component in the rest of the computation. This means that in an expression containing g, we can replace all occurrences of the function g with the function $F \circ g$. It is safe to do this, as we discard the second element. We say that the function g was evaluated in the *context* F . We can use this information in optimizing our program, because if we know that the second value is not required, we could perhaps change the code of g to not even generate that component.

The context of a function thus gives some indication of how defined the arguments of the function may be. Let us extend our example of $F \circ g$. Because we know that g is going to discard its second element we can write:

$$F \circ g = F \circ g \circ S$$

This can be read as follows. If we know that g is going to discard its second component, (the left hand side), then the output produced will be the same as first discarding the first element of the argument pair (because g swaps its arguments to produce the result) and applying F to this.

To see this, note that $F(g(u,v)) = (v,\perp)$ and that $F(g(S(u,v))) = F(g(\perp,v)) = (v,\perp)$. Thus for the function g, evaluated in the context of F , it is safe to first apply S to its arguments. So the projection F denotes the context in which the function is to be evaluated. An alternative reading of the above statement is that if we know that g is to be evaluated in a context which requires its result to be as defined as F, then we need only have g's arguments as defined as S to guarantee the result.

We will henceforth assign Greek letters α and β to projections. We also define some notation to capture the above concept.

Notation 4.1 *If, for projections α and β, and some function g we have that $\alpha \circ g = \alpha \circ g \circ \beta$, then we write this as $g : \alpha \Rightarrow \beta$*

The equation $\alpha \circ g = \alpha \circ g \circ \beta$ could be interpreted as saying that "if α's worth of output is demanded, then it is safe to demand β's worth of input" [HL91]. We will therefore call the above equation the *safety equation.*

Turning back to the functions *before* and *doubles*, we have already noted that *before* is head strict. The *before* function is head strict in any context. We associate the *ID* projection with 'any context' because this projection (context) makes no optimization assumptions about its arguments. We can thus begin thinking about different projections conveying to us different *levels* of information. In this case, *ID* gives us no information about the context of the function and we will soon find that the smaller the projection, the more information is conveyed.

Turning back to our example, we can now write $before : ID \Rightarrow H$. That is, $ID \circ before = ID \circ before \circ H$ which just reduces to our definition of head strictness.

We concluded that *doubles* was not head strict, but what happens when it is evaluated in a context other than *ID*, say in a head strict context? It is easy to verify that if *doubles* is evaluated in a head strict context, (in a context which expects the head argument to every cons cell to be defined), then we need have the input to *doubles* as defined as head strict. Put more formally, $H \circ doubles = H \circ doubles \circ H$. Since we

know that the *before* function is head strict, we have that

$$before \circ doubles = before \circ H \circ doubles = before \circ H \circ doubles \circ H$$

That is, if the result of *doubles* is used in a head strict context, then the argument to *doubles* can be provided in a head strict context. We can express this succinctly by $doubles : H \Rightarrow H$.

4.3 The Goal of a Projection Analysis

This leads us to the main goal of projection analysis. Given a projection which represents the context in which an expression is to be evaluated, we want to be able to determine a context in which each of the sub-expressions may be evaluated. In addition, this derived context must be *safe* in the sense of our safety equation.

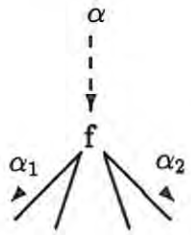
The propagation of information in a projection analysis is opposite to that found in abstract interpretation. Here we want to propagate information downwards, from the root of the tree to the leaves, as shown in figure 4.2.

In essence, we start with information about how the result of a computation will be used, and use this to derive information about how the arguments will be used. More accurately, we want to propagate variations of the context, so that the resulting projection satisfies our safety condition. Unary functions from projections to projections are called *projection transformers*. Given some projection α, and a function f, we want to find the β such that $\alpha \circ f = \alpha \circ f \circ \beta$. The β will take the form of $\tau\ \alpha$ where τ is the projection transformer. Moreover, we want the smallest projection β that satisfies the equation. Taking $\beta = ID$, that is, setting $\tau = \lambda x . ID$ will obviously always satisfy the equation, but smaller β will yield more useful information.

It is these *projection transformers* that are the holy grail of the projection analysis.

As in the abstract interpretation framework, a finite domain of projections ensures that we can effectively compute fixed points for recursive descriptions of projection transformers.

Figure 4.2 Backward Analysis of a function



The rest of this chapter introduces the theory behind these concepts, and is essentially orientated around the development of a technique which can be used to propagate projections, in a safe manner, through expressions.

4.4 Projections, formally

This section introduces the projections used to model the context analysis. The mathematical projections described here are based very closely on those described in section 4.1.

In domain theory, a continuous function α is a *projection* if, for every object u,

$$\alpha\, u \sqsubseteq u \tag{4.1}$$

$$\alpha\,(\alpha\, u) = \alpha\, u \tag{4.2}$$

In other words, a projection is an idempotent function less than the identity. This can also be expressed as

$$\alpha \sqsubseteq ID \tag{4.3}$$

$$\alpha \circ \alpha = \alpha \tag{4.4}$$

The identity function ID is the function defined by

$$ID\, u = u$$

for all u. The function BOT is defined by

$$BOT\, u = \bot$$

for all u, and projections form a complete lattice under the $\sqsubseteq$ ordering, with ID as the top element, and BOT as the least element.

As we have seen, the functions H, F and S are also projections. We now develop some of the mathematics surrounding projections so that we can apply these results in the development of the projection analysis proper.

The following definition is essentially a rewording of the piece of notation introduced earlier.

Definition 4.2 (Safety Equation) *A function f is β-strict in a context α if $\alpha \circ f = \alpha \circ f \circ \beta$, and we write this as $f : \alpha \Rightarrow \beta$.*

As before, we call this equation the *safety* equation. Given a context in which a function f is to be evaluated, say α, we want to be able to determine the projection transformer τ such that $\alpha \circ f = \alpha \circ f \circ (\tau\ \alpha)$. If this equation is satisfied, then we are sure that the projection $\tau\ \alpha$ applied to the arguments of the function does not affect the result.

A useful rewriting of $f : \alpha \Rightarrow \beta$ is shown in the next proposition.

Proposition 4.1 *$f : \alpha \Rightarrow \beta$ iff $\alpha \circ f \sqsubseteq f \circ \beta$*

We also show a composition result, which will prove useful in the propagation of some contexts.

Proposition 4.2 *If $f : \alpha \Rightarrow \beta$ and $g : \beta \Rightarrow \gamma$ then $f \circ g : \alpha \Rightarrow \gamma$*

Thus for $before : ID \Rightarrow H$ and $doubles : H \Rightarrow H$ we can write $before \circ doubles : ID \Rightarrow H$.

The set of all projections over a domain will form a complete lattice, and this allows us to find fixed points.

Unfortunately, the greatest lower bound of two projections $\alpha$and β cannot be defined by the greatest continuous function smaller than α and β as this is not necessarily a projection (see [WH87] for an example), and so we define the greatest lower bound as

Definition 4.3 (Greatest Lower Bound) *The greatest lower bound of a set of projections is defined by $\bigsqcap A = \bigsqcup\{\beta \mid \text{for all } \alpha \in A, \beta \sqsubseteq \alpha\}$*

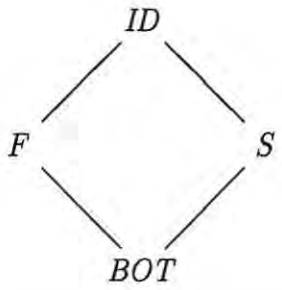
The greatest lower bound of projections α and β is thus the greatest projection less than than both α and β.

For our projections F and S , we then have:

$$\begin{aligned}(F \sqcup S)(u,v) &= (u,v) &= ID\\ (F \sqcap S)(u,v) &= (\bot,\bot) &= BOT\end{aligned}$$

which gives us the following lattice of projections.

Figure 4.3 The lattice of projections F , S , ID and BOT



4.5 Capturing Strictness

The above tools do not capture precisely the notion of strictness. Their failing is that the form of the projections only allow us to express *sufficiency* and not *necessity*. In other words, the projections as defined above can only tell how much of the argument is sufficient for a particular result. For the purposes of strictness analysis, we need to know how much of an expression is *necessary*. If we consider the *before* function which is head strict($f = f \circ H$), we know that it is *sufficient* to use $1 : 2 : \bot$ instead of the list $1 : 2 : \bot : 4 : []$ as $H(1 : 2 : \bot : 4 : []) = 1 : 2 : \bot$. To determine strictness information, it turns out that we need to know that it is *necessary* that the value is more defined than $\bot$, and as yet there is no machinery available to convey this type of information.

To accomplish this, we create a projection, which we will call STR (strict), which must only accept objects that are more defined than $\bot$. Unfortunately, we have $\bot$ as the

least element of our domain already, and so we need to introduce another, even less defined element, which we will call *abort* and write as $\lightning$. (More formally, we *lift*[1] the domain.) We thus define STR as:

$$\begin{aligned} STR \ \lightning &= \lightning \\ STR \ \bot &= \lightning \\ STR \ u &= u \text{ otherwise} \end{aligned}$$

Thus STR is a projection, that is strict in *abort*. Moreover, it accepts any other value except for $\bot$, putting us in position to be able to express strictness. As mentioned before, we can view projections as specifying a degree of definedness. Anything that is mapped to abort is definitely not needed, and STR indicates that it requires its argument to be more defined that $\bot$. The lifting of the domain has thus allowed us to say that it is *necessary* that a function be more defined than $\bot$. Theorem 4.1 captures this.

We also define a new function, $FAIL$, which maps its argument to $\lightning$.

$$FAIL\, u = \lightning \text{ for all } u$$

The $\lightning$ is less than any other element in our domain, including the $\bot$, thus whereas before BOT was our least projection, we now have $FAIL$.

Our old projection BOT , we will now call ABS (absent) and is defined by

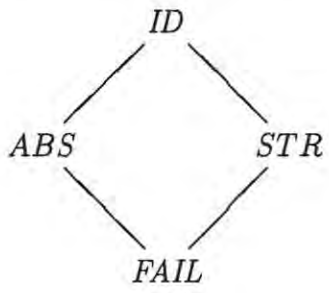
$$\begin{aligned} ABS \ \lightning &= \lightning \\ ABS \ u &= \bot \text{ if } u \neq \lightning \end{aligned}$$

Intuitively, if an expression is labeled with this context then we know that the value of the expression is ignored, whereas if the expression was labeled with STR , then we know that the expression is going to be needed. If it is labeled with ID then we do not know anything. (See section 4.5.1)

The four projections ID , STR , ABS , $FAIL$ form a complete lattice as shown in figure 4.4 and each projection plays a big part in describing the properties of interest. ID is obviously the largest projection, and $FAIL$ the smallest. Both ABS and STR are more defined than $FAIL$, and less defined than ID , and they are incomparable.

[1]See definition 1.11

Figure 4.4 The projections *ID* , *STR* , *ABS* , *FAIL*



Definition 4.4 *A value u is* unacceptable *to a context α if $\alpha\, u = \lightning$*

With the above definition, *all* values are unacceptable to *FAIL* , whereas only those more defined than $\bot$ are acceptable to *STR* .

Now we present a propagation result for acceptability in contexts.

Proposition 4.3 *If $f : \alpha \Rightarrow \beta$ and u unacceptable to β, then fu unacceptable to α.*

We are now, at last, in a position where we can define simple strictness. Having the *abort* has allowed us to talk about necessity and we can now make statements about $\bot$ and strictness.

Theorem 4.1 *$f : STR \Rightarrow STR$ iff f is strict*

Proof In the forwards direction, the only value unacceptable to *STR* is $\bot$ and so by the previous result, we have that $f\,\bot$ is unacceptable to *STR* , and so f is strict. In the backward direction we have that f is strict, and we must show that $f : STR \Rightarrow STR$, that is $STR(f\,u) \sqsubseteq f(STR\,u)$ for all u. There are two cases to consider. If $u \neq \bot$ then $STR\,u = u$ and $STR(f\,u) \sqsubseteq f\,u$ as *STR* is a projection. If $u = \bot$ then the the left hand side is $STR(f\,\bot) = STR\,\bot = \lightning$ and the right hand side is $f(STR\,\bot) = f\,\lightning = \lightning$ □

To determine whether a function is strict or not we need to perform a context analysis of the function in a *STR* context. If we determine that the result is *STR* , that is, $f : STR \Rightarrow STR$, then by the above theorem we know that the function is strict.

It can also be shown that $f : FAIL \Rightarrow FAIL$ and $f : ABS \Rightarrow ABS$ for all functions f. Other useful results are that $f : STR \Rightarrow FAIL$ if $f\,u = \bot$ for all u, and $f : STR \Rightarrow ABS$

if $f\ u = f\perp$ for all u. If $f\ u = f\perp$ for all u then we say that f ignores its argument.

4.5.1 Interpretations of Projections

We can now collect all of the interpretations that we have given to the projections. We will read a context of an expression as defining the *demand* on the expression. In other words, the context tells us to what degree an expression evaluation is to be performed [DW90].

If an expressions e has been labeled with one of the following contexts, then we interpret them as follows

FAIL No degree of evaluation yields an acceptable value for this context. We can implement this expression with code that aborts.

ID This context gives us no information about e, and we implement this by the (lazy) manner of constructing a graph for the expression.

ABS This context indicates that the result from the expression evaluation is not demanded (as it maps the value of the expression to $\perp$ for all values), and so we can implement this expressions by returning a dummy value.

STR This context requires evaluation of the expression far enough to guarantee that the result is not $\perp$, and can be implemented by evaluating the expression to weak head normal form [Pey86].

We will follow Davis [DW89] and reinterpret these projections in terms of safety in the following section.

4.6 Context Analysis

We introduce a first-order language on which we will perform the context analysis:

$e ::= x$		variables
$\mid$	c	constants
$\mid$	$f e_1 \ldots e_n$	function applications
$\mid$	**if** e_0 **then** e_1 **else** e_2	conditional

with function definitions having the form

$$f\, x_1 \ldots x_2 = e$$

Definition 4.5 (Projection Transformer) *A* projection transformer *is a function from projections to projections. Thus a projection transformer applied to a projection, will yield another projection.*

If we have a function of one argument, and a projection α, we want to ensure that $f : \alpha \Rightarrow \beta$ holds, where β is the projection which is applied to the argument of f. In general we deduce β from α, and this is done by finding a suitable projection transformer f^1 which when applied to α, yields β. Thus $\beta = f^1\alpha$.

What we deduce in this section is a set of rules (forming the context analysis), which allows us to find projection transformers and propagate projections through the terms of the language above.

Extending the safety equation to a function of n arguments, we want the following to hold:

$$\alpha\,(f\, u_1 \ldots u_i \ldots u_n) \sqsubseteq f\, u_1 \ldots (\beta_i u_i) \ldots u_n$$

for all $u_1 \ldots u_n$, where $\beta_i = f^i\alpha$.

It is easy to show that the safety requirements for all $f^i \ldots f^n$ are satisfied iff

$$\alpha\,(f\, u_1 \ldots u_n) \sqsubseteq f(\beta_1 u_1) \ldots (\beta_n u_n)$$

for all $u_1 \ldots u_n$, where $\beta_i = f^i\alpha$.

We do likewise for expressions, defining a transformer e^x for each free variable x in e, which takes a projection applied to the result of e to a projection that may safely be

applied to each instance of x in e. Thus we have the following safety requirement.

$$\alpha\, e \sqsubseteq e[(\beta\, x)/x]$$

where $\beta = e^x\alpha$ and this equation holds for all values of the variables in e.

In a denotational setting, we would write:

$$\alpha\,(E[\![e]\!]\rho\,) \sqsubseteq E[\![e]\!](\rho\,[(\beta\,(\rho\,[\![x]\!]))/x])$$

for each environment ρ.

This equation is exactly the same as the safety equation we introduced earlier, except that it is now phrased in denotational semantics.

We now need to define the f^i and e^x which satisfy the above safety conditions. We initially consider the definitions for the function transformers for non-primitive f. If we have f defined as

$$f\, x_1 \ldots x_n = e$$

then for each i from 1 to n we define

$$f^i\, \alpha = e^{x_i}\alpha$$

We read the last equation as: Evaluating the function f in some context α will cause its *ith* argument to be evaluated in context $e^{x_i}\alpha$. Thus e^{x_i} is the projection transformer that transforms α. If our expression is recursive, we can expect recursive representations of the projection transformers.

Thus we create a projection transformer for each argument to the function, named by superscripting the functions name with the arguments position. We will later develop a rule for combining these transformers.

The last equation above reduces searching for a transformer for a function, to finding a set of transformers for the expression (based on each argument).

Consider an expression consisting of just the variable y, and a projection α which will be applied to this expression. If we are performing our analysis with respect to some variable x, different from y, then we can safely replace α with ABS , as the value is essentially ignored. The same rule applies to any constant c. If however, the expression is x, we can only safely propagate α.

To summarize, we define the transformer e^x by the following rules:

$$\begin{aligned} x^x\alpha &= \alpha \\ y^x\alpha &= ABS \quad \text{if } x \neq y \\ c^x\alpha &= ABS \quad \text{if } c \text{ is a constant} \end{aligned}$$

These can be read as: Evaluating x in context α causes x to be evaluated in context α. Similarly, evaluating y in context α causes x to be ignored.

We can now try our new techniques on a simple function, the function K defined by $K\ x\ y = x$. We want to generate the two projections transformers (K^1 and K^2) for this function K, when evaluated in a context α. By the above definitions, this reduces to finding the transformers over the expression, giving us:

$$\begin{aligned} K^1\alpha &= x^x\alpha &= \alpha \\ K^2\alpha &= x^y\alpha &= ABS \end{aligned}$$

which can be interpreted to say, that for the first argument we have $K : \alpha \Rightarrow \alpha$ and for the second $K : \alpha \Rightarrow ABS$. In other words, when K is evaluated in a context α, then the first argument can be evaluated in context α, and the second can be ignored.

4.6.1 The Guard Property

We will now use some of the propositions in section 4.5 to help us in defining the projection transformer e^x.

We have already seen that $f : FAIL \Rightarrow FAIL$ holds for all functions, and so it is safe to set $e^x FAIL = FAIL$. Likewise, $f : ABS \Rightarrow ABS$ holds for all functions and it is safe to set $e^x ABS = ABS$.

We call the strict part of a projection α, $STR \sqcap \alpha$, and if α is equal to its strict part then α is called a *strict* projection. Likewise, the non-strict form of a projection α is $\alpha \sqcup ABS$. As explained in [WH87, DW89], we can restrict the projection analysis to a strict context, as for all f and projections α, if we have that $f : \alpha \sqcap STR \Rightarrow \beta$ then $f : \alpha \sqcup ABS \Rightarrow \beta \sqcup ABS$. In defining the $e^x\ \alpha$ we can use these facts to simplify

matters, and so we introduce the guard operator, $\rhd$, defined by:

$$\begin{array}{lcl} FAIL \rhd \beta & = & FAIL \\ ABS \rhd \beta & = & ABS \\ \alpha \rhd \beta & = & \beta \text{ if } \alpha \text{ strict and not } FAIL \\ (ABS \sqcup \alpha) \rhd \beta & = & ABS \sqcup \beta \text{ if } \alpha \text{ strict and not } FAIL \end{array}$$

From the above we can see that it is safe to set:

$$e^x\ \alpha = \alpha \rhd e^x\ \alpha'$$

We use this rule implicitly and so assume in all others that α is strict and not $FAIL$.

If f is some primitive function such as $(+)$ and $(=)$ which is strict in all of its arguments, we set:

$$f^i\ \alpha = \alpha \rhd STR$$

4.6.2 Combining Projections

If we are to analyse function application, we will need some way to combine projections. To see this, consider a function $f\ e_1\ e_2$ and a context α. If we want to find $(f\ e_1\ e_2)^x\ \alpha$, then we need to find a δ such that:

$$\alpha\ (f\ e_1\ e_2) \sqsubseteq (f\ e_1\ e_2)[\delta\ x/x]$$

From the definition of f^i above, we know that

$$\alpha\ (f\ e_1\ e_2) \sqsubseteq f\ (f^1\ \alpha\ e_1)(f^2\ \alpha\ e_2)$$

and assuming that we are finding the projection with respect to the variable x occurring in e, we have from the definition of e_i^x that

$$f\ (f^1\ \alpha\ e_1)(f^2\ \alpha\ e_2) \sqsubseteq f(e_1[(\beta_1\ x)/x])(e_2[(\beta_2\ x)/x])$$

where $\beta_1 = e_1^x\ f^1\ \alpha$ and $\beta_2 = e_2^x\ f^2\ \alpha$

We need to find a δ for the entire application such that

$$f\ (e_1[(\beta_1\ x)/x])(e_2[(\beta_2\ x)/x]) \sqsubseteq (f\ e_1\ e_2)[(\delta x)/x])$$

Clearly, the projection defined by $\beta_1 \sqcup \beta_2$ satisfies this criteria, remembering that a 'greater' projection is a safe approximation for a smaller one. We also have the condition that if an argument to any of these projections is abort, then their result is also abort, thus we define our projection to be $\beta_1 \,\&\, \beta_2$ where we define the operator & as follows:

Definition 4.6 (The & Operator)

$$\begin{array}{lcll} (\alpha \,\&\, \beta)\ u & = & \lightning & \textit{if } \alpha\, u = \lightning \textit{ or } \beta\, u = \lightning \\ (\alpha \,\&\, \beta)\ u & = & (\alpha \sqcup \beta)\, u & \textit{otherwise} \end{array}$$

What we have shown then, is that

$$(f\ e_1\ e_2)^x \alpha = e_1^x(f^1\ \alpha) \,\&\, e_2^x(f^2\ \alpha)$$

4.6.3 Properties of &

It can easily be shown that & satisfies the following properties:

$$\begin{array}{rcl} \alpha \,\&\, \alpha & = & \alpha \\ ABS \,\&\, \alpha & = & \alpha \\ ABS \,\&\, STR & = & STR \\ FAIL \,\&\, \alpha & = & FAIL \\ \alpha \,\&\, (\beta \sqcup \delta) & = & (\alpha \,\&\, \beta) \sqcup (\alpha \,\&\, \delta) \\ (\alpha \,\&\, \beta) \,\&\, \delta & = & \alpha \,\&\, (\beta \,\&\, \delta) \\ \alpha \,\&\, \beta & = & \beta \,\&\, \alpha \end{array}$$

All of these are quite easy to show. As an example, we will prove that $ABS \,\&\, STR = STR$. To do this, we have to show that for all possible values of u, $(ABS \,\&\, STR)\, u = STR\, u$.

Because all functions are strict in abort, this obviously holds if $u = \lightning$. If $u = \bot$, then the right hand side reduces to abort by definition of STR, and the left hand side reduces to abort as well because $STR \bot = \lightning$ and by the definition of &. If u is a value other than $\bot$ or $\lightning$, then the right hand side reduces to u, and the left to $STR u \sqcup ABS u$ which is $u \sqcup \bot$ which is u.

We now give the rule for the conditional, derived in [WH87]:

$$(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2\ )^x \alpha\ = e_0^x\ STR\ \&\ (e_1^x\ \alpha\ \sqcup\ e_2^x\ \alpha\ )$$

which can be read as saying that when the conditional statement is evaluated under context α, then in e_0, x will be evaluated under STR, and then either x will be evaluated under α in e_1 or under α in e_2.

4.6.4 Examples

We will now return the the constant function K introduced in section 4.6, and try to analyse the function in a STR context, with respect to each of the variables x and y.

First, we do the analysis with respect to x. Then, we have:

$$\begin{aligned}(K\ x\ y)^x\ STR &= x^x(K^1\ STR)\ \&\ y^x(K^2\ STR)\\ &= x^x(STR)\ \&\ y^x(ABS)\\ &= STR\ \&\ ABS\\ &= STR\end{aligned}$$

which tells us that K is strict in its first argument, as expected. Performing the analysis with respect to y, we get:

$$\begin{aligned}(K\ x\ y)^y\ STR &= x^y(K^1\ STR)\ \&\ y^y(K^2\ STR)\\ &= x^y(STR)\ \&\ y^y(ABS)\\ &= ABS\ \&\ ABS\\ &= ABS\end{aligned}$$

which tells us that K ignores its second argument.

To show the use of fixed points, we will now consider a recursive function given by:

$$f\ x\ y = \textbf{if } x = 0 \textbf{ then } x \textbf{ else } f\ x\ 3$$

Calculating f^1 we get:

$$\begin{aligned} f^1\,\alpha &= (\textbf{if } x = 0 \textbf{ then } x \textbf{ else } (f\ x\ 3)\ )^x\alpha \\ &= (x = 0)^x\ STR\ \&\ (x^x\alpha \sqcup (f\ x\ 3)^x\alpha\ ) \\ &= x^x(=^1\ STR)\ \&\ 0^x(=^2\ STR)\ \&\ (\alpha \sqcup x^x\ (f^1\alpha\ )\ \&\ 3^x\ (f^2\alpha\ )) \\ &= (STR \rhd STR)\ \&\ ABS\ \&\ (\alpha \sqcup f^1\alpha\ \&\ ABS) \\ &= STR\ \&\ (\alpha \sqcup f^1\alpha\ ) \end{aligned}$$

which we have to now solve by fixed point iteration. Since we are looking for strictness information, we set $\alpha = STR$, and get:

$$\begin{aligned} f^{1(0)} &= FAIL \\ f^{1(1)} &= STR\ \&\ (STR \sqcup FAIL) \\ &= STR\ \&\ STR \\ &= STR \\ f^{1(2)} &= STR\ \&\ (STR \sqcup STR) \\ &= STR \end{aligned}$$

We thus conclude that the function is strict in the first argument. For the second argument, we get:

$$\begin{aligned} f^2\,\alpha &= (\textbf{if } x = 0 \textbf{ then } x \textbf{ else } (f\ x\ 3)\ )^y\alpha \\ &= (x = 0)^y\ STR\ \&\ (x^y\alpha \sqcup (f\ x\ 3)^y\alpha\ ) \\ &= ABS\ \&\ (ABS \sqcup (x^y(f^1\alpha\ )\ \&\ 3^y(f^2\alpha\ )) \\ &= ABS \end{aligned}$$

No fixed point iteration is needed here, and we can conclude that if the function is evaluated in any context then the second argument is ignored.

4.7 Related Work

The presentation of the projection analysis in this chapter has followed that of Wadler and Hughes [WH87] very closely. Their paper considers projection analysis for a first-

order monomorphic language, and also includes extensions of the analysis to cater for lists. Hughes [Hug89a] has subsequently shown how to handle polymorphism in a projection analysis and also produced a higher order projection analysis in [Hug87]. Davis and Wadler [DW90] have extended the work of [WH87] to create a high fidelity (it can find strictness in more than one argument at a time) analysis.

In [Bur90b], Burn shows how the projection analysis results can be used in compiling a lazy functional language by introducing evaluators as a means of specifying when the evaluation order can be changed.

4.8 Conclusion

In this chapter we introduced projections as a technique for capturing strictness information. The important concepts highlighted in this chapter include:

Definedness Projections were seen to capture the definedness level of an expression, by mapping that which was definitely not needed to $\perp$.

Contexts Projections specified contexts, and by using this information regarding the definedness of the context in which an expression was to be evaluated, it was possible to derive information about the necessary definedness of the sub-expressions.

Safety equation The safety equation is a precise description of the above point, specifying that the level of definedness induced upon a sub-expression was not to change the result of the evaluation of the expression in a particular context.

Context Analysis A context analysis was introduced by a set of rules illustrating how the projection specifying the context of an expression could be changed to specify the context of the sub-expressions. That is, the context analysis was a specification of the projection transformers.

Chapter 5

Implementation Issues

The previous chapters have all focused on the theoretical aspects of various analysis techniques. We will now look at some theoretical issues relating to the implementation of the analyses.

Because solving for recursive functions requires finding a fixed point of a functional, the fixed point operator is present in all of the analysis frameworks that have been presented. This operator has always been interpreted as the least upper bound of the ascending Kleene chain:

$$\mathbf{Y}^{\mathbf{S}}_{\sigma} f = \bigsqcup_{i \in \omega} f^i \perp^{\mathbf{S}}_{\sigma}$$

We thus build up a sequence of approximations to a function.

If we had a recursive definition for some function $f : \sigma \to \tau$ given by:

$$f\, x = \ldots f \ldots$$

we would build a chain:

$$\begin{array}{lcll} f^0\, x & = & \perp & \text{The base of the recursion} \\ f^1\, x & = & \ldots f^0 \ldots & \text{The first approximation} \\ f^2\, x & = & \ldots f^1 \ldots & \end{array}$$

Because the sequence $f^0, f^1, \ldots$ is increasing, and we have ensured that the lattice $[D_\sigma \to D_\tau]$ is of finite height, a fixed point will definitely be reached. Obviously, if $f^x = f^{x-1}$, then any greater x will result in the same value, and we have found a fixed

point. The problem therefore reduces to finding out when two successive functions in the chain are equal.

To compare two functions, we have to compare their entire graphs. This is obviously a very expensive operation. If we consider strictness analysis, where our base domains have only two elements, this effectively implies that we have to build a truth table for each function. Since the number of points (rows) is 2^n for an n-argument function, the cost of the comparisons grow exponentially with the number of arguments.

Important questions to consider are then:

- Can we optimize the comparison of the function graphs? This optimization will obviously improve the efficiency of the analysis.

- Can we represent the graph of a function more efficiently? The function $cat\ l = foldr\ append\ l\ []$ invokes $foldr$ at an instantiation of $[[\mathbf{4} \to \mathbf{4} \to \mathbf{4}] \to \mathbf{6} \to \mathbf{4} \to \mathbf{4}]$ which has well over a million elements in its argument domain [HH91]. (Recall that if we worked with the simplest type which has domain $\mathbf{2}$, we would by the results of chapter 3 use $(\mathbf{2}_\perp)_\perp = \mathbf{4}$ for the representations of the lists.) A more efficient representation of the graph would thus decrease the resource requirements of the analysis.

- Can we determine upper and lower bounds for the number of iterations needed before a fixed point is found? If an upper bound existed then a naïve approach to finding the fixed point would be to simply iterate that many times, eliminating entirely the need to perform any equality checks between successive iterations. The Nielsons [NN91] prove various results relating to the finding of bounds for fixed point iteration in various types of lattices.

- Can we put bounds on the true fixed point and find some approximation to it, yielding safe but approximate results. This would allow us to balance the time spent on finding fixed points and the resolution of the analysis.

The following sections answer these questions.

5.1 Pending Analysis

The pending analysis is based on the simple observation that if in the evaluation of fx we find that the result depends on that same fx again, then we can return $\perp$ as the result of the second call. This result is proved correct in [YH86] with respect to the semantics of recursive monotone boolean functions, that is, those functions that have the form $f(x_1, \ldots, x_n) = body$ where *body* is an expression of the form:

$$exp ::= \mathbf{0} \mid \mathbf{1} \mid exp_1 \sqcup exp_2 \mid exp_1 \sqcap exp_2 \mid x_i \mid f(exp_1, \ldots, exp_n)$$

The functions that we find in abstract interpretation are often of this form.

Consider a function,

$$f(x_1, x_2, x_3) = (x_1 \sqcap x_3) \sqcup (f(x_2, \mathbf{1}, \mathbf{0}) \sqcap f(x_3, x_1, \mathbf{1}))$$

For a particular argument permutation, we can recursively evaluate this expression:

$$\begin{aligned} f(\mathbf{1},\mathbf{1},\mathbf{0}) &= (\mathbf{1} \sqcap \mathbf{0}) \sqcup (f(\mathbf{1},\mathbf{1},\mathbf{0}) \sqcap (f(\mathbf{0},\mathbf{1},\mathbf{1})) \\ &= \mathbf{0} \sqcup (\mathbf{0} \sqcap f(\mathbf{0},\mathbf{1},\mathbf{1})) && \text{on application of the theorem} \\ &= \mathbf{0} \sqcap f(\mathbf{0},\mathbf{1},\mathbf{1}) \\ &= \mathbf{0} \end{aligned}$$

Similarly,

$$\begin{aligned} f(\mathbf{0},\mathbf{1},\mathbf{1}) &= (\mathbf{0} \sqcap \mathbf{1}) \sqcup (f(\mathbf{1},\mathbf{1},\mathbf{0}) \sqcap f(\mathbf{1},\mathbf{0},\mathbf{1})) \\ &= \mathbf{0} \sqcup (\mathbf{0} \sqcap f(\mathbf{1},\mathbf{0},\mathbf{1})) && \text{from the previous result} \\ &= \mathbf{0} \sqcap f(\mathbf{1},\mathbf{0},\mathbf{1}) \\ &= \mathbf{0} \end{aligned}$$

and

$$\begin{aligned} f(\mathbf{1},\mathbf{0},\mathbf{1}) &= (\mathbf{1} \sqcap \mathbf{1}) \sqcup (f(\mathbf{0},\mathbf{1},\mathbf{0}) \sqcap f(\mathbf{1},\mathbf{1},\mathbf{1})) \\ &= \mathbf{1} \end{aligned}$$

An important observation of the above is that we can optimize even more by memoizing [Hug85] the result of all the evaluations. In the above example we used the result of $f(\mathbf{1},\mathbf{1},\mathbf{0})$ instead of re-evaluating it. The main disadvantage of this technique is that there is no obvious way to extend it to higher order functions.

5.2 Frontiers

The frontier representation was developed by Clack and Peyton Jones [CP85] in an attempt to represent the graph of a function more efficiently by exploiting the monotonicity of the abstract functions. Although [CP85] was only concerned with monotone functions of the form $f : \mathbf{2}^n \to \mathbf{2}$, the representation applies to all monotone functions of the form $f : X \to \mathbf{2}$ for any finite domain X.

In calculating the Kleene chain, we need to keep successive graphs of functions to compare. One way to represent the graph of a function $f : \mathbf{2} \to \mathbf{2}$ is by the set:

$$\{(x, y) \in \mathbf{2} \times \mathbf{2} \mid f(x) = y\}$$

but this leads to the efficiency problems mentioned earlier.

We can improve on this by representing the function graph by the set,

$$\{x \in \mathbf{2} \mid f(x) = \mathbf{1}\}$$

because if $f(x) \neq \mathbf{1}$ then $f(x) = \mathbf{0}$.

This too can be improved upon; because f is monotonic, we know that

$$x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

Using this information, if we know that $x \sqsubseteq y$, and $f(x) = f(y) = \mathbf{1}$, then we need only record the fact that $f(x) = \mathbf{1}$ because by monotonicity we are sure that any y such that $x \sqsubseteq y$ means that $f(x) \sqsubseteq f(y)$ implying that $f(y)$ must be $\mathbf{1}$. We can thus refine our previous graph representation to:

$$Min\{x \in \mathbf{2} \mid f(x) = \mathbf{1}\}$$

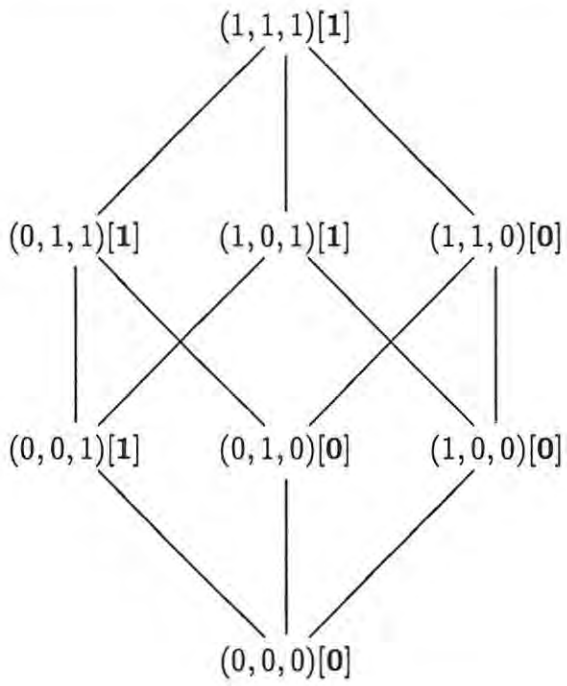
where $Min(X)$ are the minimal elements of X, defined by:

$$Min(X) = \{x \in X \mid \text{ for each } y \in X, y \sqsubseteq x \Rightarrow y = x\}$$

We call the resulting set a *frontier*. If the set is composed of those points that evaluate to $\mathbf{1}$, we call the set a $\mathbf{1}$-frontier. Likewise, the $\mathbf{0}$-frontier is composed of maximal (rather than minimal) elements of $\{x \in \mathbf{2} \mid f(x) = \mathbf{0}\}$.

As an illustration, if we had a function $f : \mathbf{2}^3 \rightarrow \mathbf{2}$, the arguments would form a lattice. Figure 5.1 shows this lattice with the arguments in parentheses and an example output resulting from applying f to the argument within square brackets.

Figure 5.1 Lattice of arguments for $f : \mathbf{2}^3 \rightarrow \mathbf{2}$



If the result of applying the function to an argument results in a **0**, we will call the argument node a **0**-node. Similarly for a **1**-node.

In the naïve representation, this whole lattice of values would have to be kept as a representation of the graph of the function.

If we represented the graph by all of the **1**-nodes, we will have only four points. If however, we took the minimal set of these four points, we will only have *one* node to keep, namely that for the argument $(\mathbf{0}, \mathbf{0}, \mathbf{1})$. By monotonicity all the nodes of the graph are actually represented by this one point. Thus the **1**-frontier for this function is the set $\{(\mathbf{0}, \mathbf{0}, \mathbf{1})\}$. Similarly we could have generated the **0**-frontier which is the set $\{(\mathbf{1}, \mathbf{1}, \mathbf{0})\}$.

An implementation of an analysis which uses the above frontier representation now

has to generate the frontier by applying the abstract function to particular argument points. If 'good' points are chosen then a lot of the argument space will be 'covered' by the frontier. As mentioned in [CP85], "It is this heuristic which offers the hope of almost always getting better than worst case performance". A typical (and naïve) algorithm for computing the frontier would involve a loop which chose a point, evaluates the abstract function at that point, and then modifies the frontier accordingly. Clack and Peyton Jones [CP85] have found that when generating a frontier, the frontier often 'flips' from being somewhere near the top(bottom) of the lattice to being somewhere near the bottom(top), and so suggest generating the **0**-frontier and the **1**-frontier in parallel to avoid more expense in finding it.

This elegant representation of the function graphs helps in two ways:

- Less resources are needed to keep the representations of successive function graphs for comparison purposes.
- The operation of comparing two graphs for equality now reduces to the comparing of two frontiers for equality. Since the number of points in a frontier will generally be less than that in the entire graph, this is an obvious improvement.

5.3 Reducing the size of a lattice

Hunt and Hankin [HH91] claim that higher order functions are often badly behaved (in the sense that the fixed points are not near the top or the bottom of the lattice) and that frontier analysis is not enough. They suggest that the problem can be solved by working in smaller lattices and establishing bounds on the fixed point.

The family of (finite) lattices, $\mathcal{L}$, used in strictness analysis (see chapter 2 and section 3.2.1) can be defined inductively by:

- $\mathbf{2} \in \mathcal{L}$
- $D_\perp \in \mathcal{L}$ if $D \in \mathcal{L}$
- $D_1 \times \ldots \times D_n \in \mathcal{L}$ if $D_i \in \mathcal{L}, 1 \leq i \leq n$
- $[D \rightarrow D'] \in \mathcal{L}$ if $D, D' \in \mathcal{L}$

To formalize the notion of one lattice being smaller than another, we define an *abstraction ordering*, $\preceq$, on members of $\mathcal{L}$:

$$\begin{aligned} \mathbf{2} &\preceq D \\ D_1 \times \ldots \times D_n &\preceq D'_1 \times \ldots \times D'_n \text{ if } D_i \preceq D'_i, 1 \leq i \leq n \\ D_\perp &\preceq D'_\perp \text{ if } D \preceq D' \\ [D \to E] &\preceq [D' \to E'] \text{ if } D \preceq D' \text{ and } E \preceq E' \end{aligned}$$

The objective is to define ways of relating the fixed points in the various lattices. To do this, we introduce 'abstraction' and 'concretisation' maps. For each $D, D' \in \mathcal{L}$ with $D' \preceq D$, the abstraction and concretisation maps are of the form:

$$Abs_{D,D'} \in [D \to D'] \text{ and } Conc_{D',D} \in [D' \to D]$$

To place upper and lower bounds on the fixed point of a lattice, we introduce two families of these maps, the *safe* and the *live* versions. The safe maps will give overestimates of values and thus allow us to derive upper bounds on fixed points, and the live maps will give underestimates and allow us to derive lower bounds on the fixed points.

As an example (see [HH91] for the complete definition), the live and safe maps of the *Abs* function are defined by:

$$Abs^s_{D,\mathbf{2}}\, x = \begin{cases} \mathbf{0} & \text{if } x = \perp_D \\ \mathbf{1} & \text{otherwise} \end{cases}$$

$$Abs^l_{D,\mathbf{2}}\, x = \begin{cases} \mathbf{1} & \text{if } x = \top_D \\ \mathbf{0} & \text{otherwise} \end{cases}$$

The central theorem then shows that for all lattices $D, D' \in \mathcal{L}$ such that $D' \preceq D$, and for all $f \in [D \to D']$

- $Conc^s_{D',D}(fix_{D'}(Abs^s_{[D\to D],[D'\to D']}\, f)) \sqsupseteq fix_D\, f$
- $Conc^l_{D',D}(fix_{D'}(Abs^l_{[D\to D],[D'\to D']}\, f)) \sqsubseteq fix_D\, f$

This tells us that we can find safe and live approximations to the value of a functions fixed point over some member of $\mathcal{L}$ by abstracting the functional to a smaller member

of $\mathcal{L}$ and finding the fixed point in this lattice and this is exactly what we need to provide some optimizations. Consider finding the fixed point of some function f with functional $F \in [A \to A]$. If we find that the lattice A is too large to work with, we can choose a smaller lattice (by our abstraction ordering) A'. By using the two formulae above, we can determine upper and lower bounds on the fixed point of A by working in the smaller lattice A'.

If the lower bound of the fixed point, fix_{lb} does not indicate that the function is strict in a way in which we are interested (that is, the lower bound gives us an overestimate of the function, and if this function is not strict then the actual function would not be strict anyway) then we need not do anymore computation. If however we find that fix_{lb} indicates that the function may be strict in ways that we are interested in, then we can compute fix_{ub}. If this result indicates that the function is strict, then we can retire as we have found the needed information. If it does not indicate that the function is strict, then we are still not sure (because we have an underestimate) and so we can choose yet another lattice A'' such that $A' \preceq A'' \preceq A$ and repeat this entire process again.

This entire operation has allowed us to calculate an approximation to a fixed point to any desired level of definedness. At any time we can stop and take what information we have, or keep choosing bigger lattices until we find what we want.

5.4 Related Work

Chronologically, the frontier analysis was developed by Clack and Peyton Jones [CP85]. Pending analysis was then developed by Young and Hudak [YH86]. Hunt and Hankin [Hun89, HH91] gave a more formal description to the frontier analysis in terms of minimal and maximal sets. The theory of reducing the sizes of lattices in which to find fixed points can also be found in [HH91]. Martin and Hankin [MH87, Mar89] extended the original frontiers method to allow representation of functions of the form $f : X \to \mathbf{2}$ for finite lattices X.

5.5 Conclusion

This chapter has addressed some issues relating to the implementation of the analyses presented in the previous chapters.

Exponential The most intensive operation found in the analyses was the finding of fixed points. Even taking into account that the domains involved are often small, the cost was seen to rise exponentially for each argument.

Pending Analysis The pending analysis was introduced as a way of optimizing the search for fixed points for first order languages if the equations could be expressed as recursive monotone boolean functions.

Frontier Analysis The frontier analysis is a technique which takes advantage of the monotonicity of the functions involved, allowing for a tight representation of function graphs by the maximal elements of a frontier. This provides a more efficient means for storing and comparing graphs of functions.

Lattice Sizes For certain functions it was found that the resulting domains are still too large for even the frontier analysis to be effective, and ways of reducing the size of the lattice and putting bounds on the fixed point were introduced.

Chapter 6

Review and Summary

In this chapter we will look at the work presented in this thesis by stepping back and building an overall picture of the analysis techniques, the role that they play in an implementation of a functional language, and various relationships between them.

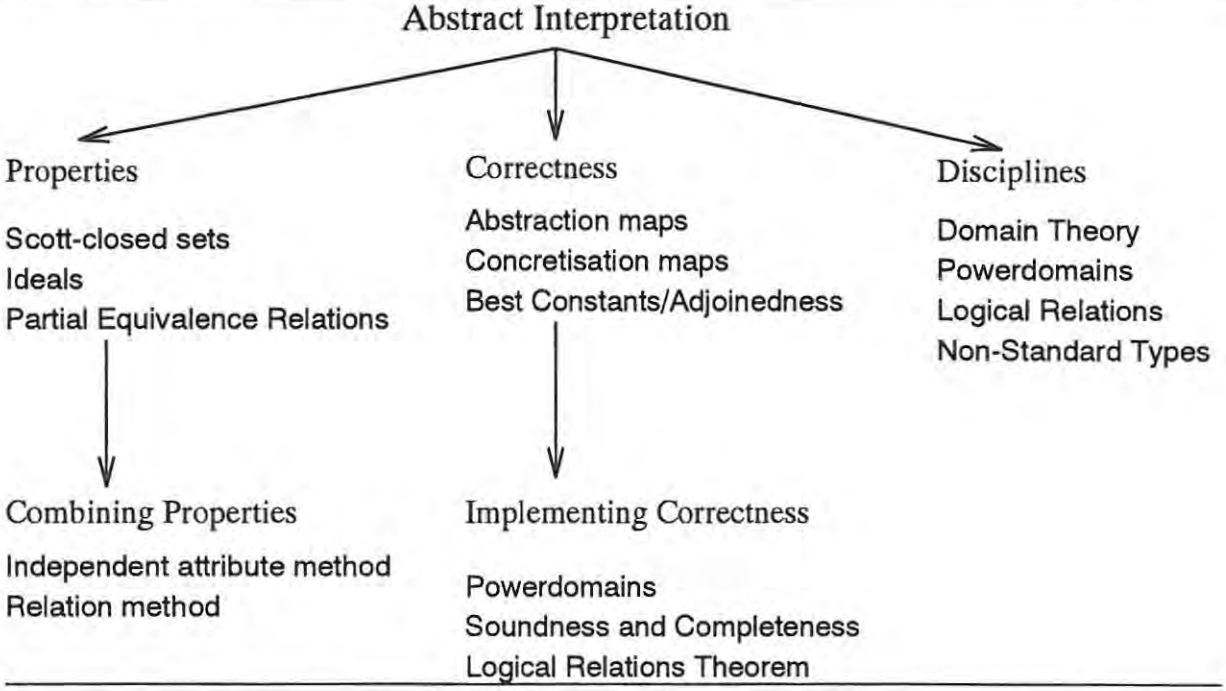
6.1 Review

- *Need for Analysis*: In the first chapter we examined the need for analysis techniques in the implementation of a functional language, arguing that it is the cost of abstraction that has led to the poor performance. Referential transparency was then introduced as a mechanism which allowed for abstraction and also a freedom of execution order, permitting lazy functional languages. All of the analyses presented in this thesis address the inefficiencies of these languages, and are often used to change the execution order of programs in a safe manner to permit the joint benefits of lazy languages and eager execution orders.

- *Abstract Interpretation*: In chapter 2 we found that the basic concepts behind the abstract interpretation framework were very simple, and based on the idea of giving a different interpretation to a language. An interpretation took the form of defining a set of domains to be used at the base types, and a set of constants to operate over these domains. Moreover, it was ensured that the domains were finite to ensure the effective computation of fixed points when finding the mean-

ing of recursive abstract functions. Abstraction and concretisation maps were introduced as a way of relating the standard and abstract interpretations, and it was shown that if these functions were adjoined then best interpretations would result, forcing a more accurate representation of information. The Correctness Theorem of abstract interpretation proved that the analysis for all expressions was correct if the analysis of the constants was correct, with a value computed by an abstract function being correct if it described all possible outputs of the concrete function. This analysis was also seen to be a forwards analysis as information was propagated from the leaves of an expression tree to the expression itself.

Figure 6.1 A View of Abstract Interpretation



- *Abstract Interpretation and other disciplines*: Chapter 3 looked at various extensions and reformulations of the abstract interpretation framework.

  A non-standard type system was then introduced, based on program logics. This particular type system could be proved correct with respect to the standard semantics of a language by proving that it is equivalent to an abstract interpretation utilizing Scott-closed sets.

In chapter 2, properties were equated with Scott-closed sets, an implication of which is that if an analysis is to detect some property of interest, that property has to be describable by Scott-closed sets. The abstract interpretation using partial equivalence relations is an attempt to get around this restriction by having the elements of the abstract domain represent partial equivalence relations over the standard domain as opposed to Scott-closed sets. Constancy was a property that was illustrated as being describable by pers and not by Scott-closed sets.

Finally, the abstract interpretation framework was shown to be equally well represented by using domain theory or relations. Figure 6.1 represents a summary of the important aspects behind the second and third chapter.

- *Projection Analysis*: This framework was based on the propagation of the information about an expression, the context of the expression, to the sub-expressions. Given a function, and a projection describing the context of the function, a projection analysis is able to determine a projection transformer that can propagate new projections into the sub-expressions.

  It was found that to capture strictness, the domain over which the projections operated had to be lifted. Smaller projections were seen to convey more information, and the projections formed a complete lattice, again allowing for the effective computation of fixed points. This analysis was seen to be an example of a backwards analysis in which information is propagated from the expression to the sub-expressions.

- *Implementing the Analyses*: In any implementation of the above analyses, the greatest costs are incurred when trying to find fixed points of recursive definitions. Pending analysis was introduced as a way of saving some computation in the first order case, and the frontier analysis which exploited the monotonicity of the functions involved, was shown to greatly reduce the number of points needed to represent a function, thus making the operation of comparing and storing graphs of a function much more efficient. The results on reducing the lattice size enable the successive approximating of the least fixed point, allowing the implementor to trade computation time against accuracy of results.

Figure 6.2 A View of Static Analysis

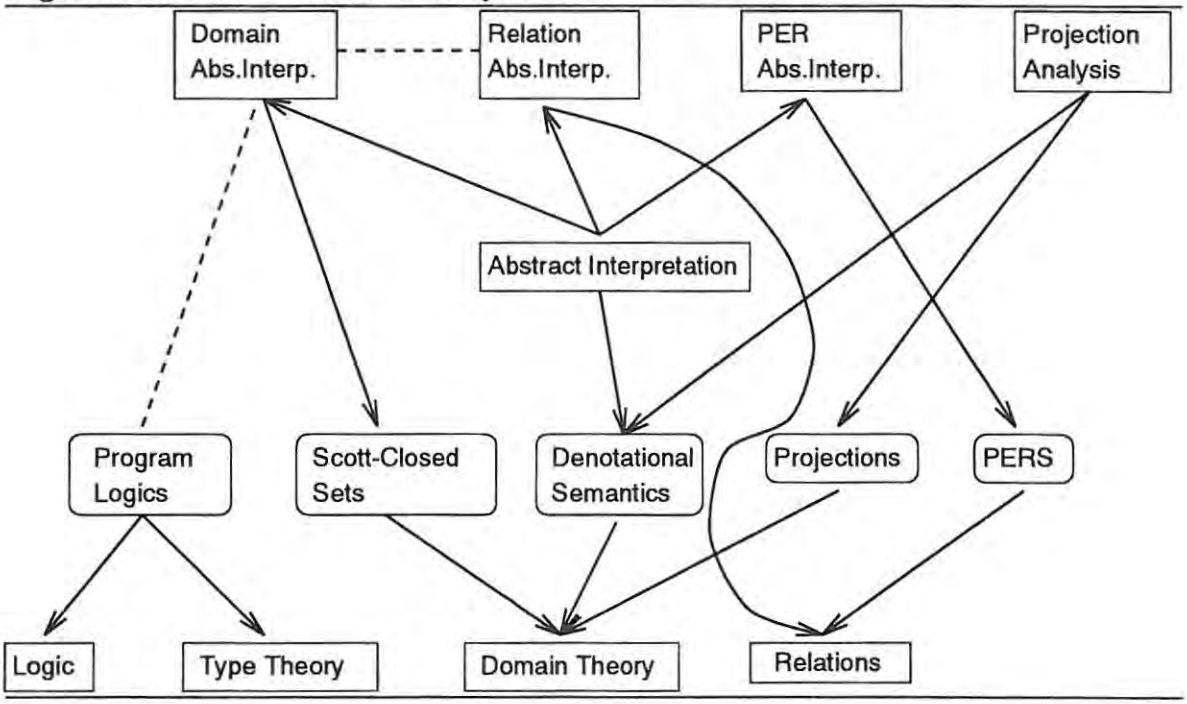


Figure 6.2 shows how the techniques, frameworks and mathematics presented in this thesis fit together. The dependencies are directional and indicated by a solid line while the dashed lines indicate that the techniques are equivalent.

6.2 Summation

One of the prime motivations for functional languages is their referential transparency and abstraction, and it has long been a goal of the functional language community to exploit the firm mathematical foundations of these languages. Much progress has been made, especially in the areas described in this work. But there is still some considerable gap between the theoretical techniques and their realization in current functional language compilers.

There also appears to be a substantial skills gap – the mathematics behind the techniques can be problematic for the compiler writer, and the theoreticians sometimes fail to appreciate implementation difficulties.

Bridging this gap will involve refining the formal techniques and an on-going process of organizing the material and simplifying its presentation to make it more accessible. This work contributes to that process.

References

[Abr85] S Abramsky. Strictness Analysis and Polymorphic Invariance. In H Ganzinger and N D Jones, editors, *Workshop on Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, 1985.

[Abr90] S Abramsky. Abstract Interpretation, Logical Relations and Kan Extensions. *Journal of Logic and Computation*, 1(1):5–39, 1990.

[Bac78] J Backus. Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, 1978.

[BHA85] G L Burn, C L Hankin, and S Abramsky. The Theory of Strictness Analysis for Higher Order Functions. In H Ganzinger and N D Jones, editors, *Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*, pages 42–62. Springer-Verlag, 1985.

[BHA86] G L Burn, C L Hankin, and S Abramsky. The Theory of Strictness Analysis for Higher Order Functions. In *Science of Computer Programming*, volume 7, pages 249–278. Elsevier, 1986.

[BL93] G L Burn and D Le Métayer. Proving the Correctness of Compiler Optimisations Based on Strictness Analysis. Technical Report DoC 93/42, Imperial College, 1993.

[BPR88] G L Burn, S L Peyton Jones, and J D Robson. The Spineless G-Machine. In *Proceedings of the 1988 ACM conference on Lisp and Functional Programming*, pages 244–258. ACM, July 1988.

[Bur87a] G L Burn. *Abstract Interpretation and the Parallel Evaluation of Functional Languages.* PhD thesis, Imperial College, University of London, 1987.

[Bur87b] G L Burn. Evaluation Tranformers - A Model for the Parallel Evaluation of Functional Languages (Extend Abstract). In G Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 446–470. Springer-Verlag, 1987.

[Bur90a] G L Burn. A Relationship Between Abstract Interpretation and Projection Analysis (Extended Abstract). In *17th POPL*, pages 151–156, 1990.

[Bur90b] G L Burn. Using Projection Analysis in Compiling Lazy Functional Programs. In *1990 ACM conference on Lisp and Functional Programming*, pages 227–241, 1990.

[Bur91] G L Burn. The Abstract Interpretation of Higher-Order Functional Languages: From Properties to Abstract Domains. In R Heldal, C Kestler Holst, and P Wadler, editors, *Proceedings of the 1991 Glasgow Functional Programming Workshop*, pages 56–72, 1991.

[Bur92] G L Burn. Properties of Program Analysis Techniques. Technical Report DoC92/19, Imperial College, 1992.

[CC77] P Cousot and R Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CC79] P Cousot and R Cousot. Systematic Design of Program Analysis Frameworks. In *Conference Record of the 6th ACM Symposium on Principles of Programming Languages*, pages 269–282, 1979.

[CC92] P Cousot and R Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4), 1992.

[CP85] C Clack and S L Peyton Jones. Strictness analysis - a practical approach. In *FPCA '85*, volume 201 of *Lecture Notes in Computer Science*, pages 35–39. Springer-Verlag, 1985.

[DW89] K Davis and P Wadler. Backward Strictness Analysis: Proved and Improved. In *Proceedings of the 1989 Glasgow Workshop on Functional Programming*, Springer Workshops in Computing. Springer-Verlag, 1989.

[DW90] K Davis and P Wadler. Strictness Analysis in 4D. In *Proceedings of the 1990 Glasgow Workshop on Functional Programming*, Springer Workshops in Computing. Springer-Verlag, 1990.

[Gol88] B F Goldberg. *Multiprocessor Execution of Functional Programs.* Phd dissertation, Yale University, April 1988.

[GS80] C A Gunter and D S Scott. Semantic Domains. In *Handbook of Theoretical Computer Science Vol B.* Elsevier, 1980.

[GS91] C K Gomard and P Sestoft. *Program Analysis Matters.* PhD thesis, DIKU, University of Copenhagen, 1991.

[HH91] S Hunt and C Hankin. Fixed points and frontiers: A new perspective. *Journal of Functional Programming*, 1(1):91–120, 1991.

[HKL91] G Hogen, A Kindler, and R Loogen. Automatic Parallelization of Lazy Functional Programs. Technical report, Technical University of Aachen, 1991.

[HL91] J Hughes and J Launchbury. Towards Relating Forwards and Backwards Analyses. In *Glasgow workshop on Functional Programming*, 1991.

[HL92a] J Hughes and J Launchbury. Relational Reversal of Abstract Interpretation. Technical report, Glasgow University, 1992.

[HL92b] J Hughes and J Launchbury. Reversing Abstract Interpretation. In B Krieg-Brückner, editor, *ESOP '92*, volume 582 of *Lecture Notes in Computer Science*, pages 269–286. Springer-Verlag, 1992.

[HS91] S Hunt and D Sands. Binding Time Analysis: A New PERspective. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1991.

[Hug85] J Hughes. Lazy Memo-functions. Technical Report PGM 42, Chalmers University, 1985.

[Hug87] J Hughes. Backwards Analysis of Functional Programs. Technical Report CSC/87/R3, University of Glasgow, 1987.

[Hug89a] J Hughes. Projections for Polymorphic Strictness Analysis. Technical Report CSC/89/R16, University of Glasgow, July 1989.

[Hug89b] J Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, 1989.

[Hug90] J Hughes. *Compile-time Analysis of Functional Programs*, chapter 5, pages 117–153. Research Topics in Functional Programming, Addison-Wesley, 1990.

[Hun89] S Hunt. Frontiers and open sets in abstract interpretation. In *Functional Programming Languages and Computer Architecture, Fourth International Conference*. ACM Press, September 1989.

[Hun91] S Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice*. PhD thesis, University of London, 1991.

[Jen91] T P Jensen. Strictness Analysis in Logical Form. In J Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[Jen92] T P Jensen. *Abstract Interpretation in Logical Form*. PhD thesis, University of London, 1992.

[JM81] N D Jones and S S Muchnick. Complexity of flow analysis, inductive assertion synthesis and a language due to Dijkstra. In S S Muchnick and N D Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[JN90] N D Jones and F Nielson. Abstract Interpretation: a Semantics-Based Tool for Program Analysis (DRAFT VERSION). Technical report, DIKU, University of Copenhagen, 1990.

[Kam92] S Kamin. Head-strictness is not a monotonic abstract property. *Information Processing Letters*, 41:195–198, 1992.

[KM87] T-M Kuo and P Mishra. On strictness and its analysis. In *Proceedings of the 14th ACM conference on principles of programming languages*, 1987.

[KM89] T-M Kuo and P Mishra. Strictness Analysis: A new perspective based on type inference. In *Proceedings of the 4th International Conference on Functional Programming and Computer Architecture*, 1989.

[Lan63] P J Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6:308–320, 1963.

[Mar89] C Martin. *Algorithms for Finding Fixedpoints in Abstract Interpretation.* PhD thesis, Imperial College of Science, Technology and Medicine, University of London, 1989.

[MH87] C Martin and C Hankin. Finding Fixed Points in Finite Lattices. In G Kahn, editor, *Functional Programmin Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science.* Springer-Verlag, 1987.

[MJ85] A Mycroft and N D Jones. A Relational Framework for Abstract Interpretation. In N D Jones, editor, *Programs as Data Objects*, number 215 in Lecture Notes in Computer Science, pages 156–171, 1985.

[Myc81] A Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, 1981.

[MZ92] R Muller and Y Zhou. Abstract Interpretation in Weak Powerdomains. In *1992 ACM conference on Lisp and Functional Programming.* ACM press, 1992.

[Nie89] F Nielson. Two-level Semantics and Abstract Interpretation. *Theoretical Computer Science*, 69:117–242, 1989.

[NN91] H R Nielson and F Nielson. Bounded Fixed Point Iteration. Technical Report DAIMI PB-359, Aarhus University, 1991.

[NN92] F Nielson and H R Nielson. The Tensor Product in Wadler's Analysis of Lists. In B Krieg-Brückner, editor, *ESOP '92*, volume 582 of *Lecture Notes in Computer Science*, pages 351–369. Springer-Verlag, 1992.

[Pey86] S L Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1986.

[Plo80] G Plotkin. Lambda-definability in the full type hierarchy. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 363–373. Academic Press, 1980.

[Sch86] D A Schmidt. *Denotational Semantics - A Methodology for Language Development.* Allyn and Bacon, INC., 1986.

[Sco76] D S Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3):522–587, 1976.

[Sco82] D S Scott. Domains for Denotational Semantics. In E M Schmidt M Nielson, editor, *Automata, Languages and Programming*, volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer-Verlag, 1982.

[Sew91] J R Seward. Towards a strictness analyser for Haskell: Putting theory to practice. Master's thesis, University of Manchester, 1991.

[Sto77] J E Stoy. *The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

[Tri90] P Trinder. *A Functional Database.* PhD thesis, University of Glasgow, 1990.

[Tur79] D A Turner. A New Implementation Technique for Applicative Languages. *Software-Practice and Experience*, 9:31–49, 1979.

[Veg84] S R Vegdahl. A Survey of Proposed Architectures for the Execution of Functional Languages. *IEEE Transactions on Computers*, c-33(12), December 1984.

[Wad87] P Wadler. Strictness analysis on non-flat domains (by Abstract Interpretation over infinite domains). In *Abstract Interpretation of Declarative Languages*, chapter 12, pages 266–275. Ellis Horwood, 1987.

[WH87] P Wadler and R J M Hughes. Projections for Strictness Analysis. In G Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 385–407. Springer-Verlag, September 1987.

[YH86] J Young and P Hudak. Finding Fixpoints on Function Spaces. Technical Report YALEU/DCS/RR-505, Yale Univeristy, 1986.