University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Abstract Interpretation of Functional Languages: From Theory to Practice

Sebastian Hunt

A thesis submitted for the degree of
Doctor of Philosophy of the University of London

October 1991

Abstract

Abstract interpretation is the name applied to a number of techniques for reasoning about programs by evaluating them over non-standard domains whose elements denote properties over the standard domains. This thesis is concerned with higher-order functional languages and abstract interpretations with a formal semantic basis.

It is known how abstract interpretation for the simply typed lambda calculus can be formalised by using binary logical relations. This has the advantage of making correctness and other semantic concerns straightforward to reason about. Its main disadvantage is that it enforces the identification of properties as *sets*. This thesis shows how the known formalism can be generalised by the use of ternary logical relations, and in particular how this allows abstract values to denote properties as *partial equivalence relations*. A framework based on this generalisation is developed, with the issues of induced interpretations for constants and the treatment of recursive types being considered in some detail.

Certain kinds of program properties can be captured by the use of *projections*, and analyses capturing these properties have previously been developed for first-order languages. It is shown how these same properties can be understood as partial equivalence relations and, using the above framework, how they can be captured by abstract interpretations for higher-order languages.

One of the most costly operations involved in automating analyses based on abstract interpretation is the computation of fixed points. For the case of first-order languages and interpretations based on the two-point lattice, there is an efficient algorithm for finding fixed points which uses the *frontier* representation for abstract functions. It is shown how frontiers may be understood as representations of upper-closed and lower-closed subsets of a function's domain and how a frontiers algorithm can be understood in these terms.

It is then shown how this view of frontiers may itself be seen as a special case of Birkhoff's Representation Theorem for finite distributive lattices. This allows frontiers to be applied in a far wider setting and a generalised frontiers algorithm is developed to take advantage of this.

Finally, it is observed that for many functions, especially in the higher-order case, finding fixed points in an abstract interpretation is an intractable problem because of the sizes of the abstract domains. A solution to this problem is developed which uses Galois connections to place upper and lower bounds on the values of fixed points in large lattices by working within smaller lattices.

Acknowledgments

I count myself lucky to have been supervised by Chris Hankin, whose help and encouragement have been unfailing. He has become a good friend.

Many thanks to my friends and office mates Dave Sands, Jesper Andersen and Thomas Jensen for being so generous with their time and patience. Many of their good ideas have certainly found their way into this thesis.

I am grateful to all those at Imperial who have helped to make my time here a happy and a fruitful one, both academically and not so academically. I would particularly like to thank Samson Abramsky, Geoff Burn, Mark Dawson, Abbas Edalat, Tony Field, Simon Hughes, Paul Kelly, Hessam Khoshnevisan, Mark Ryan and Lyndon While. Thanks to Paul Taylor for the use of his diagram macros.

Going further afield, I have benefitted greatly from discussions with Patrick Cousot, John Hughes, Neil Jones, John Launchbury and many others.

Finally, and most importantly, I want to thank Gill, for her love and friendship and for having such faith in me.

Sebastian Hunt
October 1991

Declaration

Some of the material presented in this thesis has appeared elsewhere in another form, and some is based on joint work with other authors.

The correspondence between projections and partial equivalence relations and the associated use of ternary logical relations described in Chapter 3, together with an early version of the head-strictness analysis described in Chapter 5, were first presented at the 1990 Glasgow Workshop on Functional Programming ([Hun90b, Hun90a]).

The constancy analysis described in Chapter 5 was developed jointly with David Sands and was presented at the Symposium on Partial Evaluation and Semantics-Based Program Manipulation 1991 ([HS91]).

Chapters 7 and 9 are based on work which was originally presented at the Fourth International Conference on Functional Programming and Computer Architecture 1989 ([Hun89]). This work was subsequently developed jointly with Chris Hankin ([HH91]), but the generalisation described in Chapter 8 is original to this thesis.

Contents

Abstract **2**

Acknowledgments **3**

Declaration **4**

1 Introduction **10**

1.1 The Theory of Abstract Interpretation 10
1.2 The Practice of Abstract Interpretation 12
1.3 Alternative Analysis Techniques 13
1.4 Overview of Thesis 14
1.5 Notation and Terminology 16

2 Abstract Interpretation Using Sets **20**

2.1 A Simply Typed Lambda Calculus 20
2.2 Interpretations 22
2.2.1 The Standard Interpretation 23
2.3 Strictness and Scott-Closed Sets 23
2.4 Abstract Interpretation 27
2.4.1 A Finite Interpretation for Strictness Analysis 27
2.4.2 Relations 29
2.4.3 Concretisation Maps and Logical Relations 30
2.4.4 Best Interpretations for Constants 33
2.5 Polymorphism 34

3 Pers and Ternary Logical Relations **37**

3.1 BTA, Projections and Equivalence Relations 38
3.1.1 Projections 39
3.1.2 Equivalence Relations 41

3.2 Complete Partial Equivalence Relations 43
3.2.1 Complete Pers 45
3.3 Ternary Logical Relations 47
3.4 Logical Concretisation Maps 49
3.4.1 Co-Step Functions as Basic Properties 50
3.5 Inherited Properties 52

4 Abstract Interpretation Using Pers 55
4.1 Correctness 55
4.1.1 Least Fixed Point Interpretations 58
4.1.2 Monotone Concretisation Maps 59
4.2 Best Interpretations for Constants 60
4.2.1 Left Adjoints for Concretisation Maps 60
4.2.2 Abstraction Maps 62
4.3 Non-Injective Conc Maps and Expected Forms 64
4.4 Pers Subsume Sets 67

5 Example Analyses 69
5.1 A Constancy Analysis 69
5.1.1 The Abstract Domains and Concretisation Maps 70
5.1.2 The Interpretations of Constants 71
5.1.3 Non-Injective Concretisation Maps 74
5.1.4 Constancy in a Strict Language 76
5.2 List Types 77
5.2.1 Interpretations 78
5.2.2 Logical Relations 78
5.2.3 Inherited Properties 79
5.2.4 List Constants and the Standard Interpretation 79
5.3 A Head-Strictness Analysis 80
5.3.1 The Abstract Domains and Concretisation Maps 82
5.3.2 Testing For Strictness and Head-Strictness 83
5.3.3 The Interpretations of Constants 84
5.3.4 An Example Analysis for Head Strictness 88

6 Recursive Types 90
6.1 Enriching the Languages of Types and Terms 90
6.2 The Interpretation of Unit and Sum Types 92

6.3 Domains and **O**-Categories . . . 93
6.3.1 Functors on $\mathbf{Dom}_\perp$ and $\mathbf{Dom}^{ep}$ . . . 94
6.4 A Category of Complete Pers . . . 95
6.4.1 Functors on **CPD** and $\mathbf{CPD}^{ep}$ . . . 96
6.5 Solving Recursive Equations in $\mathbf{CPD}^{ep}$ . . . 98
6.6 The Standard Interpretation of Terms . . . 100
6.7 The Abstract Interpretation of Terms . . . 101
6.7.1 An Unsolved Monotonicity Problem . . . 102
6.7.2 Abstract Interpretation for The Restricted Case . . . 104
6.7.3 Correctness . . . 106
6.7.4 Defining fold and unfold . . . 107
6.7.5 Meet Preservation Is Not Inherited . . . 109
6.7.6 An Example . . . 110
6.8 Strictness Analysis . . . 112

7 The Frontiers Algorithm **114**
7.1 Exploiting Monotonicity . . . 115
7.2 A Basic Frontiers Representation . . . 116
7.3 The Basic Frontiers Algorithm . . . 118
7.4 The Algorithm as a Search . . . 122

8 The Generalised Frontiers Algorithm **124**
8.1 The Theory of Finite Distributive Lattices . . . 124
8.2 A Generalised Frontiers Representation . . . 128
8.3 The Generalised Frontiers Algorithm . . . 130
8.3.1 Implementing the Tests . . . 132
8.4 The Operations of the Algorithm . . . 133
8.4.1 Operations on $\mathcal{J}(A)$ and $\mathcal{M}(A)$ . . . 135
8.4.2 Operations on A . . . 140

9 Approximate Fixed Points **143**
9.1 A problem of complexity . . . 143
9.1.1 Reducing the Size of a Lattice . . . 145
9.1.2 Applying *Conc* to a Frontier . . . 148
9.1.3 Using the Upper and Lower Bounds . . . 150

10 Conclusions **151**
10.1 Summary . . . 151

10.2 Suggestions for Further Work 153

Appendix: Proofs from Chapters 7 and 8 **156**

A.1 Proofs from Chapter 7 156

A.2 Proofs from Chapter 8 158

Bibliography **161**

List of Figures

1.1 Two Approaches to Correctness. 11

2.1 Typing Axiom and Rule Schemata for $\Lambda_{\mathcal{T}}$ 21
2.2 The Semantic Valuation Function Induced by I 23
2.3 The Standard Interpretation 24
2.4 A Finite Interpretation for Strictness Analysis 29

5.1 An Abstract Interpretation for Constancy Analysis 71
5.2 The Lattice $[\mathbf{2} \times \mathbf{2} \to \mathbf{2}]$ 74
5.3 Extensions to the Standard Interpretation 80
5.4 Interpretations Of Constants for Head-Strictness Analysis 85

6.1 The Language of Terms $\Lambda_{\mathcal{T}^\mu}$ 91
6.2 The Additional Typing Rules for $\Lambda_{\mathcal{T}^\mu}$ 91
6.3 The Standard Interpretation of Terms in $\Lambda_{\mathcal{T}^\mu}$ 101
6.4 The Abstract Interpretation of Terms in $\Lambda_{\mathcal{T}^{\overline{\alpha}}}$ 105

7.1 Algorithm A: an algorithm to find $f^{-1}\{0\}$ and $f^{-1}\{1\}$ 118
7.2 Algorithm B: a naïve algorithm to find $\mathbf{F\text{-}0}(f)$ and $\mathbf{F\text{-}1}(f)$ 119
7.3 The Basic Frontiers Algorithm 122
7.4 The Search Space Before and After Update 123

8.1 The Generalised Frontiers Algorithm 131
8.2 A Family of Finite Distributive Lattices 133
8.3 Join and Meet Irreducible Elements for $A \in \mathcal{A}$ 134
8.4 Minimal Upper Bounds and Maximal Lower Bounds in $\mathcal{J}(A)$ 136
8.5 Components of the Isomorphism $\mathcal{J}(A) \cong \mathcal{M}(A)$ 137
8.6 Upper Complements in $\mathcal{M}(A)$ 139

Chapter 1

Introduction

Functional languages have many advantages over more traditional imperative languages: run-time efficiency is not one of them. By abstracting away from 'low-level' details such as the management of store, functional languages focus attention on *what* is to be computed rather than *how* it is to be done. For each input A, a functional program specifies an output B, but much of the burden of working out the best route from A to B falls on the compiler. Thus for functional languages the provision of *highly optimising* compilers is of crucial importance. This thesis is concerned with techniques for constructing and implementing provably correct automated program analyses for use within optimising compilers for functional languages.

1.1 The Theory of Abstract Interpretation

Abstract interpretation is a mathematical framework in which program analyses can be formalised and proved correct. It is based on the idea of evaluating a program over a non-standard domain, whose elements denote *properties* over the domain of the standard interpretation. Of course, the intention is that an abstract interpretation should be *correct*: the property denoted by the abstract interpretation of a program should be one which is satisfied by the standard interpretation of that program. It is also intended that the abstract interpretation can be computed in finite time. For most properties of interest, this requirement implies that in general the property computed will only be approximate: an abstract interpretation may be able to determine that for all inputs a program computes a non-negative integer, whereas in fact the result is always in the range 0–255.

The theory of abstract interpretation was developed for imperative flow-chart languages by Patrick and Radhia Cousot ([CC77]). It was later adapted and ap-

$$
\begin{tikzcd}
 & \llbracket e \rrbracket \arrow[dl, "S"'] \arrow[dr, "J"] & \\
s : P & & a \arrow[ll, "\gamma"']
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \llbracket e \rrbracket \arrow[dl, "S"'] \arrow[dr, "J"] & \\
s \arrow[rr, leftrightarrow, "R"] & & a
\end{tikzcd}
$$

(i) *(ii)*

Figure 1.1: Two Approaches to Correctness.

plied to first-order functional languages by Alan Mycroft ([Myc81]) to formalise *strictness analysis*: an analysis for lazy functional languages to detect function parameters which are always evaluated. Mycroft's work was in turn extended to higher-order functional languages by Geoffrey Burn, Chris Hankin and Samson Abramsky ([BHA86]). Despite the common underlying idea, the theories of abstract interpretation for imperative and functional languages tend to be rather different in character. In particular, the original work of the Cousots was based on an essentially *operational* semantic framework, whereas Mycroft's work and that which has followed it is based on the use of *denotational* semantics, in which terms are mapped to values in some (usually structured) domain. Flemming Nielson ([Nie84]) developed a very general framework for abstract interpretation in a denotational setting, although it was not immediately applicable to higher-order functional languages. In this thesis we follow the [Myc81, BHA86] line of development in using denotational semantics.

The key issue in abstract interpretation is that of correctness. Figure 1.1 shows two schemes for formalising the notion of correctness. In both diagrams the map labeled S takes the term e to its standard denotation s, and the map labeled J takes e to its abstract interpretation a. In *(i)* correctness is captured by the requirement that s *satisfies* P, written $s : P$, where P is the property denoted by the abstract interpretation a. The map γ taking a to the property it denotes, is known as a *concretisation* map. In *(ii)* correctness is captured by the requirement that s and a be *related* by some correctness relation R.

The concretisation map and correctness relation approaches to correctness are really two sides of the same coin. In the relational framework due to Samson Abramsky ([Abr90]) the equivalence of the two approaches takes the form of identifying $\gamma(a)$ with the set $\{d \mid d \; R \; a\}$. Thus in this case satisfaction is just set membership: $s : P \iff s \in P$. We will see that in some respects this view of correctness is overly restrictive. By using ternary instead of binary correctness relations, and by using partial equivalence relations instead of sets as properties, we can significantly extend the applicability of abstract interpretation.

The diagrams of Figure 1.1 hide all the internal structure of an abstract interpretation and the fact that the real challenge is to construct a framework in which correctness can be reasoned about in a *compositional* way. The Cousots had already done this for the imperative case but functional languages, and particularly higher-order languages, posed new problems. The approach of [Myc81, BHA86, Nie84] was to concentrate on the concretisation maps (and the associated *abstraction* maps, see Chapter 4) and this entailed the use of various power domains. By contrast, [MJ85][1] concentrated on the correctness relation itself, the key insight being the relevance of *logical relations* ([Plo73]) to the theory of abstract interpretation of higher-order functional languages. In [Abr90] this idea was refined and applied to the simply typed rather than the untyped lambda calculus, and the connection with the earlier power domain approaches was made clear. We take [Abr90] as our starting point.

1.2 The Practice of Abstract Interpretation

To be of any use, the abstract interpretation of a term must be computable in finite time. In the abstract interpretation of functional languages this is usually ensured by using *finite lattices* as abstract domains. However, just because the abstract interpretation of a term is computable in theory, doesn't mean we automatically have an efficient method for computing it. The real problems in this regard are caused by recursively defined functions. For reasons which are outlined in Chapter 7, in the implementation of an abstract interpretation, unlike the implementation of the standard interpretation, it is necessary to explicitly compute the graph of a recursively defined function. This involves an iterative process, generating a sequence of approximations to the least fixed point of a function and checking for convergence on each iteration. The use of finite lattices means that the function graphs are themselves finite and that the fixed point iteration always terminates, but the cost of computing the graph of a function can be very high, both in space and time.

Economical representations for the graphs of the abstract functions occurring in first-order strictness analysis, known as *frontiers*, and an algorithm for constructing them, were developed by Chris Clack and Simon Peyton Jones ([CJ85, JC87]). First-order strictness analysis uses the lattice **2**, which has two elements 0 and 1 with

[1] A similar idea is already present in [Nie84].

$0 \leq 1$, and the functions used in abstract interpretation are *monotone*. The naïve representation of a function's graph is just the look-up table $\{(a, b) \mid f(a) = b\}$, but if f is a monotone function into $\mathbf{2}$ and $a \leq a'$, then once it has been established that $f(a) = 1$, it is known without any more calculation that $f(a') = 1$ as well. The frontiers technique exploits this structure to reduce the space taken up in representing a function's graph, and to reduce the amount of effort involved in computing it.

This original work was subsequently extended to functions over a larger class of lattices, together with methods for coping with higher-order functions, by Chris Martin and Chris Hankin ([MH87, Mar89]). The original formulations of the frontiers algorithm, and those of its subsequent extensions, were rather complex. We are able to offer a much clearer explanation of the algorithm by exposing connections with the theory of finite lattices. Furthermore, once these connections have been exposed it is possible to generalise the use of frontiers in a very strong way, by exploiting Birkhoff's Representation Theorem for finite distributive lattices.

Even with clever representations such as frontiers, some of the lattices which occur in the abstract interpretation of higher-order functions are so large that the problem of establishing the graph of a function becomes intractable. In the setting of imperative languages, the Cousot's characterised a class of approximation techniques known as *widening* and *narrowing* to cope with similar problems (in fact, in the Cousots' work approximation is unavoidable since the lattices are not required to be finite). We describe an example of such an approximation technique, appropriate for functional languages, which uses Galois connections to move between larger and smaller lattices, allowing us to establish upper and lower bounds on the least fixed points of functions.

1.3 Alternative Analysis Techniques

There are other ways of formalising program analyses, and there are analyses which do not appear to fall within the remit of the abstract interpretation frameworks we have mentioned. Particularly interesting in the setting of functional languages, are those analyses which have been formalised using *projections* (a class of domain retractions). Projections were first used for this purpose by Phil Wadler and John Hughes ([WH87]), to formalise a kind of strictness analysis. One of the striking features of this work was that a new strictness property known as *head-strictness* was identified, which the abstract interpretation technique of [BHA86] and [Abr90] could not capture. Projections were subsequently used by John Launchbury ([Lau89]) to formalise an analysis known as binding time analysis. Again, abstract interpretation in the style of [BHA86, Abr90] was unable to capture the relevant property. However, a drawback of these projection based analysis techniques is that they are restricted to first-order languages. We will show how the elusive properties *can* be captured within abstract interpretations for higher-order languages, by using an abstract interpretation framework based on partial equivalence relations.

1.4 Overview of Thesis

The current chapter concludes with a section introducing our basic notation and terminology. The rest of the thesis divides roughly into two parts:

- Chapters 2 to 6 deal with the underlying mathematical framework which formalises the definition of abstract interpretation and in which analyses can be specified and proved correct;

- Chapters 7 to 9 deal with the practice of abstract interpretation, and in particular with methods of constructing representations for the abstract interpretations of recursive definitions.

The reader is warned that our treatment of the practice of abstract interpretation is itself quite theoretical. However, we have endeavored to take the work to the point where the step to an actual implementation is a small one, and in fact the work of Chapters 7 and 9 has been successfully incorporated in the implementation of a strictness analyser for a higher-order functional language.

In **Chapter 2** we introduce a simply typed lambda calculus with constants. The notion of *interpretation*, an assignment of domains to types and values to constants, is described and the standard interpretation is specified. We give the definition of a *binary logical relation* (a type-indexed family of relations) between interpretations. Using strictness analysis as an example, we summarise [Abr90]'s use of the Binary Logical Relations Theorem in formalising correctness for abstract interpretation. We introduce the idea of presenting a logical relation as a type-indexed family of *concretisation maps*, where a concretisation map takes an abstract value to the set of values from the standard interpretation which are related to it.

In **Chapter 3** we highlight the limitation of the binary logical relations framework, using *constancy* as an example of a property which cannot naturally be captured using sets. We review Launchbury's use of *projections* to capture this property, and go on to show how equivalence relations can be used as an alternative to projections. We then introduce *partial equivalence relations* (pers), which generalise equivalence relations, and the lattice of *complete* pers, which are the appropriate pers for domains. We describe *ternary* logical relations and the Ternary Logical Relations Theorem. We show how a ternary logical relation can also be represented as a family of concretisation maps, but in this case a concretisation map takes each abstract value to a *relation* instead of a set. We show how the property of a family of relations being logical can be characterised as a property of its associated family of concretisation maps: we call a family of maps with this property a *logical concretisation map*. We show that if each base-type member of a logical concretisation map is a well defined map into the lattice of complete pers, then so is every higher-type member. This introduces the idea of *inheritance* for logical relations/logical concretisation maps and we describe some key examples of inherited properties.

In **Chapter 4** we use the correspondence between pers and ternary logical relations to develop a framework for abstract interpretation. We formalise the notion of correctness in this setting and show that the correctness of an interpretation is implied by the correctness of the interpretations of the individual constants. We adapt [Abr90]'s result concerning least fixed point interpretations. We go on to introduce the idea of *best interpretations* for constants and show their existence to be guaranteed if the base-type concretisation maps preserve meets. This leads to the definition of *abstraction maps*, which give the best interpretation for a constant as a function of its standard interpretation. We discuss the way in which non-injective concretisation maps can interfere with the derivation of best interpretations.

In **Chapter 5** we present two example abstract interpretations. The first is designed to capture the property of constancy introduced in Chapter 3. We show the concretisation maps for this interpretation to be non-injective. The second example interpretation is designed to capture the *head-strictness* property of [WH87]. To do this we have to extend our language of types with list types. We show how the framework developed in Chapters 3 and 4 can be adapted to allow this.

In **Chapter 6** we consider the implications of extending our language with *recursive types*. We introduce a category of complete pers and show how the framework of [SP82] can be used to give meaning to recursive descriptions of pers on the standard domain interpretation of a recursive type. We go on to consider the extension of the abstract interpretation framework to the new language. We find that in spite of being able to give meaning to recursively described pers, we are only able to induce finite lattices for abstract interpretations if we restrict the use of $\rightarrow$ in recursive types. Under this restriction we describe an extended framework and illustrate its use by applying it to the constancy analysis of Chapter 5. We show that the extended framework does not generalise strictness analyses such as the head-strictness analysis of Chapter 5.

In **Chapter 7** we explain the need to compute the complete graph of recursively defined functions when implementing abstract interpretations. We introduce [JC87]'s *frontier* representations, which apply to a restricted form of monotone function. We observe that such frontiers correspond to lower and upper subsets of a function's argument domain, and based on this observation we derive an algorithm for computing the frontier representations of a function.

In **Chapter 8** we observe that the finite lattices used in abstract interpretation are typically *distributive*. We show how this observation leads to a generalised definition of frontiers as representations of upper and lower sets of *irreducible* elements. We develop an algorithm for constructing these generalised frontier representations. We then introduce a family of finite distributive lattices, suitable for use in abstract interpretation, and give the details of how the various operations required by the generalised frontiers algorithm can be implemented for lattices in this family.

In **Chapter 9** we show that the abstract lattice interpretations for types which occur in quite simple programs can be so large that computing the whole graph of a function on such lattices is intractable. We describe a way of coping with this problem by using a certain family of Galois connections to approximate values in large lattices by values in smaller ones. The key result is that the fixed points of the approximations of a function in a small lattice, can be used to place upper and lower bounds on the fixed point of the original function in the larger lattice. We consider briefly the interaction of this approximation technique with the use of frontiers.

In **Chapter 10** we review the main contributions of the thesis and suggest some directions for future work.

1.5 Notation and Terminology

In this section we review some basic lattice and domain theoretic notation and terminology. An excellent introduction to lattice theory is [DP90].

Posets

A *poset* is a pair $(P, \leq)$ where P is a set, known as the *carrier* of the poset, and $\leq$ is a partial order (a reflexive, transitive and anti-symmetric relation) on P. We will follow usual practice in writing P both for the carrier and the poset, allowing context to determine which is intended. Two posets P and Q are said to be *order isomorphic* if there is a one-one and onto map $f : P \to Q$, such that for all $x, y \in P$

$$f(x) \leq f(y) \iff x \leq y.$$

The one-element poset $\{\cdot\}$ is written $\mathbf{1}$.

Chains

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

Lattices

Given $x, y \in P$, an element $z \in P$ such that $x \leq z$ and $y \leq z$ is called an *upper bound* for x and y. As its name suggests, a *least upper bound* (also known as a *join*) for x and y is an upper bound z such that if z' is any other upper bound for x and y, then $z \leq z'$. The notions of *lower bounds* and *greatest lower bounds* (also known as *meets*) are defined analogously. If the join (meet) of x and y exists, it is necessarily unique and is written $x \vee y$ ($x \wedge y$).

A *join semi-lattice* (*meet semi-lattice*) is a poset in which every pair of elements has a join (meet). A *lattice* is a poset in which every pair of elements has both a join and a meet. The two-point lattice $\mathbf{2}$, has carrier $\{0, 1\}$ ordered by $0 \leq 1$.

The definitions of least upper bound and greatest lower bound can clearly be generalised to apply to arbitrary subsets of elements, not just pairs. A *complete* lattice has least upper bounds and greatest lower bounds of all subsets X, written $\bigvee X$ and $\bigwedge X$ respectively. Any finite lattice is automatically complete.

Duality

For any poset $P = (P, \leq)$, the *opposite* of P, written P^{OP}, is the poset $(P, \geq)$. The notion of opposite gives rise to the notion of *duality*: for example the concepts of join and meet are dual because $x \vee y$ in P is the same as $x \wedge y$ in P^{OP}. Hence the opposite of a join semi-lattice is a meet-semi lattice and vice versa. Note that lattices are *self dual* in that L is a lattice if and only if L^{OP} is a lattice. The importance of this concept of duality is that it allows us to reduce repetition of definitions and proofs which are essentially the same apart from the 'orientation' of the partial orders involved. We will see numerous examples in the rest of this section and in the latter part of this thesis.

Upper and Lower Sets

Let P be a poset. An *upper closed* (or just *upper*) subset $X \subseteq P$ is one for which

$$x \in P \text{ and } x \leq x' \Rightarrow x' \in P.$$

The *lower* sets are defined dually. A subset $X \subseteq P$ is upper if and only if its complement $P \setminus X$ is lower. Hence X is lower if and only if $P \setminus X$ is upper. The collection of all lower subsets of P forms a complete lattice $\mathcal{L}(P)$ when ordered by subset inclusion. The collection of all upper subsets of P also forms a complete lattice $\mathcal{U}(P)$, but we order this by *reverse* inclusion: for $Y, Y' \in \mathcal{U}(P)$, $Y \leq Y' \iff Y' \subseteq Y$. Defined in this way the lattices $\mathcal{U}(P)$ and $\mathcal{L}(P)$ are order isomorphic, with the isomorphism being given by $X \mapsto P \setminus X$ in both directions.

The lattices $\mathcal{L}(X)$ and $\mathcal{U}(X)$ are both closed under intersection and union: in $\mathcal{L}(X)$ intersection gives meets and union gives joins, while in $\mathcal{U}(X)$ union gives meets and intersection gives joins. The upward closure of a subset $X \subseteq P$ is the upper set

$$\uparrow X = \{x' \in P \mid \exists x \in X.\, x \sqsubseteq x'\}.$$

The lower closure $\downarrow X$ is defined dually. It is easy to see that a set X is upper if and only if $X = \uparrow X$, and that X is lower if and only if $X = \downarrow X$.

Monotone Functions

Let A and B be posets. A map $f : A \to B$ is *monotone* if $a \leq a' \Rightarrow f(a) \leq f(a')$. The *pointwise* ordering on monotone maps is a partial order defined by $f \leq g \iff \forall a \in A.\, f(a) \leq g(a)$. The poset of all monotone maps under the pointwise ordering is written $[A \to_m B]$. If B is a lattice then so is $[A \to_m B]$. The posets $[A \to_m B]$ and $[A^{\mathrm{OP}} \to_m B^{\mathrm{OP}}]$ are dual, that is to say:

$$[A^{\mathrm{OP}} \to_m B^{\mathrm{OP}}] = [A \to_m B]^{\mathrm{OP}}.$$

Products and Sums

Let A_1 and A_2 be posets. The *product* of A_1 and A_2 is the cartesian product $A_1 \times A_2$ ordered by $(a_1, a_2) \leq (a_1', a_2') \iff a_1 \leq a_1' \text{ and } a_2 \leq a_2'$. If A_1 and A_2 are both lattices then so is $A_1 \times A_2$. The *separated sum* of A_1 and A_2, written $A_1 + A_2$, has as carrier the set $\{\bot\} \cup \{in_1(a_1) \mid a_1 \in A_1\} \cup \{in_2(a_2) \mid a_2 \in A_2\}$, and is ordered by $\bot \leq x$ for all $x \in A_1 + A_2$ and $in_i(a) \leq in_j(a') \iff i = j \textit{ and } a \leq a'$. Both products and separated sums can be generalised to the n-ary case.

Lifting and Topping

The *lifting* of a poset A, written $A_\bot$, has as carrier the set $\{\bot\} \cup \{\mathit{lift}(a) \mid a \in A\}$, and is ordered by $\bot \leq x$ for all $x \in A_\bot$ and $\mathit{lift}(a) \leq \mathit{lift}(a') \iff a \leq a'$. The *topping* of A, written $A^\top$, has as carrier $\{\top\} \cup \{\mathit{colift}(a) \mid a \in A\}$, and is ordered by $x \leq \top$ for all $x \in A^\top$ and $\mathit{colift}(a) \leq \mathit{colift}(a') \iff a \leq a'$.

Domains

We assume some familiarity with domain theory. The partial order on a domain is usually written $\sqsubseteq$ and joins and meets are usually written $x \sqcup y$ and $x \sqcap y$. In this thesis we take a *domain* to be a Scott domain, i.e., a bounded-complete ω-algebraic cpo with least element (a cpo is a poset in which every ω-chain has a least upper bound). The reader unfamiliar with these notions is referred to [GS90]. Any finite lattice is a domain. If D and E are domains then so are $D \times E$, $D + E$ and $D_\bot$.

Continuous Functions

Let D and E be domains. A map $f : D \to E$ 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, which we write $[D \to E]$. Note that for finite domains, $[D \to E]$ and $[D \to_m E]$ are the same thing.

Category Theory

In Chapter 6 we assume some familiarity with category theory. Although the category theory used is not very advanced, we do not attempt to make Chapter 6 self-contained. A very good introduction to category theory for computing science is [BW90].

Chapter 2

Abstract Interpretation Using Sets

In this chapter we introduce a framework for the abstract interpretation of higher-order functional languages. The key limitation of this framework is that it enforces an understanding of properties as *sets*. In subsequent chapters we show how the framework can be extended to overcome this limitation.

We begin by describing our functional language, a simply typed lambda calculus with constants.

2.1 A Simply Typed Lambda Calculus

We assume a finite set of base types

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

which includes *bool* and *int*. Our language of types is then:

$$\sigma, \tau \in \mathcal{T} \ ::= \iota \mid \sigma_1 \times \sigma_2 \mid \sigma_1 \to \sigma_2.$$

At each type we assume the sets Var_σ and Con_σ, from which are drawn variables and constants:

$$\begin{aligned} x, y, \ldots \in Var &= \textstyle\bigcup_{\sigma \in \mathcal{T}} Var_\sigma \\ c \in Con &= \textstyle\bigcup_{\sigma \in \mathcal{T}} Con_\sigma \end{aligned}$$

These sets are subject to the following conditions:

1. if σ and τ are distinct types then Var_σ and Var_τ are disjoint;

$$x : \sigma \quad \text{if } x \in \mathit{Var}_\sigma$$

$$c : \sigma \quad \text{if } c \in \mathit{Con}_\sigma$$

$$\frac{e : \tau}{\lambda\, x \,.\, e : \sigma \to \tau} \ \text{if } x \in \mathit{Var}_\sigma \qquad\qquad \frac{e_1 : \sigma \to \tau \qquad e_2 : \sigma}{e_1\ e_2 : \tau}$$

$$\frac{e_1 : \sigma_1 \qquad e_2 : \sigma_2}{(e_1, e_2) : \sigma_1 \times \sigma_2} \qquad \frac{e : \sigma_1 \times \sigma_2}{\mathbf{fst}(e) : \sigma_1} \qquad \frac{e : \sigma_1 \times \sigma_2}{\mathbf{snd}(e) : \sigma_2}$$

Figure 2.1: Typing Axiom and Rule Schemata for $\Lambda_{\mathcal{T}}$

2. if σ and τ are distinct types then Con_σ and Con_τ are disjoint;

3. *Var* and *Con* are disjoint.

Remark Because of these disjointness conditions we may think of each variable and constant as being decorated with its type, so the type system described below is essentially a Church (explicit) system rather than a Curry (implicit) system ([Bar91b]). We have avoided explicit type decorations to reduce notational clutter.

The syntax of terms is then given by:

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

Terms are assumed well-formed according to the typing axiom and rule schemata shown in Figure 2.1. Note that the disjointness conditions on *Var* and *Con* ensure that each well-formed term has exactly one type.

The constants are assumed to include the following:

- $\mathbf{n} : \mathit{int},\ n \in \omega$;
- $\mathbf{plus}, \mathbf{minus}, \mathbf{mult} : \mathit{int} \to \mathit{int} \to \mathit{int}$;
- $\mathbf{true}, \mathbf{false} : \mathit{bool}$;
- $\mathbf{iszero} : \mathit{int} \to \mathit{bool}$;
- $\mathbf{if}_\sigma : \mathit{bool} \to \sigma \to \sigma \to \sigma$, for each $\sigma \in \mathcal{T}$;
- $\mathbf{Y}_\sigma : (\sigma \to \sigma) \to \sigma$, for each $\sigma \in \mathcal{T}$.

2.2 Interpretations

An *interpretation* I, is a pair

$$(\left\{D^I_\iota\right\}_{\iota\in\mathcal{T}_0}, \left\{K^I_\sigma\right\}_{\sigma\in\mathcal{T}}),$$

where:

1. Each D^I_ι is a domain : the interpretation of base type ι. The interpretation for base types is extended to an interpretation for all types $\left\{D^I_\sigma\right\}_{\sigma\in\mathcal{T}}$ as follows:

$$\begin{aligned} D^I_{\sigma_1\times\sigma_2} &= D^I_{\sigma_1}\times D^I_{\sigma_2} \\ D^I_{\sigma\to\tau} &= [D^I_\sigma \to D^I_\tau]. \end{aligned}$$

2. Each K^I_σ is a map $Con_\sigma \to D^I_\sigma$: the interpretation of constants of type σ.

The least element of D^I_σ is written $\perp^I_\sigma$ and, when it exists, the greatest element is written $\top^I_\sigma$. For $c \in Con_\sigma$, we will write c^I to mean $K^I_\sigma[\![c]\!]$.

An I-*environment* is a partial map ρ from Var to $\bigcup_{\sigma\in\mathcal{T}} D^I_\sigma$ with finite domain, denoted $dom(\rho)$, and such that if $x \in Var_\sigma$ and $x \in dom(\rho)$, then $\rho(x) \in D^I_\sigma$. The set of all I-environments is written Env^I. For $\rho \in Env^I$, $x \in Var_\sigma$, $d \in D^I_\sigma$, let $\rho[x \mapsto d]$ be the I-environment with domain $dom(\rho) \cup \{x\}$ which maps x to d and is everywhere else equal to ρ. The semantic valuation function induced by I is $[\![\_]\!]^I : \Lambda_\mathcal{T} \to Env^I \to \bigcup_{\sigma\in\mathcal{T}} D^I_\sigma$, defined in Figure 2.2. It will always be assumed that when an expression e is evaluated in an environment ρ, the free variables of e are contained in $dom(\rho)$. Our use of λ-abstraction on the right hand side of the definition of $[\![\_]\!]^I$ may be justified by the fact that the category of domains and continuous maps is cartesian closed (see [LS86]).

A simple induction on the structure of terms serves to show that for any environment $\rho \in Env^I$, if $e : \sigma$ then $([\![e]\!]^I\rho) \in D^I_\sigma$. If e is closed then the value of $[\![e]\!]^I\rho$ does not depend on ρ and in this case we will sometimes just write $[\![e]\!]^I$ for this value.

Definition 2.2.1 *If an interpretation J is such that D^J_σ is a finite lattice for each $\sigma \in \mathcal{T}$, we say that J is a* finite interpretation.

Note that for interpretations as we have defined them over $\mathcal{T}$, an interpretation J is finite if and only if D^J_ι is a finite lattice for each $\iota \in \mathcal{T}_0$. Note also that for a finite interpretation, $[D^J_\sigma \to D^J_\tau]$ is just the finite lattice of *monotone* maps from D^J_σ to D^J_τ. For any finite interpretation, assuming a (necessarily finite) tabulation of the

$$
\begin{array}{lcl}
\multicolumn{3}{l}{[\![\_]\!]^I : \Lambda_{\mathcal{T}} \to Env^I \to \bigcup_{\sigma \in \mathcal{T}} D^I_\sigma} \\
[\![x]\!]^I \rho & = & \rho(x) \\
[\![c]\!]^I \rho & = & c^I \\
[\![\lambda\, x \,.\, e]\!]^I \rho & = & \lambda\, d \in D^I_\sigma \,.\, [\![e]\!]^I \rho[x \mapsto d] \quad \text{if } x \in Var_\sigma \\
[\![e_1\ e_2]\!]^I \rho & = & ([\![e_1]\!]^I \rho)([\![e_2]\!]^I \rho) \\
[\![(e_1, e_2)]\!]^I \rho & = & ([\![e_1]\!]^I \rho, [\![e_2]\!]^I \rho) \\
[\![\mathbf{fst}(e)]\!]^I \rho & = & \pi_1([\![e]\!]^I \rho) \\
[\![\mathbf{snd}(e)]\!]^I \rho & = & \pi_2([\![e]\!]^I \rho)
\end{array}
$$

Figure 2.2: The Semantic Valuation Function Induced by I

interpretation for each constant, the interpretation of any term can also be finitely tabulated. Put another way, let J be a finite interpretation, let $\rho \in Env^J$ and for each σ let the predicate P_σ be defined by

$$P_\sigma(e, a) \iff [\![e]\!]^J \rho = a$$

for $e : \sigma$ and $a \in D^J_\sigma$. Then if the predicate $P_\tau(c, \_)$ is decidable for each constant $c : \tau$, it follows that the predicate $P_\sigma(e, \_)$ is decidable for each expression $e : \sigma$.

2.2.1 The Standard Interpretation

The *standard* interpretation, **S** (S for Standard) is shown in Figure 2.3 and hopefully contains no surprises. The standard interpretations of *bool* and *int* are the flat domains of booleans and integers respectively. The boolean and integer constants are interpreted accordingly. Each $\mathbf{if}_\sigma$ is interpreted as a conditional and each $\mathbf{Y}_\sigma$ is interpreted as a least fixed point operator.

2.3 Strictness and Scott-Closed Sets

In this section we sketch the way in which the Scott-closed subsets of a domain can be viewed as a class of properties related to the strictness of functions over those domains.

Let D and E be domains (more generally, they could just be posets with least elements). A function $f : D \to E$ is said to be *strict* if $f(\bot_D) = \bot_E$. Informa-

$$D^{\mathrm{s}}_{bool} = \mathbf{B} = \{tt, \mathit{ff}\}_\perp \qquad D^{\mathrm{s}}_{int} = \mathbf{Z} = \{\ldots, -1, 0, 1, \ldots\}_\perp$$

$$\mathbf{n}^{\mathrm{s}} = n$$

$$\mathbf{true}^{\mathrm{s}} = tt \qquad \mathbf{false}^{\mathrm{s}} = \mathit{ff}$$

$$\mathbf{iszero}^{\mathrm{s}}\ n = \begin{cases} \perp & \text{if } n = \perp \\ tt & \text{if } n = 0 \\ \mathit{ff} & \text{otherwise} \end{cases}$$

$$\mathbf{plus}^{\mathrm{s}}\ n\ m = \begin{cases} \perp & \text{if } n = \perp \text{ or } m = \perp \\ n + m & \text{otherwise} \end{cases}$$

$$\mathbf{minus}^{\mathrm{s}}\ n\ m = \begin{cases} \perp & \text{if } n = \perp \text{ or } m = \perp \\ n - m & \text{otherwise} \end{cases}$$

$$\mathbf{mult}^{\mathrm{s}}\ n\ m = \begin{cases} \perp & \text{if } n = \perp \text{ or } m = \perp \\ n * m & \text{otherwise} \end{cases}$$

$$\mathbf{if}^{\mathrm{s}}_{\sigma}\ v\ d\ d' = \begin{cases} \perp^{\mathrm{s}}_{\sigma} & \text{if } v = \perp \\ d & \text{if } v = tt \\ d' & \text{if } v = \mathit{ff} \end{cases}$$

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

Figure 2.3: The Standard Interpretation

tion about the strictness of functions can be useful when compiling lazy functional languages, because knowing that a function is strict allows evaluation order to be changed without compromising the intended semantics of a program, thus increasing scope for optimisation and exploitation of parallelism. See [Myc81], [BHA86], [Bur87] and [Bur91] for extensive discussions of the motivations.

An idea which is implicit in [BHA86], and one which is brought out more explicitly in [Bur87], is that non-empty Scott-closed sets can be used to describe a range of properties, which we call *strictness properties*, including that of a function simply being strict. Let D be a domain and let $X \subseteq D$. Then X is *Scott-closed* if:

1. X is lower;

2. whenever $\{d_n\}$ is an ω-chain in D such that each $d_n \in X$, then $\bigsqcup\{d_n\} \in X$.

Let $Y \subseteq D$: the *Scott closure* of Y, written Y^*, is the smallest Scott-closed subset of D containing Y. The set of all non-empty Scott-closed subsets of D, written $\mathcal{P}_H(D)$ is known as the *Hoare power domain* and forms a complete meet semi-lattice when ordered by subset inclusion, with arbitrary meets given by intersection (adopting the convention that $\bigcap \emptyset = D$). Note that the least element of $\mathcal{P}_H(D)$ is just $\{\bot\}$.

The general form of a strictness property for a function $f : D \to E$ is taken to be:

$$f(X) \subseteq Y \tag{2.3.1}$$

with $X \in \mathcal{P}_H(D)$ and $Y \in \mathcal{P}_H(E)$, where $f(X) = \{f(d) \mid d \in X\}$. Strictness of f can be described easily in this form, since $f(\bot_D) = \bot_E$ if and only if $f(\{\bot_D\}) \subseteq \{\bot_E\}$. The usefulness of the general form of strictness property is most easily seen by an example involving lists (although our language does not cater for functions on lists we will consider extensions to $\mathcal{T}$ and $\Lambda_{\mathcal{T}}$ which remedy this in Chapter 6). Suppose that L is a domain of finite, partial and infinite lists of elements from $\mathbf{Z}$. Let *len* and *sum* be functions in $[L \to \mathbf{Z}]$ such that

$$\begin{array}{lcl@{\qquad}lcl} len\ [\,] & = & 0 & sum\ [\,] & = & 0 \\ len\ n : l & = & 1 + (len\ l) & sum\ n : l & = & n + (sum\ l) \end{array}$$

where $[\,]$ is the empty list and $n : l$ is the list with head n and tail l. The function *len* calculates the length of a list of integers and *sum* is the function which sums the members of a list of integers. It should be reasonably clear that both these functions are strict, but we can say more than this. For example, $len\ l = \bot$ whenever l is either $\perp_L$, or ends in $\perp_L$, or is infinite. Let *Inf* be the set of all such l:

$$\{n_1 : \ldots : n_k : \perp_L \mid k \geq 0, n_1, \ldots, n_k \in \mathbf{Z}\}^*$$

(another description of *Inf* is as the set of all those elements of L which are *not* lists ending in []). We can also see that *sum* is "stricter" than *len*, in the sense that $sum\ l = \perp$ whenever l is *either* in *Inf* *or* contains an undefined element. Let *Fin* be the set:

$$\mathit{Inf} \cup \{n_1 : \ldots : n_k : [\,] \mid k \geq 1, n_1, \ldots, n_k \in \mathbf{Z}, \exists i : 1 \leq i \leq k.\, n_i = \perp\}$$

Both *Inf* and *Fin* are non-empty Scott-closed sets. Thus we can describe properties of *len* and *sum* in form (2.3.1) which are more informative than simple strictness:

$$\begin{array}{rcl} len(\mathit{Inf}) & \subseteq & \{\perp\} \\ sum(\mathit{Fin}) & \subseteq & \{\perp\}\,. \end{array}$$

If *len* and *sum* were defined in a lazy functional programming language, knowledge of these strictness properties could be used to advantage when compiling or interpreting their definitions. They tell us not only that arguments to *len* and *sum* can be evaluated eagerly or in parallel with evaluation of the application but that they can be evaluated further than the outermost constructor: the whole structure of the list can be evaluated and in the case of *sum* all the elements can be evaluated to normal form (see [Bur87, Bur91]). To see how strictness properties can be composed, define $sing : \mathbf{Z} \to L$ by

$$sing\ n = n : [\,].$$

Now although *sing* is not strict, it *is* the case that $(sing\ n) \in \mathit{Fin}$ if $n = \perp$, i.e.,

$$sing(\{\perp\}) \subseteq \mathit{Fin}.$$

Combining this with our knowledge of *sum*, we can conclude that

$$sum \circ sing(\{\perp\}) \subseteq \{\perp\}\,,$$

i.e., that $sum \circ sing$ is strict. (This is not true for $len \circ sing$, since $sing(\{\perp\}) \not\subseteq \mathit{Inf}$.)

2.4 Abstract Interpretation

In this section we outline the development of a framework for abstract interpretation of the simply typed lambda calculus due to Abramsky ([Abr90][1]). Our main aim is to introduce the central ideas on which the following chapters are based. A secondary aim is to convince the reader that the framework is conceptually simple, has a well developed theory, and hence that extending its applicability to a richer class of program analyses is a worthwhile enterprise. We proceed by way of a well known example - strictness analysis. The particular analysis we consider is that developed in [BHA86], which was the first such analysis formalised using abstract interpretation for a higher-order functional language.

2.4.1 A Finite Interpretation for Strictness Analysis

We wish to reason about the strictness properties of functions defined in $\Lambda_{\mathcal{T}}$ under the standard interpretation. For each type σ, we have identified a complete meet semi-lattice of properties of interest, namely $\mathcal{P}_H(D^{\mathbf{S}}_\sigma)$. The goal is to construct a finite interpretation $\mathbf{B}$ (B for BHA) such that for a term $e : \sigma \to \tau$, the finite interpretation $[\![e]\!]^{\mathbf{B}}$ will allow us to infer strictness properties of form (2.3.1) for the standard interpretation $[\![e]\!]^{\mathbf{S}}$. The elements of $D^{\mathbf{B}}_\sigma$ are known as *abstract* values (of type σ) and by contrast, the elements of $D^{\mathbf{S}}_\sigma$ are known as *concrete* values. The idea is that each abstract value of type σ should correspond to a *property* of concrete values of type σ, which in this case is taken to mean a member of $\mathcal{P}_H(D^{\mathbf{S}}_\sigma)$.

In accordance with our requirement that $\mathbf{B}$ be finite, we choose a finite subset of properties with which to work. At the base types an obvious choice is to settle for the two extremes, the least and greatest elements of $\mathcal{P}_H(D^{\mathbf{S}}_\iota)$, these being $\{\bot_\iota\}$ and $D^{\mathbf{S}}_\iota$ respectively. The finite interpretation of each base type $D^{\mathbf{B}}_\iota$ is thus chosen to be the two point lattice $\mathbf{2} = \{0, 1\}$, where $0 \sqsubseteq 1$. The intention is that the bottom point 0 corresponds to $\{\bot_\iota\}$ and the top point 1 corresponds to $D^{\mathbf{S}}_\iota$. One way of formalising this correspondence is in terms of *concretisation* maps $\gamma^{\mathbf{B}}_\iota : D^{\mathbf{B}}_\iota \to \mathcal{P}_H(D^{\mathbf{S}}_\iota)$, defined by

- $\gamma^{\mathbf{B}}_\iota\ 0 = \{\bot_\iota\}$;
- $\gamma^{\mathbf{B}}_\iota\ 1 = D^{\mathbf{S}}_\iota$.

---

[1] Earlier versions of [Abr90] were circulated to a number of researchers in manuscript form from September 1985 onwards, one version going under the title 'Strictness Analysis via Logical Relations'.

Given this understanding of the base types, what might we expect for the constants? Consider **plus**. We may think of an application of the finite interpretation of **plus**, say ($\mathbf{plus}^{\mathbf{B}}\ a\ b$) as representing a set of applications of the standard interpretation:

$$X = \{\mathbf{plus}^{\mathbf{S}}\ n\ m \mid n \in (\gamma^{\mathbf{B}}_{int}\ a), m \in (\gamma^{\mathbf{B}}_{int}\ b)\}\,.$$

Clearly, since $\mathbf{plus}^{\mathbf{S}}$ is strict in both arguments, X will be $\{\bot\}$ if either $a = 0$ or $b = 0$. On the other hand, if both $a = 1$ and $b = 1$, then X will just be $\mathbf{Z}$. We want the value of $\mathbf{plus}^{\mathbf{B}}\ a\ b$ to be consistent with this view, so we define $\mathbf{plus}^{\mathbf{B}}$ by

$$\mathbf{plus}^{\mathbf{B}}\ a\ b = \begin{cases} 0 & \text{if } a = 0 \text{ or } b = 0 \\ 1 & \text{otherwise} \end{cases}$$

or, more concisely, $\mathbf{plus}^{\mathbf{B}}\ a\ b = a \sqcap b$. Now consider the first-order conditional $\mathbf{if}_{int}$ in the same way. Corresponding to an application ($\mathbf{if}^{\mathbf{B}}_{int}\ a\ b\ b'$) we may think of the set:

$$Y = \{\mathbf{if}^{\mathbf{S}}_{int}\ v\ d\ d' \mid v \in (\gamma^{\mathbf{B}}_{bool}\ a), d \in (\gamma^{\mathbf{B}}_{int}\ b), d' \in (\gamma^{\mathbf{B}}_{int}\ b')\}$$

If $a = 0$ then Y is $\{\bot\}$. If $a = 1$ then, since $\gamma^{\mathbf{B}}_{bool}\ 1 = \{\bot, tt, f\!f\}$, we have $Y = \{\bot\} \cup (\gamma^{\mathbf{B}}_{int}\ b) \cup (\gamma^{\mathbf{B}}_{int}\ b')$. A little thought will show that this is just $\gamma^{\mathbf{B}}_{int}(b \sqcup b')$[2], leading to the definition of $\mathbf{if}^{\mathbf{B}}_{int}$ as:

$$\mathbf{if}^{\mathbf{B}}_{int}\ a\ b\ b' = \begin{cases} 0 & \text{if } a = 0 \\ b \sqcup b' & \text{otherwise} \end{cases}$$

The full interpretation **B** is shown in Figure 2.4. Note that as in the standard interpretation, each $\mathbf{Y}^{\mathbf{B}}_{\sigma}$ is a least fixed point operator. The justification for this is discussed in Chapter 4 at the end of Section 4.1. The formalisation of the correspondence between points in **2** and sets in $\mathcal{P}_H(D^{\mathbf{S}}_{\iota})$, and the extension of this correspondence to the higher types, is the key to a full understanding of the interpretations of constants in **B** and to showing how **B** can be used to reason about strictness properties in **S**. This is the subject of sub-section 2.4.3 but first we pause to introduce some notation for relations.

[2]To some extent this is a happy accident, since in general joins are not preserved by concretisation maps.

$$D^{\mathbf{B}}_{\iota} = \mathbf{2} = \begin{array}{c} 1 \\ | \\ 0 \end{array}$$

$$\begin{aligned}
&\mathbf{n}^{\mathbf{B}} = 1 \text{ (the top of } \mathbf{2}\text{, not the integer!)} \\
&\mathbf{true}^{\mathbf{B}} = \mathbf{false}^{\mathbf{B}} = 1 \\
&\mathbf{iszero}^{\mathbf{B}}\ a = a \\
&\mathbf{plus}^{\mathbf{B}}\ a\ b = a \sqcap b \\
&\mathbf{minus}^{\mathbf{B}} = \mathbf{mult}^{\mathbf{B}} = \mathbf{plus}^{\mathbf{B}} \\
&\mathbf{if}^{\mathbf{B}}_{\sigma}\ a\ b_1\ b_2 = \begin{cases} \bot^{\mathbf{B}}_{\sigma} & \text{if } a = 0 \\ b_1 \sqcup b_2 & \text{if } a = 1 \end{cases} \\
&\mathbf{Y}^{\mathbf{B}}_{\sigma}\ f = \bigsqcup_{i \in \omega} f^i\ \bot^{\mathbf{B}}_{\sigma}
\end{aligned}$$

Figure 2.4: A Finite Interpretation for Strictness Analysis

2.4.2 Relations

We write $R : A \leftrightarrow B$ to mean that R is a relation between sets A and B. The *domain* of a relation R is the set $\{a \mid \exists b.\, a\ R\ b\}$, the *range* of R is the set $\{b \mid \exists a.\, a\ R\ b\}$, and the *graph* of R is the set $\{(a, b) \mid a\ R\ b\}$. The set of all relations between A and B is denoted $\mathcal{R}(A, B)$. Equality on relations is extensional, that is, $P = Q$ means $a\ P\ b \iff a\ Q\ b$. Relations $P, Q \in \mathcal{R}(A, B)$ are naturally ordered by *implication*, defined thus:

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

This is a partial order, in particular $P = Q \iff P \leq Q$ *and* $Q \leq P$.

For $\{R_i\}_{i \in I}$ a family of relations in $\mathcal{R}(A, B)$, the *conjunction* of the R_i is written $\bigwedge_{i \in I} R_i$ and is defined by $a\ (\bigwedge_{i \in I} R_i)\ b \iff \forall i \in I.\, a\ R_i\ b$. Note that if $I = \emptyset$ then $\bigwedge_{i \in I} R_i$ is the universal relation between A and B (we should really decorate $\bigwedge$ to specify A and B but we will rely on context instead). It is easy to see that $(\mathcal{R}(A, B), \leq, \bigwedge)$ is a complete meet semi-lattice and hence a complete lattice. For $P_1 : A_1 \leftrightarrow B_1$ and $P_2 : A_2 \leftrightarrow B_2$, the *product* of P_1 and P_2 is $P_1 \times P_2 : A_1 \times A_2 \leftrightarrow B_1 \times B_2$, defined by

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

Meets of products can be calculated elementwise, i.e., $(a_1, a_2)\ (\bigwedge_{i \in I} P_{i,1} \times P_{i,2})\ (b_1, b_2) \iff a_1\ (\bigwedge_{i \in I} P_{i,1})\ b_1$ *and* $a_2\ (\bigwedge_{i \in I} P_{i,2})\ b_2$.

We could have been more economical with our notation and simply taken a relation between A and B to be a subset of $A \times B$, identifying R with its graph. In this case $P \leq Q$ would be synonymous with $P \subseteq Q$ and $\bigwedge_{i \in I} R_i$ would just be $\bigcap_{i \in I} R_i$. Note however, that the product of P_1 and P_2 would *not* be the cartesian product of the sets P_1 and P_2. It is partly to avoid the possibility of such confusions that we adopt this slightly more abstract notion of relation.

2.4.3 Concretisation Maps and Logical Relations

There are a number of ways in which the relation between a finite interpretation and the standard interpretation might be formalised. The approach we began to sketch above, and shall develop further, is to concentrate on the concretisation maps (our $\gamma^{\mathbf{B}}$ maps). This is also the basis of the approach taken by the Cousots in their original work on abstract interpretation of imperative flow-chart languages. Mycroft's development of an abstract interpretation technique for first-order functional languages ([Myc81]) focussed on the use of an *abstraction map*, *abs*, from a standard domain to the corresponding abstract domain such that $abs(d)$ was the 'best' abstract value which correctly described d (see Subsection 2.4.4). From this starting point, Mycroft *induced* concretisation maps of a similar nature to the γ_ι maps informally described above, but using the Plotkin power domain rather than the Hoare power domain. This idea was extended to the higher-order case in [BHA86], this time using the Hoare power domain. By contrast with both of these approaches, [Abr90] takes a directly *relational* approach to the problem. (It should be pointed out that this was not the first approach to using relations in this way. The idea dates back at least to [Nie84]. Closer to [Abr90] is the relational framework of [MJ85] which considers the untyped lambda calculus.)

At each type σ, [Abr90] defines a binary relation $R_\sigma : D_\sigma^{\mathbf{S}} \leftrightarrow D_\sigma^{\mathbf{B}}$ to describe the way in which abstract and concrete values are related. This approach eliminates the need to use any theory of power domains and leads to an extremely simple proof of correctness. The development hinges on the following definition, an instance of an idea originally due to M. Gordon ([Plo73]).

Definition 2.4.1 (Binary Logical Relation) *Let I and J be interpretations. A relation R between I and J, written $R : I \leftrightarrow J$, is a type indexed family of binary relations $\{R_\sigma\}_{\sigma \in \mathcal{T}}$ with $R_\sigma : D_\sigma^I \leftrightarrow D_\sigma^J$. Such an R is* logical *if for all $\sigma, \tau \in \mathcal{T}$:*

1. $f \; R_{\sigma \to \tau} \; h \iff \forall d \in D_\sigma^I, a \in D_\sigma^J .\, d \; R_\sigma \; a \Rightarrow (f \; d) \; R_\tau \; (h \; a)$;

2. $(d_1, d_2) \; R_{\sigma_1 \times \sigma_2} \; (a_1, a_2) \iff d_1 \; R_{\sigma_1} \; a_1$ *and* $d_2 \; R_{\sigma_2} \; a_2$.

Informally, a logical relation is one which relates functions which map related arguments to related results, and which relates pairs elementwise. Note that a logical relation is completely determined by its base type members, so that to specify a logical relation R we need only define the R_ι. A relation $R : I \leftrightarrow J$ can be extended to environments in a pointwise manner: let $\rho \in Env^I$ and $\delta \in Env^J$, then we write $\rho\ R\ \delta$ to mean that $dom(\rho) = dom(\delta)$ and $\rho(x)\ R_\sigma\ \delta(x)$ for all $x \in dom(\rho) \cap Var_\sigma$, for all σ.

Definition 2.4.2 *An* abstract interpretation *is a pair* (J, R^J) *where* J *is a finite interpretation and* $R^J\colon \mathbf{S} \leftrightarrow J$ *is a logical relation.*

If we wish to understand abstract interpretation in terms of concretisation maps instead of relations, we can easily do so by exploiting the isomorphism $\mathcal{R}(A, C) \cong C \to \wp A$: given a logical relation $R : \mathbf{S} \leftrightarrow J$, we define the family of concretisation maps $\{\gamma_\sigma\}_{\sigma \in \mathcal{T}}$ with $\gamma_\sigma : D^J_\sigma \to \wp D^{\mathbf{S}}_\sigma$, by:

$$\gamma_\sigma\ a = \{d \in D^{\mathbf{S}}_\sigma \mid d\ R_\sigma\ a\}\,. \tag{2.4.3}$$

The logical relation used to establish correctness of $\mathbf{B}$ is $R^{\mathbf{B}}\colon \mathbf{S} \leftrightarrow \mathbf{B}$, defined as follows:

- $d\ R^{\mathbf{B}}_\iota\ 0 \iff d = \bot^{\mathbf{S}}_\iota$;
- $d\ R^{\mathbf{B}}_\iota\ 1$ for all $d \in D^{\mathbf{S}}_\iota$.

Note that if we define $\{\gamma^{\mathbf{B}}_\sigma\}$ in terms of $R^{\mathbf{B}}$ via (2.4.3), the base type members $\gamma^{\mathbf{B}}_\iota$ will have the same definition as in Subsection 2.4.1.

The correctness requirement for $\mathbf{B}$ is that whenever d and a are the standard and finite interpretations respectively of some closed[3] term $e : \sigma$, then it is always the case that $d\ R^{\mathbf{B}}_\sigma\ a$. To see that this is what we need, consider the case of $e : \iota \to \jmath$ with standard interpretation f and finite interpretation h. Suppose that $h\ 0 = 0$. Now if $f\ R^{\mathbf{B}}_{\iota\to\jmath}\ h$ then since $R^{\mathbf{B}}$ is logical and $\bot^{\mathbf{S}}_\iota\ R^{\mathbf{B}}_\iota\ 0$, we have $(f\ \bot^{\mathbf{S}}_\iota)\ R^{\mathbf{B}}_\jmath\ (h\ 0) = 0$. But the *only* $d \in D^{\mathbf{S}}_\jmath$ such that $d\ R^{\mathbf{B}}_\jmath\ 0$ is $\bot^{\mathbf{S}}_\jmath$, so $f\ \bot^{\mathbf{S}}_\iota = \bot^{\mathbf{S}}_\jmath$. Thus the proposed correctness condition guarantees that for expressions denoting functions at the base types, strictness of the finite interpretation implies strictness of the standard interpretation. It can easily be shown ([Abr90]) that this implication holds for functions

[3]This is just a convenience which allows us to consider the interpretations of e without specifying an environment. The restriction is dropped in the formal statement and proof of correctness.

at all types. The importance of this fact is that strictness in the finite interpretation is decidable so we have a sound and effective test for strictness in the standard interpretation. (This does not mean that a practical implementation of the test is straightforward: see Chapters 7–9 and [HH91].) While sound, the test cannot be complete, since in our language, which has the power of a universal Turing machine, any test for termination can easily be reduced to a test for strictness. A simple example illustrating the weakness of the test is given by the term $\lambda\, \mathtt{x}\,.\, \mathbf{if}_{int}\ \mathbf{true}\ \mathtt{x}\ \mathbf{3}$, which clearly denotes a strict function under the standard interpretation but which has finite interpretation $\lambda\, a \in \mathbf{2}\,.\, 1$. Chapter 4 contains a rather more detailed discussion of how correctness allows us to reason in a useful way about the standard interpretation of terms.

The formal proof of correctness of **B** uses the following theorem, due in this particular form to [Abr90] (Proposition 3.2) but originally due to Plotkin ([Plo80], Proposition 1, see also [Sta85], Fundamental Theorem of Logical Relations).

Theorem 2.4.4 (The Binary Logical Relations Theorem) *Let I and J be interpretations and let $R : I \leftrightarrow J$ be a logical relation. Suppose that $c^I R_\tau c^J$ for each τ and for each $c : \tau$. Then for all σ, for all $e : \sigma$, for all $\rho \in Env^I$ and $\delta \in Env^J$:*

$$\rho\ R\ \delta \Rightarrow (\llbracket e \rrbracket^I \rho)\ R_\sigma\ (\llbracket e \rrbracket^J \delta).$$

The formal statement of correctness for **B** is as follows: **B** is *correct* if for all σ, for all $e : \sigma$, for all $\rho \in Env^{\mathbf{S}}$ and $\delta \in Env^{\mathbf{B}}$:

$$\rho\ R^{\mathbf{B}}\ \delta \Rightarrow (\llbracket e \rrbracket^{\mathbf{S}} \rho)\ R^{\mathbf{B}}_\sigma\ (\llbracket e \rrbracket^{\mathbf{B}} \delta).$$

Since $R^{\mathbf{B}}$ is logical, the Binary Logical Relations Theorem implies that for **B** to be correct it is sufficient that the $c^{\mathbf{B}}$ be correct, i.e., that $c^{\mathbf{S}}\ R^{\mathbf{B}}_\tau\ c^{\mathbf{B}}$ for each $c : \tau$ (it is also clearly necessary, since the expressions include the constants). This idea of reducing a global correctness condition to local conditions on the constants is a central one in the Cousots' original work ([CC77]), and can fairly be said to be the cornerstone of any abstract interpretation framework.

The correctness of the constants is easily established for **B** ([Abr90]) but in itself this is rather weak because $R^{\mathbf{B}}$ is $\top$-*universal*, which is to say that each $D^{\mathbf{B}}_\sigma$ has a greatest element $\top^{\mathbf{B}}_\sigma$, and $d\ R^{\mathbf{B}}_\sigma\ \top^{\mathbf{B}}_\sigma$ for all $d \in D^{\mathbf{S}}_\sigma$. This means that it would be *correct* to take $c^{\mathbf{B}} = \top^{\mathbf{B}}_\sigma$ for all $c : \sigma$, but of course the resulting strictness test would be extremely poor. However, it can be shown that the interpretations of the constants in **B** are not only correct but are actually the *best possible* interpretations, given the interpretations of the types and the definition of $R^{\mathbf{B}}$.

2.4.4 Best Interpretations for Constants

To see that best interpretations exist, it helps to re-introduce the concretisation maps described above (this is not quite the approach taken in [Abr90] but seems more intuitive in some respects). The $\gamma_\iota^{\mathbf{B}}$ are extended to a type indexed family $\gamma^{\mathbf{B}} = \{\gamma_\sigma^{\mathbf{B}}\}$ with $\gamma_\sigma^{\mathbf{B}} : D_\sigma^{\mathbf{B}} \to \mathcal{P}_H(D_\sigma^{\mathrm{S}})$ by taking

$$\gamma_\sigma^{\mathbf{B}}\ a = \{d \in D_\sigma^{\mathrm{S}} \mid d\ R_\sigma^{\mathbf{B}}\ a\}$$

for each $a \in D_\sigma^{\mathbf{B}}$. One way of verifying that this makes the $\gamma_\sigma^{\mathbf{B}}$ well defined (i.e., that each $\gamma_\sigma^{\mathbf{B}}\ a$ is non-empty and Scott-closed) is to demonstrate ([Abr90]) that each $R_\sigma^{\mathbf{B}}$ is:

1. *strict* : $\bot_\sigma^{\mathrm{S}}\ R_\sigma^{\mathbf{B}}\ \bot_\sigma^{\mathbf{B}}$;
2. *S-monotone* : $d' \sqsubseteq d\ R_\sigma^{\mathbf{B}}\ a \sqsubseteq a' \Rightarrow d'\ R_\sigma^{\mathbf{B}}\ a'$;
3. *inductive* : whenever $\{d_n\}$ and $\{a_n\}$ are ω-chains such that $d_n\ R_\sigma^{\mathbf{B}}\ a_n$ for all n, then $\bigsqcup\{d_n\}\ R_\sigma^{\mathbf{B}}\ \bigsqcup\{a_n\}$.

S-monotonicity also ensures that the $\gamma^{\mathbf{B}}$ maps are monotone. The statement that $c^{\mathbf{B}}$ is a correct interpretation for $c : \sigma$ is then equivalent to the statement that $c^{\mathrm{S}} \in (\gamma_\sigma^{\mathbf{B}}\ c^{\mathbf{B}})$. Now the smaller the $c^{\mathbf{B}}$ are, the smaller will be the value of $[\![e]\!]^{\mathbf{B}}$ for any given $e : \sigma$, and hence, since $\gamma_\sigma^{\mathbf{B}}$ is monotone, the smaller the set $(\gamma_\sigma^{\mathbf{B}}\ [\![e]\!]^{\mathbf{B}})$. Clearly, assuming the correctness of $\mathbf{B}$, the smaller $(\gamma_\sigma^{\mathbf{B}}\ [\![e]\!]^{\mathbf{B}})$ the more we know about $[\![e]\!]^{\mathrm{S}}$ (in the extreme case that $\gamma_\sigma^{\mathbf{B}}\ [\![e]\!]^{\mathbf{B}} = \{\bot\}$, we know that $[\![e]\!]^{\mathrm{S}} = \bot$). The question naturally arises: for any $c \in \mathit{Con}$, is there a least value for $c^{\mathbf{B}}$ which is correct? More generally: for any $d \in D_\sigma^{\mathrm{S}}$, is there a least $a \in D_\sigma^{\mathbf{B}}$ such that $d \in (\gamma_\sigma^{\mathbf{B}}\ a)$? The answer is yes, and this can be shown by demonstrating the following:

Fact 2.4.5 *Each $\gamma_\sigma^{\mathbf{B}}$ preserves meets, i.e., for any $S \subseteq D_\sigma^{\mathbf{B}}$*

$$\gamma_\sigma^{\mathbf{B}}(\textstyle\bigsqcap S) = \displaystyle\bigcap_{a \in S} (\gamma_\sigma^{\mathbf{B}}\ a)$$

(this can be proved by a simple adaptation of the proof of our Proposition 4.2.5).

The import of this fact is that it follows (see Section 4.2) that $\gamma_\sigma^{\mathbf{B}}$ has a *left adjoint* (here just the *lower component* of a *Galois connection*), that is to say, a monotone map $\alpha^{\mathbf{B}}_{\sigma} : \mathcal{P}_H(D^{\mathbf{S}}_{\sigma}) \to D^{\mathbf{B}}_{\sigma}$ such that for all $X \in \mathcal{P}_H(D^{\mathbf{S}}_{\sigma})$ and $a \in D^{\mathbf{B}}_{\sigma}$:

$$X \subseteq (\gamma^{\mathbf{B}}_{\sigma}\ a) \iff \alpha^{\mathbf{B}}_{\sigma}(X) \sqsubseteq a.$$

Now for any $X \in \mathcal{P}_H(D^{\mathbf{S}}_{\sigma})$, $d \in X$ if and only if $\downarrow d \subseteq X$, and so for any $a \in D^{\mathbf{B}}_{\sigma}$:

$$d \in (\gamma^{\mathbf{B}}_{\sigma}\ a) \iff \downarrow d \subseteq (\gamma^{\mathbf{B}}_{\sigma}\ a) \iff \alpha^{\mathbf{B}}_{\sigma}(\downarrow d) \sqsubseteq a.$$

In other words, $\alpha^{\mathbf{B}}_{\sigma}(\downarrow d)$ is the least $a \in D^{\mathbf{B}}_{\sigma}$ such that $d \in (\gamma^{\mathbf{B}}_{\sigma}\ a)$. The family of maps $abs = \{abs_{\sigma}\}$ with $abs_{\sigma} : D^{\mathbf{S}}_{\sigma} \to D^{\mathbf{B}}_{\sigma}$, is therefore defined by taking $abs_{\sigma}(d) = \alpha^{\mathbf{B}}_{\sigma}(\downarrow d)$ so that the best finite interpretation for any constant $c : \sigma$ is given by setting $c^{\mathbf{B}} = abs_{\sigma}(c^{\mathbf{S}})$. [Abr90] shows how the family abs may be defined inductively by:

- $abs_{\iota}(d) = \begin{cases} 0 & \text{if } d = \bot \\ 1 & \text{otherwise} \end{cases}$
- $abs_{\sigma\to\tau}(f) = \lambda\, a \in D^{\mathbf{B}}_{\sigma} \,.\, \bigsqcup\{abs_{\tau}(f\ d) \mid abs_{\sigma}(d) \sqsubseteq a\}$
- $abs_{\sigma\times\tau}(d, d') = (abs_{\sigma}(d), abs_{\tau}(d'))$.

(The last clause is our own addition since [Abr90] does not consider product types.)

2.5 Polymorphism

The language we have chosen to consider is only simply typed and thus does not allow *polymorphic* function definitions. This is a serious limitation, since polymorphic type systems contribute greatly to the expressive power and clarity of functional languages such as Miranda and ML. The failure to deal with polymorphism is not something which can easily be remedied, since giving a semantics to polymorphic languages is rather harder than for simply typed languages. In fact, the development of a good semantic model for polymorphism is still very much the subject of active research ([Fre89, BFSS90, AMSW90, AP90]). For languages using Hindley-Milner style polymorphism, there is one rather crude way of adapting analyses intended for simply typed languages. That is to resolve a polymorphic function definition into the set of its monomorphic instances which are actually used in a given program. Since there may be many such instances for each polymorphic definition and, since the instantiated types may be rather complex (hence their finite lattice interpretations may be rather large), this can lead to a very computationally expensive analysis.

One way of avoiding this expense is to establish that a *polymorphic invariance* result holds for the property of interest ([Abr86]). Given an expression of polymorphic function type $e : \forall\alpha.\, \sigma_1(\alpha) \to \sigma_2(\alpha)$, we will write the instantiation of e with $\alpha \mapsto \tau$ as e_τ. In [Abr86] it is shown that for the [BHA86] interpretation, strictness *analysis* is a polymorphic invariant, by which it is meant that:

$$(\forall\tau.\, [\![e_\tau]\!]^{\mathbf{B}}\ \bot^{\mathbf{B}}_{\sigma_1(\tau)} = \bot^{\mathbf{B}}_{\sigma_2(\tau)}) \iff ([\![e_\iota]\!]^{\mathbf{B}}\ \bot^{\mathbf{B}}_{\sigma_1(\iota)} = \bot^{\mathbf{B}}_{\sigma_2(\iota)}),$$

where τ ranges over all types and ι is any base type. This says that the [BHA86] analysis can either find the simplest instance of a polymorphic function to be strict, in which case all instances are strict, or it can find no instance to be strict. Thus if we wish to know whether various e_τ are strict, we need only interpret e_ι. Abramsky's proof of the polymorphic invariance result is very syntactic and rather heavy going. By contrast, the same result is proved much more elegantly in [AJ91], using a categorical semantics for Hindley-Milner style polymorphism based on a relational analogue to functors (this work builds on [Hug88], a precursor of [HL91]: see below). In future we hope to be able to adapt this technique to the analyses we describe in later chapters.

Unfortunately, polymorphic invariance does not give us all that we need. The **B** interpretation of a term can carry more information than just whether a function is strict. Let `twice` be the lambda term $\lambda\texttt{f}\,.\,\lambda\texttt{x}\,.\,\texttt{f(f x)}$. In the polymorphic lambda calculus this can be given the type $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$. In the simply typed lambda calculus we have to use a distinct lambda term $\lambda\texttt{f}_\tau\,.\,\lambda\texttt{x}_\tau\,.\,\texttt{f}_\tau(\texttt{f}_\tau\ \texttt{x}_\tau)$, with $\texttt{f}_\tau : \tau \to \tau$ and $\texttt{x}_\tau : \tau$, for each instance $\texttt{twice}_\tau$ required. Now the **B** interpretation of the term

$$\texttt{twice}_{int\to int}\ (\lambda\texttt{g}\,.\,\lambda\texttt{n}\,.\,\mathbf{plus}\ \mathbf{1}\ (\texttt{g n}))$$

is just the identity on $[\mathbf{2} \to \mathbf{2}]$, which shows the term to denote a strict function under **S**. But this cannot be determined from the **B** interpretation of $\texttt{twice}_\iota$ using Abramsky's polymorphic invariance result, which only reveals that $\texttt{twice}_{int\to int}$ is strict in its first argument[4].

An alternative to [Myc81, BHA86] style abstract interpretation for functional languages, is the *projection* based analysis technique of [WH87]. This is limited to

[4]Baraki's recent work ([Bar91a]), using a semantic model of polymorphism very similar to that used in [AJ91], indicates that it may be possible to do better than this by using the abstract interpretation of the simplest instance of a polymorphic function to place an upper bound on the abstract interpretation of more complex instances.

first-order languages but is able to capture properties which cannot be expressed in the [Myc81, BHA86] frameworks. For strictness analysis of first-order functions using projections, [HL91] shows that using *polymorphic projections* can give us rather more than polymorphic invariance. However, this work exploits the correspondence between polymorphic functions and natural transformations (a fundamental notion in category theory), which only holds in the first-order case. In the following chapters we show how some aspects of the use of projections in program analysis can be generalised by the use of a certain class of binary relations, and that this allows analyses which can capture the same properties as projections to be defined for higher-order languages. It remains to be seen whether the ideas of [HL91] can be adapted to the higher-order case, but it is interesting to note that the partial equivalence relations used in the framework developed in the next two chapters play a prominent role in some of the above cited work on semantic models for polymorphism.

Chapter 3

Pers and Ternary Logical Relations

The abstract interpretation framework described in the previous chapter hinges on the use of a binary logical relation. We have seen that the framework allows us to infer properties of functions in the following way. A finite interpretation J and a logical relation $R^J : \mathbf{S} \leftrightarrow J$ are defined such that J is correct with respect to R^J. Then if $f^{\mathrm{S}} : D^{\mathrm{S}}_\sigma \to D^{\mathrm{S}}_\tau$ and $f^J : D^J_\sigma \to D^J_\tau$ are the alternative interpretations of some term $e : \sigma \to \tau$, we are guaranteed that $f^{\mathrm{S}}\ R^J_{\sigma\to\tau}\ f^J$. If γ^J is derived from R^J via (2.4.3), it is not hard to show (see Section 4.1 in the next chapter) that to say that $f^{\mathrm{S}} R^J_{\sigma\to\tau} f^J$ is equivalent to saying that for any $a \in D^J_\sigma$ and $b \in D^J_\tau$:

$$f^J\ a = b \Rightarrow f^{\mathrm{S}}(\gamma^J_\sigma\ a) \subseteq (\gamma^J_\tau\ b).$$

This was implicit in the strictness analysis example. In that example the relation $R^{\mathbf{B}}$ was such that the sets $(\gamma^{\mathbf{B}}_\sigma\ a)$ were always Scott-closed but in general this need not be the case: another analysis described in [Abr90] is a *termination analysis* based on an interpretation $\mathbf{T}$ and logical relation $R^{\mathbf{T}}$ such that the sets $(\gamma^{\mathbf{T}}_\sigma\ a)$ are always upwards closed. The point we wish to bring out here is that whatever *kinds* of sets they are, the $(\gamma^J_\sigma\ a)$ are always sets, and the properties of functions which may be inferred using abstract interpretations within Abramsky's framework are always of the form $f^{\mathrm{S}}(\gamma^J_\sigma\ a) \subseteq (\gamma^J_\tau\ b)$. It is important to be clear about this limitation, because there are many properties of interest which cannot be captured in this way. In this chapter and the next we develop a generalised version of Abramsky's framework which allows us to construct abstract interpretations for reasoning about some of these properties.

3.1 Binding Time Analysis, Projections and Equivalence Relations

One type of program analysis which exposes the above mentioned limitation is the *binding time analysis* ([Jon88]) used in partial evaluation. The goal of this analysis is to determine which parts of a program can be evaluated given partial information about the program's inputs. Suppose we have an interpreter **eval** : $code \times input \to output$, defined in a functional language in terms of a number of auxiliary function definitions

$$\begin{array}{rcl} f_1(x) & = & e_1 \\ f_2(x) & = & e_2 \\ & \vdots & \\ f_k(x) & = & e_k. \end{array}$$

The arguments to **eval** are the program to be interpreted and a vector of values for the program's run-time parameters. A much studied partial evaluation problem is the automatic generation of a compiler from the definition of the interpreter. Each time the generated compiler is applied to some program, say p, it produces as output a version of the interpreter which is specialised to p. To generate such a compiler, it is necessary to know, for each use of each f_i in the interpreter, how much of the argument to f_i will be known when the code to be compiled becomes known. In the partial evaluation literature, inputs, or arguments, or more generally, sub-expressions that will be known are termed *static* and those which will not be known are termed *dynamic*. The role of binding time analysis is to identify (a subset of) the static parts of a program. The core of a binding time analysis is a method which, for each f_i defined in a program, can determine how much of $f_i(x)$ will be static given information about how much of x is static. It is important to note that a method is required which does not need to know the actual value which x is to take, since a compiler must be generated without knowing what programs are to be compiled.

To illustrate the idea, let A and B be domains, let $b \in B$ and consider the three functions $c : A \to B$, $fst : A \times B \to A$ and $swap : A \times B \to B \times A$, with definitions:

$$\begin{array}{lcl} c(x) & = & b \\ fst(x, y) & = & x \\ swap(x, y) & = & (y, x). \end{array}$$

Intuitively, $c(x)$ is static even if x is dynamic, since c is a constant function[1], while $\mathit{fst}(x, y)$ is static as long as x is static, even if y is dynamic. Since *swap* computes a pair, it makes sense to ask whether either *component* of $\mathit{swap}(x, y)$ is static. Clearly, the second component of $\mathit{swap}(x, y)$ is static if x is, even if y is dynamic, and the first component of $\mathit{swap}(x, y)$ is static if y is, even if x is dynamic. Intuitively, it is probably helpful to think of the different properties of c, *fst* and *swap* as being different degrees of *constancy*. The function c is constant, *fst* is constant in its second argument, while *swap* is constant in its first argument if we only look at the second component of its result and vice versa.

We now consider two ways in which the concepts of 'static' and 'dynamic' may be formalised, using the above examples as illustrations. The first way uses projections and the second uses equivalence relations.

3.1.1 Projections

Definition 3.1.1 *Let D be a domain. A* projection *is a continuous map $\alpha : D \to D$ such that $\alpha \sqsubseteq id_D$ and $\alpha \circ \alpha = \alpha$.*

The set of all projections on D forms a complete lattice, $\mathit{Proj}(D)$ when ordered by the usual pointwise ordering. Joins are inherited from $[D \to D]$ but meets are not. The least projection on D is Abs_D, the constant bottom function, and the greatest projection is Id_D, the identity.

In [Lau88] it was suggested that 'static' be equated with Id and that 'dynamic' be equated with Abs. More generally, Launchbury proposed that for expressions taking their value in some structured domain, various notions of 'partly static' be formalised by projections which mapped the dynamic components to bottom and left static components alone. For a product domain $A \times B$, projections describing 'partial staticness' may be formed by taking the product of projections on A and B (the product of two projections α and β is defined in the usual way, by $\alpha \times \beta(a, b) = (\alpha\ a, \beta\ b)$). Thus if x is static and y is dynamic, the staticness of *swap*'s argument (x, y) is described by the projection $\mathrm{Id}_A \times \mathrm{Abs}_B$, and that of the expression $\mathit{swap}(x, y)$ by $\mathrm{Abs}_B \times \mathrm{Id}_A$.

From the point of view of binding time analysis, the key notion to be formalised is that of a function transforming the degree of staticness of its argument into a degree of staticness of its result. In [Lau89] it was shown that this could be done

[1] This assumes a non-strict semantics for the language. See the discussion in Chapter 5.

using the 'strictness relation' introduced in [WH87]. Suppose that the staticness of x is safely described by the projection α. (By 'safely' we mean that x is at least as static as α, though some larger projection may be more accurate. Thus Abs is always safe.) Then the fact that the staticness of $f(x)$ is safely described by β is formalised as the equation:

$$\beta \circ f = \beta \circ f \circ \alpha. \tag{3.1.2}$$

The simplest way to motivate this is by some examples.

First consider c. Now

$$\beta \circ c = \beta \circ c \circ \alpha$$

for any α and β, since $c = c \circ \alpha$ for any α. So in particular:

$$\mathrm{Id}_B \circ c = \mathrm{Id}_B \circ c \circ \mathrm{Abs}_A.$$

This formalises the fact that $c(x)$ is static even if x is dynamic.

For *fst* we have

$$\alpha \circ \mathit{fst} = \alpha \circ \mathit{fst} \circ (\alpha \times \beta),$$

since $\alpha(\mathit{fst}(\alpha\ a, \beta\ b)) = \alpha(\alpha\ a) = \alpha\ a$, using the idempotence of the projection α. Thus

$$\mathrm{Id}_A \circ \mathit{fst} = \mathrm{Id}_A \circ \mathit{fst} \circ (\mathrm{Id}_A \times \mathrm{Abs}_B),$$

formalising the fact that $\mathit{fst}(x, y)$ is static as long as x is.

Finally, it is easy to see that

$$(\beta \times \alpha) \circ \mathit{swap} = (\beta \times \alpha) \circ \mathit{swap} \circ (\alpha \times \beta),$$

and hence that

$$(\mathrm{Id}_B \times \mathrm{Abs}_A) \circ \mathit{swap} = (\mathrm{Id}_B \times \mathrm{Abs}_A) \circ \mathit{swap} \circ (\mathrm{Abs}_A \times \mathrm{Id}_B),$$

(the first component of $\mathit{swap}(x, y)$ is static if y is) and

$$(\mathrm{Abs}_B \times \mathrm{Id}_A) \circ \mathit{swap} = (\mathrm{Abs}_B \times \mathrm{Id}_A) \circ \mathit{swap} \circ (\mathrm{Id}_A \times \mathrm{Abs}_B),$$

(the second component of $\mathit{swap}(x, y)$ is static if x is).

In [Lau89], Launchbury presents a binding time analysis for a first-order functional language. The analysis is based on a non-standard semantics which interprets function definitions as maps between finite sub-lattices of projections, such that if $f^{\#}$

and f are, respectively, the non-standard and standard interpretations of a function definition, then

$$f^{\#}(\alpha) = \beta \Rightarrow \beta \circ f = \beta \circ f \circ \alpha.$$

Another analysis based on projections is the strictness analysis described in [WH87]. Again the analysis is for a first-order language, but in this case it is based on a *backwards* semantics. Thus the definition of a function $f : A_1 \times \cdots A_n \to B$ is interpreted as a family of maps $\{f^i : Proj(B) \to Proj(A_i)\}_{1 \leq i \leq n}$ such that:

$$f^i(\beta) = \alpha \Rightarrow \beta \circ f = \beta \circ f \circ (\mathrm{Id}_{A_1} \times \cdots \times \mathrm{Id}_{A_{i-1}} \times \alpha \times \mathrm{Id}_{A_{i+1}} \times \cdots \times \mathrm{Id}_{A_n}).$$

A distinctive feature of this analysis is that it is able to identify an interesting form of strictness for functions over lists, known as *head-strictness*, which cannot be discovered via a [BHA86] style analysis. In Chapter 5 we describe a forwards analysis for a higher-order language, which can also discover head-strictness.

3.1.2 Equivalence Relations

In [Lau88] it is noted that there is a natural way to associate an equivalence relation to a projection $\alpha \in Proj(D)$, namely the relation which equates those elements of D which α maps to the same value. In fact this can be done for any function, not just projections, and is a standard construction:

Definition 3.1.3 *For a function $f : A \to B$, the* kernel *of f is the equivalence relation* $\ker f : A \leftrightarrow A$ *defined by*

$$a \; (\ker f) \; a' \iff f \; a = f \; a'.$$

By concentrating on the equivalence relations rather than the projections, we arrive at an alternative (but equivalent) formalisation of the terms 'static' and 'dynamic'.

Definition 3.1.4 *For each domain D, we define $Id_D : D \leftrightarrow D$ to be* $\ker \mathrm{Id}_D$*, hence Id_D is just equality on D, and we define $All_D : D \leftrightarrow D$ to be* $\ker \mathrm{Abs}_D$*, hence All_D is the universal relation on D:*

$$d \; All_D \; d' \text{ for all } d, d' \in D.$$

We will write Id_σ to mean $Id_{D^{\mathrm{S}}_\sigma}$ and similarly for All_σ.

Now 'static' is interpreted by Id_D and 'dynamic' by All_D. To formalise the notion of a function transforming 'staticness' in this setting we introduce the following notation[2]: for $f : A \to B$ and binary relations $P : A \leftrightarrow A$ and $Q : B \leftrightarrow B$ we write $f : P \Rightarrow Q$ to mean

$$a\ P\ a' \Rightarrow f(a)\ Q\ f(a').$$

Now consider c again. Because c is constant, it is clear that $c(a) = c(a')$ for any $a, a' \in A$, or put another way, that $a\ All_A\ a' \Rightarrow c(a)\ Id_B\ c(a')$. Thus

$$c : All_A \Rightarrow Id_B$$

In fact this could be taken as a rather natural *definition* of constancy: it says that no matter how x varies (however dynamic x is), $c(x)$ remains fixed (static).

Recall that the product of two relations is defined such that $(a, b)\ P \times Q\ (a', b')$ iff $a\ P\ a'$ and $b\ Q\ b'$. So $(a, b)\ (Id_A \times All_B)\ (a', b') \iff a = a'$. Thus

$$fst : Id_A \times All_B \Rightarrow Id_A.$$

Similarly, we can see that

$$swap : Id_A \times All_B \Rightarrow All_B \times Id_A$$

and

$$swap : All_A \times Id_B \Rightarrow Id_B \times All_A.$$

The explication of the terms 'static' and 'dynamic' directly in terms of equivalence relations rather than projections may seem more intuitive than the use of projections, though as the following result shows the two approaches are equivalent.

Proposition 3.1.5 *Let $\alpha : A \to A$ and $\beta : B \to B$ be projections. Then for any function $f : A \to B$:*

$$\beta \circ f = \beta \circ f \circ \alpha \iff f : (\ker \alpha) \Rightarrow (\ker \beta)$$

Proof For the implication from left to right, suppose $a\ (\ker \alpha)\ a'$. Then $\alpha\ a = \alpha\ a'$. But by assumption, $\beta(f(\alpha\ a)) = \beta(f\ a)$ and $\beta(f(\alpha\ a')) = \beta(f\ a')$. Hence $\beta(f\ a) =$

[2]The resemblance of this notation to an assertion that f has a certain *function type* is not accidental since binary relations can indeed be used to give a semantics to types in such a way that $\Rightarrow$ is the natural interpretation for $\to$. See [Rey83, CL90] for example.

$\beta(f\ a')$ and so $(f\ a)\ (\ker\beta)\ (f\ a')$.

For the implication from right to left, for any $a \in A$, $\alpha\ a = \alpha(\alpha\ a)$, and so $a\ (\ker\alpha)\ (\alpha\ a)$. But then by assumption, $(f\ a)\ (\ker\beta)\ f(\alpha\ a)$, hence $\beta(f\ a) = \beta(f(\alpha\ a))$.

(Note that there is a somewhat more general result lurking here, since the only property of the projections which is used is idempotence of α.) □

Remark The natural ordering on projections is the usual pointwise ordering on functions, while the natural ordering on relations is inclusion. It is not too hard to see that for projections and their kernels these orderings are *dual*, in the sense that

$$\alpha \sqsubseteq \beta \iff (\ker\beta) \leq (\ker\alpha).$$

When comparing the analyses we develop in the rest of this thesis with analyses based on projections, this reversal of order should be born in mind.

3.2 Complete Partial Equivalence Relations

The above discussion shows that by using equivalence relations (or projections) it is easy to describe the property of constancy. On the other hand, it is not possible to capture the fact that $f : A \to B$ is constant via the use of a pair of sets in the form $f(X) \subseteq Y$, as allowed by the abstract interpretation framework described in the previous chapter. Clearly, we can describe the fact that f is the *particular* constant function with value b, by the statement that $f(A) \subseteq \{b\}$, but we cannot simply assert that f is constant in this way. This suggests that it would be worthwhile to consider ways of constructing program analyses for reasoning about properties of functions $f : D \to E$ which can be expressed in the form

$$f : P \Rightarrow Q \tag{3.2.1}$$

where $P : D \leftrightarrow D$ and $Q : E \leftrightarrow E$ are equivalence relations. In fact, for such analyses to work in the higher-order case we need to work with a less restricted class of relations than the equivalence relations. To understand why, we must consider a new construction on relations which may be viewed as the function type analogue of the product construction defined in Subsection 2.4.2. For projections there does not seem to be a natural construction for function types.

Let D and E be domains. Let $P : D \leftrightarrow D$ and let $Q : E \leftrightarrow E$. The relation $(P \Rightarrow Q) : [D \to E] \leftrightarrow [D \to E]$ is defined thus:

$$f\ (P \Rightarrow Q)\ f' \iff \forall d, d' \in D.\, d\ P\ d' \Rightarrow f(d)\ Q\ f'(d').$$

For a binary relation $P : D \leftrightarrow D$, we write $|P|$ to mean the set $\{d \mid d\ P\ d\}$ and we write $d : P$ to mean $d \in |P|$. This notation is actually consistent with that introduced in Section 3.1 because now

$$\begin{array}{ll} f : P \Rightarrow Q & \\ \quad \iff & f \in |P \Rightarrow Q| \\ \quad \iff & f\ (P \Rightarrow Q)\ f \\ \quad \iff & \forall d, d' \in D.\, d\ P\ d' \Rightarrow f(d)\ Q\ f(d'). \end{array}$$

A useful result concerning the set $|P|$ for binary relation P is the following:

Lemma 3.2.2 *Let $\{P_i\}_{i \in I}$ be a family of binary relations on D. Then*

$$|\bigwedge_{i \in I} P_i| = \bigcap_{i \in I} |P_i|.$$

Proof Trivial. □

We have seen that binding time analysis is based on properties which may be described at the base types using the equivalence relations All and Id. However, even if P and Q are equivalence relations, $P \Rightarrow Q$ may not be. As a simple example of this, consider the relation $All_D \Rightarrow Id_E$. Now $f\ (All_D \Rightarrow Id_E)\ f'$ iff $\forall d, d'.\, f(d) = f'(d')$. Clearly this entails that $f = f'$, but it also entails that f is constant. Thus assuming that neither D nor E are trivial, so that non-constant functions exist, $All_D \Rightarrow Id_E$ is not reflexive. A class of relations which includes the equivalence relations as a special case and *is* closed under the $\Rightarrow$ construction (and has many other useful closure properties besides) is obtained simply by dropping the requirement of reflexivity:

Definition 3.2.3 *Let D be a set. A binary relation $P : D \leftrightarrow D$ is a* partial equivalence relation (per) *if*

1. *P is symmetric: $d\ P\ d' \Rightarrow d'\ P\ d$;*

2. *P is transitive: $d\ P\ d'\ P\ d'' \Rightarrow d\ P\ d''$.*

If P is a per then $x\ P\ x'$ implies that $x'\ P\ x$ (by symmetry) and hence that $x\ P\ x$ and $x'\ P\ x'$ (both by transitivity). It follows that the domain and range of a per

P are both equal to $|P|$. Furthermore, the restriction of P to $|P|$ is an equivalence relation (a total one).

3.2.1 Complete Pers

We will actually be considering pers over domains and it is natural to consider restricted classes of pers which take account of the domain structure. In particular, for any domain D, we will be interested in those pers $P : D \leftrightarrow D$ which are both:

1. *strict*: $\perp \ P \ \perp$;

2. *inductive*: whenever $\{d_n\} \subseteq D$ and $\{d'_n\} \subseteq D$ are ω-chains such that $d_n \ P \ d'_n$ for all n, then $\bigsqcup\{d_n\} \ P \ \bigsqcup\{d'_n\}$.

Such pers are said to be *complete* by [AP90] (but note that [Ama91] uses the term *complete* where we have used *inductive*). For any domain D, the equivalence relations All_D and Id_D are clearly complete pers on D. More generally, given any continuous function f, the equivalence relation $\ker f$ is a complete per: it is strict because it is reflexive and it is inductive because f is continuous.

Remark In [AP90] a per P is said to be *meet closed* if the following holds: for all non-empty families $X = \{x_i\}_{i\in I}, X' = \{x'_i\}_{i\in I}$, if $x_i \ P \ x'_i$ for all $i \in I$, then $\bigsqcap X \ P \ \bigsqcap X'$. It is well known ([GHK+80]) that projections preserve meets and it follows that for any projection α, the equivalence relation $\ker\alpha$ is meet closed. The results of [AP90] are sufficient to establish that all the pers arising in the example analyses considered in this thesis (Chapter 5) are meet closed, though as yet we have found no way of exploiting this fact.

The following proposition sums up some other basic facts about complete pers.

Proposition 3.2.4 *Let D and E be domains. Let P and P' be complete pers on D, let $\{P_i\}_{i\in I}$ be a family of complete pers on D and let Q and Q' be complete pers on E. Then the following hold:*

1. *$P \Rightarrow Q$ is a complete per on $[D \to E]$;*

2. *$P \Rightarrow All_E = All_{[D\to E]}$;*

3. *$Id_D \Rightarrow Id_E = Id_{[D\to E]}$;*

4. *$P \times Q$ is a complete per on $D \times E$;*

5. *$\bigwedge_{i \in I} P_i$ is a complete per on D;*

6. *the map $\_ \Rightarrow \_ : \mathrm{CPER}(D) \times \mathrm{CPER}(E) \to \mathrm{CPER}([D \to E])$ is monotone in its second argument and anti-monotone in its first:*

$$P' \leq P \text{ and } Q \leq Q' \Rightarrow (P \Rightarrow Q) \leq (P' \Rightarrow Q').$$

Proof These are all well known (see, for example, [Ama91, Asp90]) and easy to verify. We prove 1 and 5 by way of illustration.

For 1, suppose $d\ P\ d'$. Then $\bot_{[D \to E]}(d) = \bot_{[D \to E]}(d') = \bot_E$ and so, since Q is strict, $\bot_{[D \to E]}(d)\ Q\ \bot_{[D \to E]}(d')$. Thus $\bot_{[D \to E]}\ (P \Rightarrow Q)\ \bot_{[D \to E]}$. Now assume that $\{f_n\}$ and $\{f'_n\}$ are ω-chains such that $f_n\ (P \Rightarrow Q)\ f'_n$ for all n. Suppose that $d\ P\ d'$. Then $f_n(d)\ Q\ f'_n(d')$, for each n. Hence $(\bigsqcup\{f_n\})(d) = \bigsqcup\{f_n(d)\}\ Q\ \bigsqcup\{f'_n(d')\} = (\bigsqcup\{f'_n\})(d')$, by inductiveness of Q and continuity of the functions. Thus $\bigsqcup\{f_n\}\ (P \Rightarrow Q)\ \bigsqcup\{f'_n\}$.

For 5, firstly $\bot\ P_i\ \bot$ for all $i \in I$, hence $\bot\ (\bigwedge_{i \in I} P_i)\ \bot$. Now let $\{d_n\}$ and $\{d'_n\}$ be ω-chains such that $d_n\ (\bigwedge_{i \in I} P_i)\ d'_n$ for all n. Then for all $i \in I$, $d_n\ P_i\ d'_n$ for all n and hence $\bigsqcup\{d_n\}\ P_i\ \bigsqcup\{d'_n\}$. Thus $\bigsqcup\{d_n\}\ (\bigwedge_{i \in I} P_i)\ \bigsqcup\{d'_n\}$. □

Closure under conjunction (part 5 of the above proposition) implies that for any given domain D the set of all complete pers on D forms a complete meet semi-lattice (hence a complete lattice) with partial order ($\leq$) and meets ($\bigwedge$) inherited from $\mathcal{R}(D, D)$ (see Subsection 2.4.2). We denote this lattice by $\mathrm{CPER}(D)$. The least and greatest elements of $\mathrm{CPER}(D)$ are Bot_D and All_D respectively, where Bot_D is defined to be the per such that

$$d\ Bot_D\ d' \iff d = d' = \bot.$$

As for All and Id, we will write Bot_σ to mean $Bot_{D^{\mathbf{S}}_\sigma}$.

Our aim is to construct a framework for defining program analyses based on complete pers, in a way which is analogous to the use of Scott-closed sets described in Section 2.4. As a clue to how this might be done, we observe that binary logical relations and pers are intimately related via the $\times$ and $\Rightarrow$ constructions on relations. Looking back to the definition in Chapter 2, we can see that a binary logical relation is a family $\{R_\sigma\}$ such that $R_{\sigma \times \tau} = R_\sigma \times R_\tau$ and $R_{\sigma \to \tau} = R_\sigma \Rightarrow R_\tau$. Now consider a binary logical relation $R : \mathbf{S} \leftrightarrow \mathbf{S}$ relating the standard interpretation to *itself*. It follows from Proposition 3.2.4 that if each base type member of R is a (complete) per, then every member of R is a (complete) per. However, we need rather more than this. We wish to reason about finite interpretations in which each point in the finite lattice interpretation of a type corresponds to a per over the standard domain interpretation. To do this we have to use a more general notion of logical relation.

3.3 Ternary Logical Relations

The Binary Logical Relations Theorem (2.4.4) is actually an instance of a more general theorem ([Plo80]) which concerns relations between κ interpretations, where κ is any ordinal. Although the general theorem is straightforward to state and prove, it is rather cumbersome notationally. The only other instance which we need to consider is the ternary case, which we now describe.

Definition 3.3.1 (Ternary Logical Relation) *Let I, I' and J be interpretations. A ternary relation R between I, I' and J, written $R : I \times I' \leftrightarrow J$, is a type-indexed family of relations $\{R_\sigma\}_{\sigma \in \mathcal{T}}$ where $R_\sigma : (D^I_\sigma \times D^{I'}_\sigma) \leftrightarrow D^J_\sigma$. Such an R is* logical *if for all $\sigma, \tau \in \mathcal{T}$:*

1. $(f, f')\ R_{\sigma \to \tau}\ h$
   $\iff \quad \forall d \in D^I_\sigma, d' \in D^{I'}_\sigma, a \in D^J_\sigma. (d, d')\ R_\sigma\ a \Rightarrow (f\ d, f'\ d')\ R_\tau\ (h\ a)$;

2. $((d_1, d_2), (d'_1, d'_2))\ R_{\sigma_1 \times \sigma_2}\ (a_1, a_2) \iff (d_1, d'_1)\ R_{\sigma_1}\ a_1$ *and* $(d_2, d'_2)\ R_{\sigma_2}\ a_2$.

As in the binary case, we extend $R : I \times I' \leftrightarrow J$ to environments. For $\rho \in Env^I, \rho' \in Env^{I'}$ and $\delta \in Env^J$ we write $(\rho, \rho')\ R\ \delta$ to mean that $dom(\rho) = dom(\rho') = dom(\delta)$ and $(\rho(x), \rho'(x))\ R_\sigma\ \delta(x)$ for all $x \in dom(\rho) \cap Var_\sigma$, for all σ.

Remark Although $R : I \times I' \leftrightarrow J$ is essentially a ternary relation, by grouping the first two arguments together we have reduced it to a binary one. In some ways it would have been better to work with genuine ternary relations, say $R : I \leftrightarrow I' \leftrightarrow J$. An advantage would be that the appropriate definitions of $\times$ and $\Rightarrow$ for ternary relations would have allowed us to define a logical relation as one for which $R_{\sigma \times \tau} = R_\sigma \times R_\tau$ and $R_{\sigma \to \tau} = R_\sigma \Rightarrow R_\tau$. But in practice we only have need for the restricted case in which $I = I'$, and we will be thinking of R as describing how each $a \in J$ corresponds to a set of pairs (in fact, to a complete per). The choice we have made is thus quite suggestive and (we hope) helps give a clearer picture of how we actually exploit the extra expressive power which ternary logical relations give us over binary ones.

Theorem 3.3.2 (The Ternary Logical Relations Theorem) *Let I, I' and J be interpretations and let $R : I \times I' \leftrightarrow J$ be a logical relation. Suppose that $(c^I, c^{I'}) R_\tau c^J$ for each τ and for each $c : \tau$. Then for all σ, for all $e : \sigma$, for all $\rho \in Env^I$, $\rho' \in Env^{I'}$ and $\delta \in Env^J$:*

$$(\rho, \rho')\ R\ \delta \Rightarrow (\llbracket e \rrbracket^I \rho, \llbracket e \rrbracket^{I'} \rho')\ R_\sigma\ (\llbracket e \rrbracket^J \delta).$$

Proof By induction on the structure of e. The base cases (variables and constants) are immediate from the hypothesis of the Theorem. The remaining cases are as follows, where it is assumed that $(\rho, \rho')\ R\ \delta$:

1. $e = \lambda\, x\, .\, e'$, with $x : \sigma$ and $e' : \tau$. Let $d \in D^I_\sigma$, $d' \in D^{I'}_\sigma$ and $a \in D^J_\sigma$ be such that $(d, d')\ R_\sigma\ a$. Then $(\rho[x \mapsto d], \rho'[x \mapsto d'])\ R\ \delta[x \mapsto a]$, so by induction hypothesis:

$$(\llbracket e' \rrbracket^I \rho[x \mapsto d], \llbracket e' \rrbracket^{I'} \rho'[x \mapsto d'])\ R_\tau\ \llbracket e' \rrbracket^J \delta[x \mapsto a].$$

   But by the definition of $\llbracket \_ \rrbracket^I$ we have the equalities:

   - $\llbracket \lambda\, x\, .\, e' \rrbracket^I \rho\ d = \llbracket e' \rrbracket^I \rho[x \mapsto d]$,
   - $\llbracket \lambda\, x\, .\, e' \rrbracket^{I'} \rho'\ d' = \llbracket e' \rrbracket^{I'} \rho'[x \mapsto d']$,
   - $\llbracket \lambda\, x\, .\, e' \rrbracket^J \delta\ a = \llbracket e' \rrbracket^J \delta[x \mapsto a]$.

   Hence $(\llbracket \lambda\, x\, .\, e' \rrbracket^I \rho, \llbracket \lambda\, x\, .\, e' \rrbracket^{I'} \rho')\ R_{\sigma \to \tau}\ \llbracket \lambda\, x\, .\, e' \rrbracket^J \delta$.

2. $e = e_1\ e_2$, with $e_1 : \sigma \to \tau$ and $e_2 : \sigma$. By induction hypothesis we have that $(\llbracket e_1 \rrbracket^I \rho, \llbracket e_1 \rrbracket^{I'} \rho')\ R_{\sigma \to \tau}\ \llbracket e_1 \rrbracket^J \delta$ and similarly for e_2. So, since R is logical,

$$((\llbracket e_1 \rrbracket^I \rho)(\llbracket e_2 \rrbracket^I \rho), (\llbracket e_1 \rrbracket^{I'} \rho')(\llbracket e_2 \rrbracket^{I'} \rho'))\ R_\tau\ (\llbracket e_1 \rrbracket^J \delta)(\llbracket e_2 \rrbracket^J \delta).$$

   But, by the definition of $\llbracket \_ \rrbracket^I$ we have:

   - $\llbracket e_1\ e_2 \rrbracket^I \rho = (\llbracket e_1 \rrbracket^I \rho)(\llbracket e_2 \rrbracket^I \rho)$,
   - $\llbracket e_1\ e_2 \rrbracket^{I'} \rho' = (\llbracket e_1 \rrbracket^{I'} \rho')(\llbracket e_2 \rrbracket^{I'} \rho')$,
   - $\llbracket e_1\ e_2 \rrbracket^J \delta = (\llbracket e_1 \rrbracket^J \delta)(\llbracket e_2 \rrbracket^J \delta)$.

   Hence $(\llbracket e_1\ e_2 \rrbracket^I \rho, \llbracket e_1\ e_2 \rrbracket^{I'} \rho')\ R_\tau\ \llbracket e_1\ e_2 \rrbracket^J \delta$.

3. $e = (e_1, e_2)$. Since (e_1, e_2) is interpreted as a pair and logical relations are defined elementwise on products, this follows easily from the induction hypothesis.

4. $e = \mathbf{fst}(e')$ or $e = \mathbf{snd}(e')$ with $e' : \sigma_1 \times \sigma_2$. See previous case.

$\square$

3.4 Logical Concretisation Maps

We could have given an even more restricted version of Theorem 3.3.2 since, as remarked above, we only actually need it in the case that $I = I'$. We will be considering finite interpretations J related to the standard interpretation by logical relations R^J: $\mathbf{S}^2 \leftrightarrow J$. We extend the term *abstract interpretation* to include the case that R^J is a ternary logical relation of this form. Our intention is to associate with each such logical relation the family of concretisation maps $\gamma^J = \left\{\gamma^J_\sigma\right\}$ with each $\gamma^J_\sigma : D^J_\sigma \to \textsc{cper}(D^{\mathrm{s}}_\sigma)$, such that $(\gamma^J_\sigma\ a)$ is the relation which relates d to d' exactly when R^J_σ relates (d, d') to a:

$$d\ (\gamma^J_\sigma\ a)\ d' \iff (d, d')\ R^J_\sigma\ a \tag{3.4.1}$$

Certainly, this will not be well defined (the $(\gamma^J_\sigma\ a)$ will not be complete pers) for arbitrary R^J. For now, let us consider the general case of interpretations I and J and relation $R : I^2 \leftrightarrow J$ to which we associate a family of maps $\gamma = \{\gamma_\sigma\}$ with each $\gamma_\sigma : D^J_\sigma \to \mathcal{R}(D^I_\sigma, D^I_\sigma)$ defined as in (3.4.1). We can see γ simply as a representation of R, exploiting the isomorphism $\mathcal{R}(A \times B, C) \cong C \to \mathcal{R}(A, B)$. It is useful to consider how the property of R being logical manifests itself as a property of γ.

Theorem 3.4.2 (The Logical Concretisation Map Theorem) *The relation $R : I^2 \leftrightarrow J$ is logical if and only if for all σ, τ:*

1. $\gamma_{\sigma\to\tau}\ h = \bigwedge_{a \in D^J_\sigma} \gamma_\sigma\ a \Rightarrow \gamma_\tau(h\ a)$;

2. $\gamma_{\sigma_1\times\sigma_2}(b_1, b_2) = (\gamma_{\sigma_1}\ b_1) \times (\gamma_{\sigma_2}\ b_2)$,

for all $h \in D^J_{\sigma\to\tau}$ and $(b_1, b_2) \in D^J_{\sigma_1\times\sigma_2}$.

Proof We show that 1 and 2 are equivalent to the two respective clauses in Definition 3.3.1. Firstly, note that to say two relations P and Q are equal, is to say that $f\ P\ f' \iff f\ Q\ f'$. By the definitions,

$$f\ (\gamma_{\sigma\to\tau}\ h)\ f' \iff (f, f')\ R_{\sigma\to\tau}\ h$$

and

$$f\ (\bigwedge_{a \in D^J_\sigma} \gamma_\sigma\ a \Rightarrow \gamma_\tau(h\ a))\ f' \iff \forall a \in D^J_\sigma .\, f\ (\gamma_\sigma\ a \Rightarrow \gamma_\tau(h\ a))\ f'.$$

Now

$$f\ (\gamma_\sigma\ a \Rightarrow \gamma_\tau(h\ a))\ f' \iff \forall d, d' \in D^I_\sigma.\, d\ (\gamma_\sigma\ a)\ d' \Rightarrow f\ d\ (\gamma_\tau(h\ a))\ f'\ d'.$$

But by definition $d\ (\gamma_\sigma\ a)\ d'$ iff $(d, d')\ R_\sigma\ a$ and $f\ d\ (\gamma_\tau(h\ a))\ f'\ d'$ iff $(f\ d, f'\ d')\ R_\tau$ $(h\ a)$. Hence 1 is equivalent to the statement that

$$(f, f')\ R_{\sigma\to\tau}\ h \iff \forall a \in D^J_\sigma.\forall d, d' \in D^I_\sigma.\,(d, d')\ R_\sigma\ a \Rightarrow (f\ d, f'\ d')\ R_\tau\ (h\ a).$$

This is just a re-statement of the first part of Definition 3.3.1 specialised to the case that $I = I'$. The argument for part 2 is similar. □

We now have an alternative way to define a logical relation $R : I^2 \leftrightarrow J$. At the base types we define maps $\gamma_\iota : D^J_\iota \to \mathcal{R}(D^I_\iota, D^I_\iota)$ which are then extended to a family $\gamma = \{\gamma_\sigma\}$ by using clauses 1 and 2 of the Logical Concretisation Map Theorem as definitions at the higher types. We will refer to such a family as a *logical concretisation map*. The associated logical relation R is obtained by taking $(d, d')\ R_\sigma\ a \iff d\ (\gamma_\sigma\ a)\ d'$ as a definition of each R_σ.

Proposition 3.4.3 *Suppose that $R : I^2 \leftrightarrow J$ is logical and that $(\gamma_\iota\ a) \in \text{CPER}(D^I_\iota)$ for each $\iota \in \mathcal{T}_0$, $a \in D^J_\iota$. Then $(\gamma_\sigma\ b) \in \text{CPER}(D^I_\sigma)$ for each $\sigma \in \mathcal{T}$, $b \in D^J_\sigma$.*

Proof By induction on σ. The base cases ($\sigma \in \mathcal{T}_0$) are given by the hypothesis of the Proposition. For function types, let $h \in [D^J_\sigma \to D^J_\tau]$. By the Logical Concretisation Map Theorem, we have $\gamma_{\sigma\to\tau}\ h = \bigwedge_{a\in D^J_\sigma} \gamma_\sigma\ a \Rightarrow \gamma_\tau(h\ a)$. By induction hypothesis, $\gamma_\sigma\ a$ and $\gamma_\tau(h\ a)$ are complete pers for all $a \in D^J_\sigma$. Thus, by Proposition 3.2.4 part 1, $\gamma_\sigma\ a \Rightarrow \gamma_\tau(h\ a)$ is a complete per for all $a \in D^J_\sigma$, hence $\bigwedge_{a\in D^J_\sigma} \gamma_\sigma\ a \Rightarrow \gamma_\tau(h\ a)$ is a complete per by Proposition 3.2.4 part 5. The case for product types follows immediately from the induction hypothesis by Proposition 3.2.4 part 4. □

3.4.1 Co-Step Functions as Basic Properties

The Logical Concretisation Map Theorem gives an interesting insight into the way in which we will be using abstract function spaces to describe properties over their concrete counterparts. It seems reasonable to consider pers of the form $P \Rightarrow Q$ as being the 'basic' properties for function types. If $R : I^2 \leftrightarrow J$ is logical, the Logical Concretisation Map Theorem shows that the property (per) represented via concretisation by an abstract function, is a *conjunction* of such basic properties. The following definition and proposition show that corresponding to these basic properties there are 'basic' functions in J.

Definition 3.4.4 *Let A and B be complete lattices and let $a \in A$ and $b \in B$.*

1. *The* step function $\lceil a, b \rceil : A \to B$ *is defined by*

$$\lceil a, b \rceil (a') = \begin{cases} b & \text{if } a' \sqsupseteq a \\ \bot_B & \text{otherwise.} \end{cases}$$

2. *The* co-step function $\lfloor a, b \rfloor : A \to B$ *is defined dually by*

$$\lfloor a, b \rfloor (a') = \begin{cases} b & \text{if } a' \sqsubseteq a \\ \top_B & \text{otherwise.} \end{cases}$$

We shall really only be interested in the co-step functions for the moment, but in Chapter 8 we find a practical use for both the step and the co-step functions (we will also see that there are sound lattice theoretic reasons for considering such functions to be 'basic').

Proposition 3.4.5 *Suppose $R : I^2 \leftrightarrow J$ is a logical relation such that each associated γ_σ is top preserving and monotone. Let $\sigma, \tau \in \mathcal{T}$. Then for any $a \in D^J_\sigma$ and $b \in D^J_\tau$*

$$\gamma_{\sigma\to\tau}(\lfloor a, b \rfloor) = \gamma_\sigma\ a \Rightarrow \gamma_\tau\ b.$$

Proof By the Logical Concretisation Map Theorem we know that

$$\gamma_{\sigma\to\tau}(\lfloor a, b \rfloor) = \bigwedge_{a' \in D^J_\sigma} \gamma_\sigma\ a' \Rightarrow \gamma_\tau(\lfloor a, b \rfloor (a')).$$

The required result follows by showing that $(\gamma_\sigma\ a \Rightarrow \gamma_\tau\ b)$ is the smallest member of the family $\{\gamma_\sigma\ a' \Rightarrow \gamma_\tau(\lfloor a, b \rfloor (a'))\}_{a' \in D^J_\sigma}$. Firstly it is in the family, since $a \in D^J_\sigma$ and $\lfloor a, b \rfloor (a) = b$. For every other $(\gamma_\sigma\ a' \Rightarrow \gamma_\tau\ b')$ in the family there are two cases. In the first case $a' \sqsubseteq a$, and then $\gamma_\sigma\ a' \leq \gamma_\sigma\ a$ (γ_σ monotone) and $b' = b$ (definition of $\lfloor a, b \rfloor$) hence by part 6 of Proposition 3.2.4, $(\gamma_\sigma\ a \Rightarrow \gamma_\tau\ b) \leq (\gamma_\sigma\ a' \Rightarrow \gamma_\tau\ b')$; in the second case $a' \not\sqsubseteq a$, in which case $b' = \top^J_\tau$ and so $(\gamma_\sigma\ a' \Rightarrow \gamma_\tau\ b') = (\gamma_\sigma\ a' \Rightarrow All_{D^I_\tau}) = All_{D^I_{\sigma\to\tau}}$, by Proposition 3.2.4, part 2. □

Thus each co-step function corresponds to a property of the form $P \Rightarrow Q$. Furthermore, it is easily seen that each abstract function $h \in [D^J_\sigma \to D^J_\tau]$ can be expressed as a meet of co-step functions:

$$h = \bigsqcap_{a \in D^J_\sigma} \lfloor a, h\ a \rfloor .$$

This may suggest viewing an abstract interpretation as a form of *program logic*, with the co-step functions corresponding to formulae of the form $\phi \to \psi$ and meets in the abstract lattices corresponding to conjunction of formulae, which is the idea explored in [Jen91]. Although [Jen91] is concerned with strictness analysis (and implicitly with Scott-closed sets) the approach should extend to analyses based on the use of pers.

3.5 Inherited Properties

Proposition 3.4.3 is an example of what [Abr90] calls inheritance. A property $P = \{P_\sigma\}$ of ternary relations is *inherited* if for each logical relation $R : I \times I' \leftrightarrow J$:

$$(\forall \iota \in \mathcal{T}_0.\, P_\iota(R_\iota)) \Rightarrow \forall \sigma \in \mathcal{T}.\, P_\sigma(R_\sigma).$$

If $P_\sigma(R_\sigma)$ for each σ, we say that P holds of R. The property in question in Proposition 3.4.3 is that of $(\gamma_\sigma\ a)$ being a complete per for each $a \in D^J_\sigma$. From now on we will refer to this as property CP.

The following definition gathers together some other properties of interest.

Definition 3.5.1 *For a ternary relation $R : I^2 \leftrightarrow J$, each R_σ may be:*

1. $\perp$-reflecting*: $(d, d')\ R_\sigma\ \perp^J_\sigma \Rightarrow d = d' = \perp^I_\sigma$;*

2. $\top$-universal*: D^J_σ has greatest element $\top^J_\sigma$ and $\forall d, d' \in D^I_\sigma.\, (d, d')\ R_\sigma\ \top^J_\sigma$;*

3. strict*: $(\perp^I_\sigma, \perp^I_\sigma)\ R_\sigma\ \perp^J_\sigma$;*

4. inductive*: whenever $\{d_n\}, \{d'_n\}$ and $\{a_n\}$ are ω-chains such that $(d_n, d'_n)\ R_\sigma\ a_n$ for all n, then $(\bigsqcup \{d_n\}, \bigsqcup \{d'_n\})\ R_\sigma\ \bigsqcup \{a_n\}$;*

5. right-monotone*: $(d, d')\ R_\sigma\ a \sqsubseteq b \Rightarrow (d, d')\ R_\sigma\ b$;*

6. left-$\perp$-universal*: $\forall a \in D^J_\sigma.\, (\perp^I_\sigma, \perp^I_\sigma)\ R_\sigma\ a$;*

7. locally inductive*: whenever ω-chains $\{d_n\}$ and $\{d'_n\}$ and $a \in D^J_\sigma$ are such that $(d_n, d'_n)\ R_\sigma\ a$ for all n, then $(\bigsqcup \{d_n\}, \bigsqcup \{d'_n\})\ R_\sigma\ a$;*

8. locally symmetric*: $(d, d')\ R_\sigma\ a \Rightarrow (d', d)\ R_\sigma\ a$;*

9. locally transitive*: $(d, d')\ R_\sigma\ a$ and $(d', d'')\ R_\sigma\ a \Rightarrow (d, d'')\ R_\sigma\ a$.*

Some of these properties of R_σ, or combinations of them, are equivalent to familiar properties of γ_σ. We list these equivalences below.

- R_σ is strict and $\perp$-reflecting iff γ_σ is strict (i.e., $\gamma_\sigma\ \perp^J_\sigma = Bot_{D^I_\sigma}$, recalling that $Bot_{D^I_\sigma}$ is the least complete per on D^I_σ);
- R_σ is $\top$-universal iff γ_σ is top preserving (i.e., $\gamma_\sigma\ \top^J_\sigma = All_{D^I_\sigma}$);
- R_σ is right-monotone iff γ_σ is monotone;
- the conjunction of properties 6–9 is just property CP.

The following lemmas state which properties are inherited and describe the key inter-relations between the different properties.

Lemma 3.5.2 *The following hold for properties 1–9:*

- *properties 2–8 are independently inherited;*
- *$\perp$-reflectivity (1) is inherited in the presence of $\top$-universality (2);*
- *local transitivity (9) is inherited in the presence of local symmetry (8);*

Proof The proofs are all straightforward inductions over $\mathcal{T}$. That $\top$-universality, strictness and inductiveness are inherited and that $\perp$-reflectivity is inherited in the presence of $\top$-universality, are the obvious generalisations to the ternary case of the corresponding parts of [Abr90]'s Proposition 3.4. For the purposes of illustration we prove that right-monotonicity is inherited. Let $R : I^2 \leftrightarrow J$ be a logical relation. The base cases are given.

For the case of function types, suppose that $(f, f')\ R_{\sigma\to\tau}\ h \sqsubseteq g$. Since R is logical, $(d, d')\ R_\sigma\ a \Rightarrow (f\ d, f'\ d')\ R_\tau\ (h\ a)$. But $h \sqsubseteq g \Rightarrow h\ a \sqsubseteq g\ a$, so by the induction hypothesis, $(d, d')\ R_\sigma\ a \Rightarrow (f\ d, f'\ d')\ R_\tau\ (g\ a)$. Hence, again since R is logical, $(f, f')\ R_{\sigma\to\tau}\ g$.

For the case of product types, suppose $((d_1, d_2), (d_1', d_2'))\ R_{\sigma\times\tau}\ (a_1, a_2) \sqsubseteq (b_1, b_2)$. Then since R is logical, $(d_1, d_1')\ R_\sigma\ a_1 \sqsubseteq b_1$ and $(d_2, d_2')\ R_\tau\ a_2 \sqsubseteq b_2$. By the induction hypothesis, $(d_1, d_1')\ R_\sigma\ b_1$ and $(d_2, d_2')\ R_\tau\ b_2$. Hence, since R is logical, $((d_1, d_2), (d_1', d_2'))\ R_{\sigma\times\tau}\ (b_1, b_2)$. □

Note that Proposition 3.4.3 is a corollary of Lemma 3.5.2.

Lemma 3.5.3 *The following implications hold between properties 1–9:*

- *left-$\perp$-universality (6) $\Rightarrow$ strictness (3);*
- *inductiveness (4) $\Rightarrow$ local inductiveness (7);*
- *if D^J_σ is of finite height then local inductiveness (7) $\Rightarrow$ inductiveness (4).*

Proof The first two are obvious. For the third, assume that D^J_σ is of finite height and that R_σ is locally inductive. Let $\{d_n\}$, $\{d'_n\}$ and $\{a_n\}$ be ω-chains such that $(d_n, d'_n)\ R_\sigma\ a_n$ for all n. Then local inductiveness implies that $(\bigsqcup\{d_n\}, \bigsqcup\{d'_n\})\ R_\sigma\ \bigsqcup\{a_n\}$, since $\{a_n\}$ is eventually constant. $\Box$

Chapter 4

Abstract Interpretation Using Pers

We are now in a position to give the details of how abstract interpretation can be used to reason about function definitions in terms of properties of their denotations which may be expressed using pers. The results of this chapter assume the following:

- $(J, R^J\colon \mathbf{S}^2 \leftrightarrow J)$ is an abstract interpretation;
- R^J satisfies property CP;
- $\gamma^J = \left\{\gamma^J_\sigma\right\}$ is the associated logical concretisation map (with each $\gamma^J_\sigma : D^J_\sigma \to \textsc{cper}(D^{\mathrm{s}}_\sigma)$ defined as in (3.4.1)).

Taking our lead from Abramsky's (binary) logical relations framework, there are two important questions which we might ask given this situation. Firstly, what does it mean for J to be correct? Secondly, given the appropriate notion of correctness, do best interpretations exist for the constants?

4.1 Correctness

Our definition of what it means for J to be correct will be guided by the basic idea that if $f \in [D \to E]$ is the standard denotation of some term, we wish to be able to test for conditions of the form $f : P \Rightarrow Q$, where $P \in \textsc{cper}(D)$ and $Q \in \textsc{cper}(E)$. Suppose that $f^{\mathrm{s}} : D^{\mathrm{s}}_\sigma \to D^{\mathrm{s}}_\tau$ and $f^J : D^J_\sigma \to D^J_\tau$ are the standard and abstract denotations respectively of some closed term $e : \sigma \to \tau$. Each $a \in D^J_\sigma$ and $b \in D^J_\tau$ correspond to complete pers, namely $(\gamma^J_\sigma\ a)$ and $(\gamma^J_\tau\ b)$. By analogy with the strictness analysis described in Section 2.4 we require that

$$f^J\ a = b \Rightarrow f^{\mathrm{s}} : (\gamma^J_\sigma\ a) \Rightarrow (\gamma^J_\tau\ b),$$

giving us a sound and effective test for the condition that $f^{\mathrm{s}} : (\gamma^J_\sigma\ a) \Rightarrow (\gamma^J_\tau\ b)$. The following lemma and corollary show that this idea generalises pleasantly to include both nullary functions and curried functions of more than one argument.

Lemma 4.1.1 *Let $f, f' : D_1 \to \ldots \to D_n \to E$ be continuous functions on domains, where $n \geq 0$ (if $n = 0$ then we just mean $f, f' \in E$). Let P_i be a complete per on D_i, $1 \leq i \leq n$, and let Q be a complete per on E. Then $f\ (P_1 \Rightarrow \ldots \Rightarrow P_n \Rightarrow Q)\ f'$ if and only if*

$$\begin{array}{l} \forall d_1, \ldots, d_n. \forall d'_1, \ldots, d'_n. \\ \qquad (d_1\ P_1\ d'_1)\ \mathit{and} \ldots \mathit{and}\ (d_n\ P_n\ d'_n) \Rightarrow (f\ d_1 \ldots d_n)\ Q\ (f'\ d'_1 \ldots d'_n), \end{array}$$

where borrowing the types convention, we write $P_1 \Rightarrow \ldots \Rightarrow P_n \Rightarrow Q$ to mean $P_1 \Rightarrow (\ldots (P_n \Rightarrow Q) \ldots)$.

Proof By induction on n. For $n = 0$, the two sides of the claimed equivalence are both just $f\ Q\ f'$. For the inductive case, by definition $f\ (P_1 \Rightarrow P_2 \Rightarrow \ldots \Rightarrow P_n \Rightarrow Q)$ f' iff

$$\forall d_1. \forall d'_1. d_1\ P_1\ d'_1 \Rightarrow (f\ d_1)\ (P_2 \Rightarrow \ldots \Rightarrow P_n \Rightarrow Q)\ (f'\ d'_1)$$

But by induction hypothesis, $(f\ d_1)\ (P_2 \Rightarrow \ldots \Rightarrow P_n \Rightarrow Q)\ (f'\ d'_1)$ iff

$$\begin{array}{l} \forall d_2, \ldots, d_n. \forall d'_2, \ldots, d'_n. \\ \qquad (d_2\ P_2\ d'_2)\ \mathit{and} \ldots \mathit{and}\ (d_n\ P_n\ d'_n) \Rightarrow (f\ d_1\ d_2 \ldots d_n)\ Q\ (f'\ d'_1\ d'_2 \ldots d'_n). \end{array}$$

The rest is just a straightforward logical equivalence. $\square$

Corollary 4.1.2 *Let f be given as in the statement of the Lemma. Then $f : P_1 \Rightarrow \ldots \Rightarrow P_n \Rightarrow Q$ if and only if*

$$\begin{array}{l} \forall d_1, \ldots, d_n. \forall d'_1, \ldots, d'_n. \\ \qquad (d_1\ P_1\ d'_1)\ \mathit{and} \ldots \mathit{and}\ (d_n\ P_n\ d'_n) \Rightarrow (f\ d_1 \ldots d_n)\ Q\ (f\ d'_1 \ldots d'_n). \end{array}$$

Note that when $n = 0$, $f : P_1 \Rightarrow \ldots \Rightarrow P_n \Rightarrow Q$ just asserts that $f : Q$ (equivalently, that $f \in |Q|$ or that $f\ Q\ f$). For $n \geq 2$, if we let $f^* : D_1 \times \ldots \times D_n \to E$ be the 'uncurried' version of f, comparison of the above with the definition of product for relations (Subsection 2.4.2) will reveal that $f : P_1 \Rightarrow \ldots \Rightarrow P_n \Rightarrow Q$ if and only if $f^* : P_1 \times \ldots \times P_n \Rightarrow Q$, as we might hope.

So our correctness requirement should be such that for any closed expression $e : \sigma_1 \rightarrow \ldots \sigma_n \rightarrow \tau$ with standard denotation f^{s} and abstract denotation f^J

$$f^J\ a_1 \ldots a_n = b \Rightarrow f^{\mathrm{s}} : (\gamma^J_{\sigma_1}\ a_1) \Rightarrow \ldots \Rightarrow (\gamma^J_{\sigma_n}\ a_n) \Rightarrow (\gamma^J_\tau\ b),$$

for all $a_1 \in D^J_{\sigma_1}, \ldots, a_n \in D^J_{\sigma_n}$ and $b \in D^J_\tau$. A simple rephrasing of this according to the definitions and Lemma 3.2.2 yields the equivalent

$$f^{\mathrm{s}} : \bigwedge_{a_1 \in D^J_{\sigma_1}} \cdots \bigwedge_{a_n \in D^J_{\sigma_n}} ((\gamma^J_{\sigma_1}\ a_1) \Rightarrow \ldots \Rightarrow (\gamma^J_{\sigma_n}\ a_n) \Rightarrow (\gamma^J_\tau (f^J\ a_1 \ldots a_n))). \qquad (4.1.3)$$

Although it may not be immediately obvious, this is precisely the content of the following formal definition of correctness.

Definition 4.1.4 *The interpretation J is* correct *if for each $\rho \in Env^{\mathrm{s}}$ and $\delta \in Env^J$, for every expression $e : \sigma$*

$$(\rho, \rho)\ R^J\ \delta \Rightarrow [\![e]\!]^{\mathrm{s}}\rho : \gamma^J_\sigma([\![e]\!]^J\delta).$$

Correctness of J implies that $c^{\mathrm{s}} : \gamma^J_\tau\ c^J$ for each $c : \tau$. Accordingly, the interpretation of an individual constant $c : \tau$ is said to be correct *if $c^{\mathrm{s}} : \gamma^J_\tau\ c^J$.*

To see that this really is what we want, we need the following result.

Lemma 4.1.5 *Let D and E be domains. Let $P : D \leftrightarrow D$ and let $\{Q_i\}_{i \in I}$ be a family with each $Q_i : E \leftrightarrow E$. Then*

$$P \Rightarrow (\bigwedge_{i \in I} Q_i) = \bigwedge_{i \in I} (P \Rightarrow Q_i).$$

Proof For relations S and S' recall that $S = S'$ if and only if $S \leq S'$ *and* $S' \leq S$.

To show $P \Rightarrow (\bigwedge_{i \in I} Q_i) \leq \bigwedge_{i \in I}(P \Rightarrow Q_i)$, assume that $f\ (P \Rightarrow (\bigwedge_{i \in I} Q_i))\ f'$. Now let $j \in I$. Suppose $d\ P\ d'$. Then by assumption $(f\ d)\ (\bigwedge_{i \in I} Q_i)\ (f'\ d')$ and so $(f\ d)\ Q_j\ (f'\ d')$. Hence $f\ (P \Rightarrow Q_j)\ f'$. But j was chosen arbitrarily from I, so $\forall i \in I. f\ (P \Rightarrow Q_i)\ f'$.

To show the reverse implication, assume that $f\ (\bigwedge_{i \in I}(P \Rightarrow Q_i))\ f'$. Suppose $d\ P\ d'$. Let $j \in I$. Then by assumption $f\ (P \Rightarrow Q_j)\ f'$ and so $(f\ d)\ Q_j\ (f'\ d')$. But j was arbitrary, so $\forall i \in I. (f\ d)\ Q_i\ (f'\ d')$, i.e., $(f\ d)\ (\bigwedge_{i \in I} Q_i)\ (f'\ d')$. Hence $f\ (P \Rightarrow (\bigwedge_{i \in I} Q_i))\ f'$. $\square$

We then have the following equivalence:

Proposition 4.1.6 *Let* $h : D^J_{\sigma_1} \to \ldots \to D^J_{\sigma_n} \to D^J_\tau$ *be monotonic. Then*

$$\gamma^J_{\sigma_1\to\ldots\to\sigma_n\to\tau} h = \bigwedge_{a_1 \in D^J_{\sigma_1}} \cdots \bigwedge_{a_n \in D^J_{\sigma_n}} (\gamma^J_{\sigma_1}\ a_1) \Rightarrow \ldots (\gamma^J_{\sigma_n}\ a_n) \Rightarrow (\gamma^J_\tau (h\ a_1 \ldots a_n)).$$

Proof By induction on n. For $n = 0$ both sides are just $\gamma^J_\tau h$. Let $n = k+1$. By the Logical Concretisation Map Theorem (3.4.2), $\gamma^J_{\sigma_1\to\ldots\to\sigma_n\to\tau} h = \bigwedge_{a_1 \in D^J_{\sigma_1}} (\gamma^J_{\sigma_1}\ a_1) \Rightarrow (\gamma^J_{\sigma_2\to\ldots\to\sigma_n\to\tau}(h\ a_1))$. Then by induction hypothesis $\gamma^J_{\sigma_2\to\ldots\to\sigma_n\to\tau}(h\ a_1)$ is just

$$\bigwedge_{a_2 \in D^J_{\sigma_2}} \cdots \bigwedge_{a_n \in D^J_{\sigma_n}} (\gamma^J_{\sigma_2}\ a_2) \Rightarrow \ldots (\gamma^J_{\sigma_n}\ a_n) \Rightarrow (\gamma^J_\tau (h\ a_1\ a_2 \ldots a_n)).$$

The required result follows by k applications of Lemma 4.1.5 (induction on k). □

Using this we can see that the requirement expressed by (4.1.3) is precisely what it means for a closed term to be correct according to Definition 4.1.4.

Theorem 4.1.7 (The Correctness Theorem) *If* $c^{\mathrm{S}} : \gamma^J_\tau\ c^J$ *for all types* τ *and for all constants* $c : \tau$*, then* J *is correct.*

Proof By the definition of the concretisation maps, $d : \gamma^J_\sigma\ a$ iff $(d, d)\ R^J_\sigma\ a$. The proof is then just an instantiation of the Ternary Logical Relations Theorem (3.3.2) with $I = I' = \mathbf{S}$ and $\rho = \rho'$. □

4.1.1 Least Fixed Point Interpretations

We saw in Section 2.4 that the interpretation of $\mathbf{Y}_\sigma$ in $\mathbf{B}$ is the least fixed point operator (Figure 2.4), just as it is in the standard interpretation. Interpretations with this property are said by [Abr90] to be *least fixed point* interpretations (in earlier versions of [Abr90] they were termed *normal* interpretations). The following result is a straightforward adaptation of Proposition 3.5 from [Abr90] to the ternary case.

Proposition 4.1.8 *Let* I *and* K *be least fixed point interpretations. Let* $R : I^2 \leftrightarrow K$ *be a logical relation with associated* γ *defined as in (3.4.1). If the logical relation* R *is strict and inductive then*

$$\mathbf{Y}^I_\sigma : \gamma_{(\sigma\to\sigma)\to\sigma}\ \mathbf{Y}^K_\sigma.$$

Proof Let f, f', h be such that $(f, f')\ R_{\sigma\to\sigma}\ h$. We must show that $(\mathbf{Y}^I_\sigma\ f, \mathbf{Y}^I_\sigma\ f')\ R_\sigma$ $(\mathbf{Y}^K_\sigma\ h)$. Since I and K are, by assumption, least fixed point interpretations, this amounts to showing that

$$(\bigsqcup_{i\in\omega} f^i\ \bot^I_\sigma, \bigsqcup_{i\in\omega} f'^i\ \bot^I_\sigma)\ R_\sigma\ \bigsqcup_{i\in\omega} h^i\ \bot^K_\sigma.$$

Since R is assumed to be inductive, it suffices to show that $(f^i\ \bot^I_\sigma, f'^i\ \bot^I_\sigma)\ R_\sigma$ $(h^i\ \bot^K_\sigma)$ for all i. We proceed by induction on i. For $i = 0$ we just require that R is strict, which is true by assumption. For $i = n + 1$, by induction hypothesis $(f^n\ \bot^I_\sigma, f'^n\ \bot^I_\sigma)\ R_\sigma\ (h^n\ \bot^K_\sigma)$. But then since $(f, f')\ R_{\sigma\to\sigma}\ h$ and R is logical, $(f(f^n\ \bot^I_\sigma), f'(f'^n\ \bot^I_\sigma))\ R_\sigma\ h(h^n\ \bot^K_\sigma)$. □

Corollary 4.1.9 *If J is a least fixed point interpretation then $\mathbf{Y}^{\mathrm{S}}_\sigma : \gamma^J_{(\sigma\to\sigma)\to\sigma}\ \mathbf{Y}^J_\sigma$.*

Proof Firstly, note that $\mathbf{S}$ is a least fixed point interpretation. We have assumed that R^J satisfies CP, or equivalently, properties 6–9 of Definition 3.5.1, so in particular R^J is strict and locally inductive. Furthermore, J is assumed to be a finite interpretation and so each D^J_σ is of finite height. Then by Lemma 3.5.3, R^J is strict and inductive, so the Theorem applies. □

4.1.2 Monotone Concretisation Maps

A typical situation which arises (see the examples of the next chapter) when we attempt to derive a correct interpretation for a constant, say $c : \sigma \to \tau$ is the following: for some a, b_1, b_2 we know that for all d, d' such that $d\ (\gamma^J_\sigma\ a)\ d'$, *either* $c^{\mathrm{s}}(d)\ (\gamma^J_\tau\ b_1)\ c^{\mathrm{s}}(d')$, *or* $c^{\mathrm{s}}(d)\ (\gamma^J_\tau\ b_2)\ c^{\mathrm{s}}(d')$. In this situation, if the concretisation map γ^J_τ is monotone, the answer is simple: set $c^J\ a = b_1 \sqcup b_2$. Monotonicity of γ^J_τ then ensures that $\gamma^J_\tau\ b_1 \leq \gamma^J_\tau(b_1 \sqcup b_2)$ and $\gamma^J_\tau\ b_2 \leq \gamma^J_\tau(b_1 \sqcup b_2)$, and hence that $d\ (\gamma^J_\sigma\ a)\ d' \Rightarrow c^{\mathrm{s}}(d)\ (\gamma^J_\tau(b_1 \sqcup b_2))\ c^{\mathrm{s}}(d')$. More generally, monotonicity implies that it is always *safe* to approximate upwards in the abstract lattices, in the sense that if a is a correct interpretation for c and $a \sqsubseteq a'$, then a' must also be correct. This is such a heavily used property when deriving correct interpretations for constants that monotonicity of the concretisation maps is effectively indispensable. All the abstract interpretations we are aware of, and certainly all the examples considered in this thesis, are based on monotone concretisation maps or their equivalent. In Chapter 6, we are forced to reject a possible scheme for the abstract interpretation of recursive types, because the induced concretisation maps are not monotone.

4.2 Best Interpretations for Constants

In Subsection 2.4.4 in Chapter 2, we discussed the existence of best interpretations for the constants in **B** and it is an issue of interest for any abstract interpretation (both [Nie89] and [Abr90] discuss the question at some length as it applies to the particular frameworks they consider). Although the $(\gamma^{\mathbf{B}}_\sigma\ a)$ were sets and our $(\gamma^J_\sigma\ b)$ are pers, the basic idea of best interpretations for constants is the same. Let e be a closed term. Suppose we know that $[\![e]\!]^{\mathrm{S}} : P$ for some per P (this is precisely the situation guaranteed by correctness of J if we take $P = \gamma^J_\sigma([\![e]\!]^J))$. Then the smaller that P is, the greater will be our knowledge of $[\![e]\!]^{\mathrm{S}}$. So if we can choose correct interpretations for the constants in such a way as to minimise the size of $\gamma^J_\sigma([\![e]\!]^J)$ for all $e : \sigma$, we can thus maximise the accuracy of any analysis based on J. If we assume that the γ^J_σ are monotone, this can be achieved by minimising the abstract interpretations of the constants, c^J. Just as in the example of strictness analysis, this will be possible if for each type σ and for each $d \in D^{\mathrm{S}}_\sigma$, there is a least $a \in D^J_\sigma$ such that a correctly describes d, which in the current setting means such that $d : \gamma^J_\sigma\ a$. This motivates the following definition.

Definition 4.2.1 *For a constant $c : \sigma$ we say that c^J is the* best interpretation for *c if c^J is the least value in D^J_σ such that $c^{\mathrm{S}} : \gamma^J_\sigma\ c^J$. More generally, we say that $a \in D^J_\sigma$ is* best for *$d \in D^{\mathrm{S}}_\sigma$ if a is the least value in D^J_σ such that $d : \gamma^J_\sigma\ a$.*

To obtain a maximally accurate analysis it is clearly sufficient that the interpretation of each constant be best. Whether it is also necessary is not so clear. We return to this question in Section 4.3.

We will show that a sufficient condition for the existence of best interpretations for the constants is that each γ^J_σ has a left adjoint.

4.2.1 Left Adjoints for Concretisation Maps

Definition 4.2.2 *Let A and B be complete lattices and let $f : A \to B$ and $g : B \to A$ be monotone maps. Then f and g are said to form an* adjunction *(alternatively, a* Galois connection*) with left (or lower) component f and right (or upper) component g if for all $a \in A$ and $b \in B$:*

$$f(a) \sqsubseteq b \iff a \sqsubseteq g(b).$$

In that case, g is said to have left adjoint *f and f is said to have* right adjoint *g. The statement that f and g form such an adjunction is abbreviated as $f \dashv g$.*

Proposition 4.2.3 *Let A and B be complete lattices and let $f : A \to B$ and $g : B \to A$ be monotone maps. Then the following are equivalent:*

1. $f \dashv g$
2. $f \circ g \sqsubseteq id_B$ *and* $g \circ f \sqsupseteq id_A$
3. f *preserves joins and* $g(b) = \bigsqcup \{a \in A \mid f(a) \sqsubseteq b\}$
4. g *preserves meets and* $f(a) = \bigsqcap \{b \in B \mid a \sqsubseteq g(b)\}$

Proof These are all standard ([GHK+80]). □

Corollary 4.2.4 *The components of an adjunction between complete lattices determine each other uniquely (parts 3 and 4 of the Proposition).*

The corollary to the next result shows that the existence of left adjoints is easily established since it is an inherited property.

Proposition 4.2.5 *If γ^J_ι preserves meets for all $\iota \in \mathcal{T}_0$, then γ^J_σ preserves meets for all $\sigma \in \mathcal{T}$.*

Proof By induction on types. The base cases are given. Product types are simple since meets are calculated elementwise for products. For function types $\sigma \to \tau$, let $\{h_i\}_{i \in I}$ be a family in $D^J_{\sigma\to\tau}$. Then

$$\begin{array}{rcll}
\bigwedge_{i\in I} \gamma^J_{\sigma\to\tau}\ h_i & = & \bigwedge_{i\in I} \bigwedge_{a\in D^J_\sigma} (\gamma^J_\sigma\ a \Rightarrow \gamma^J_\tau\ (h_i\ a)) & \text{Theorem 3.4.2} \\
 & = & \bigwedge_{a\in D^J_\sigma} \bigwedge_{i\in I} (\gamma^J_\sigma\ a \Rightarrow \gamma^J_\tau\ (h_i\ a)) & \\
 & = & \bigwedge_{a\in D^J_\sigma} (\gamma^J_\sigma\ a \Rightarrow \bigwedge_{i\in I} \gamma^J_\tau\ (h_i\ a)) & \text{Lemma 4.1.5} \\
 & = & \bigwedge_{a\in D^J_\sigma} (\gamma^J_\sigma\ a \Rightarrow \gamma^J_\tau\ (\bigsqcap_{i\in I} (h_i\ a))) & \text{by induction hypothesis} \\
 & = & \bigwedge_{a\in D^J_\sigma} (\gamma^J_\sigma\ a \Rightarrow \gamma^J_\tau\ ((\bigsqcap_{i\in I} h_i)\ a)) & \\
 & = & \gamma^J_{\sigma\to\tau} (\bigsqcap_{i\in I} h_i) & \text{Theorem 3.4.2}
\end{array}$$

Note that the penultimate step of the proof uses the equality $\bigsqcap_{i\in I}(h_i\ a) = (\bigsqcap_{i\in I} h_i)\ a$, which holds because J is finite. More generally it holds for meets of finite families in continuous function spaces over Scott domains, and for meets of arbitrary families in monotone function spaces over complete lattices. □

Corollary 4.2.6 *If γ^J_ι preserves meets for all $\iota \in \mathcal{T}_0$, then γ^J_σ has a left adjoint for all $\sigma \in \mathcal{T}$.*

Proof Immediate from the Proposition by Proposition 4.2.3 part 4. □

4.2.2 Abstraction Maps

Suppose that γ^J_σ is monotone and has left adjoint α^J_σ. Define the 'forgetful' map $U : \text{CPER}(D^{\mathrm{s}}_\sigma) \to \wp(D^{\mathrm{s}}_\sigma \times D^{\mathrm{s}}_\sigma)$ to take a complete per to its graph and define the 'free complete per' map $F : \wp(D^{\mathrm{s}}_\sigma \times D^{\mathrm{s}}_\sigma) \to \text{CPER}(D^{\mathrm{s}}_\sigma)$ by

$$F(S) = \bigwedge \{Q \in \text{CPER}(D^{\mathrm{s}}_\sigma) \mid \forall (d, d') \in S.\, d\ Q\ d'\}\,.$$

Thus, given the graph of an arbitrary binary relation on D^{s}_σ, the map F returns the smallest complete per containing that relation. It is clear that $F \circ U = id$ and $U \circ F \geq id$, hence $F \dashv U$, by Proposition 4.2.3 (note that $F \circ U$ is actually *equal* to the identity, not just dominated by it).

Remark $U(P)$ is really forgetting not that P is a *relation* but that P is a *complete partial equivalence* relation. It is slightly more convenient for our purposes to define U as we have done, rather than as a map $\text{CPER}(D^{\mathrm{s}}_\sigma) \to \mathcal{R}(D^{\mathrm{s}}_\sigma, D^{\mathrm{s}}_\sigma)$.

Now adjunctions compose, meaning that

$$(F \dashv U \textit{ and } \alpha^J_\sigma \dashv \gamma^J_\sigma) \Rightarrow \alpha^J_\sigma \circ F \dashv U \circ \gamma^J_\sigma,$$

hence $\alpha^J_\sigma(F(S)) \sqsubseteq a \iff S \subseteq U(\gamma^J_\sigma\ a)$. So, since $d : \gamma^J_\sigma\ a \iff d\ (\gamma^J_\sigma\ a)\ d$ and $d\ (\gamma^J_\sigma\ a)\ d \iff \{(d, d)\} \subseteq U(\gamma^J_\sigma\ a)$, we have

$$d : \gamma^J_\sigma\ a \iff \alpha^J_\sigma(F\,\{(d, d)\}) \sqsubseteq a.$$

But this says precisely that $\alpha^J_\sigma(F\,\{(d, d)\})$ is best for d.

Definition 4.2.7 *If each γ^J_σ has left adjoint α^J_σ, then define the family of* abstraction *maps $abs^J = \left\{abs^J_\sigma\right\}$ with $abs^J_\sigma : D^{\mathrm{s}}_\sigma \times D^{\mathrm{s}}_\sigma \to D^J_\sigma$, by $abs^J_\sigma(d, d') = \alpha^J_\sigma(F\,\{(d, d')\})$.*

Thus the best interpretation for a constant $c : \tau$ is obtained by setting $c^J = abs_\tau(c^{\mathrm{s}}, c^{\mathrm{s}})$. In practice, we will want to calculate abs^J_ι directly for the base types and then give an inductive definition for all other types. Proposition 4.2.9 allows us to do this, but first we require the following lemma.

Lemma 4.2.8 *Suppose that each γ^J_σ has left adjoint α^J_σ. Let $\sigma_1, \ldots, \sigma_n, \tau \in \mathcal{T}$, with $n \geq 1$. Let $G \subseteq (D^{\mathrm{s}}_{\sigma_1 \to \cdots \to \sigma_n \to \tau}) \times (D^{\mathrm{s}}_{\sigma_1 \to \cdots \to \sigma_n \to \tau})$, and let $a_1 \in D^J_{\sigma_1}, \ldots, a_n \in D^J_{\sigma_n}$.*

Then:

$$\begin{array}{l}\alpha^J_{\sigma_1\to\cdots\to\sigma_n\to\tau}(F(G))\ a_1\cdots a_n\\ \quad = \ \alpha^J_\tau(F\left\{(g\ d_1\cdots d_n, g'\ d'_1\cdots d'_n)\mid (g,g')\in G, d_i\ (\gamma^J_{\sigma_i}\ a_i)\ d'_i, 1\le i\le n\right\}).\end{array}$$

Proof We prove the case for $n=1$, taking $\sigma_1=\sigma$. The result then follows by an easy induction on n. In the following we make use of the fact that $\top^J_\tau = \sqcap\emptyset$ and hence that $\gamma^J_\tau\ \top^J_\tau = \bigwedge\emptyset = All_\tau$, since γ^J_τ preserves meets.

Let LHS and RHS be the left and right hand sides of the equation. Let $b\in D^J_\tau$. We will show that LHS $\sqsubseteq b \iff$ RHS $\sqsubseteq b$. It is easy to see that if $h\in[D^J_\sigma\to D^J_\tau]$, then $h\ a\sqsubseteq b \iff h\sqsubseteq\lfloor a,b\rfloor$ (Definition 3.4.4). Thus LHS $\sqsubseteq b$ iff $\alpha^J_{\sigma\to\tau}(F(G))\sqsubseteq\lfloor a,b\rfloor$. Then, since $\alpha^J_{\sigma\to\tau}\circ F\dashv U\circ\gamma^J_{\sigma\to\tau}$, we have LHS $\sqsubseteq b$ iff $G\subseteq U(\gamma^J_{\sigma\to\tau}\ \lfloor a,b\rfloor)$. But by Proposition 3.4.5, $\gamma^J_{\sigma\to\tau}\ \lfloor a,b\rfloor = \gamma^J_\sigma\ a\Rightarrow\gamma^J_\tau\ b$, hence

$$\text{LHS}\sqsubseteq b \iff \forall(g,g')\in G.\, g\ (\gamma^J_\sigma\ a\Rightarrow\gamma^J_\tau\ b)\ g'.$$

Now since $\alpha^J_\tau\circ F\dashv U\circ\gamma^J_\tau$, we have

$$\text{RHS}\sqsubseteq b \iff \left\{(g\ d, g'\ d')\mid (g,g')\in G, d\ (\gamma^J_\sigma\ a)\ d'\right\}\subseteq U(\gamma^J_\tau\ b).$$

But this may be re-written to give

$$\text{RHS}\sqsubseteq b \iff \forall(g,g')\in G.\,\forall d,d'\in D^I_\sigma.\, d\ (\gamma^J_\sigma\ a)\ d'\Rightarrow(g\ d)\ (\gamma^J_\tau\ b)\ (g'\ d'),$$

which is just

$$\text{RHS}\sqsubseteq b \iff \forall(g,g')\in G.\, g\ (\gamma^J_\sigma\ a\Rightarrow\gamma^J_\tau\ b)\ g'.$$

$\square$

Proposition 4.2.9 *Let $\sigma,\tau\in\mathcal{T}$. Let $f,f'\in[D^{\mathrm{s}}_\sigma\to D^{\mathrm{s}}_\tau]$, let $a\in D^J_\sigma$ and let $(d_1,d_2),(d'_1,d'_2)\in D^{\mathrm{s}}_{\sigma\times\tau}$. Then:*

1. $abs^J_{\sigma\to\tau}(f,f')\ a = \bigsqcup\left\{abs^J_\tau(f\ d, f'\ d')\mid abs^J_\sigma(d,d')\sqsubseteq a\right\}$
2. $abs^J_{\sigma\times\tau}((d_1,d_2),(d'_1,d'_2)) = (abs^J_\sigma(d_1,d'_1), abs^J_\tau(d_2,d'_2))$.

Proof

1. Firstly, we have shown that $abs^J_\sigma(d,d')\sqsubseteq a \iff d\ (\gamma^J_\sigma\ a)\ d'$, so by the definition of abs^J, the right hand side of the equation is

$$\bigsqcup\left\{\alpha^J_\sigma(F(\{(f\ d, f'\ d')\}))\mid d\ (\gamma^J_\sigma\ a)\ d'\right\}.$$

Then using the fact that left adjoints preserve joins, we have

$$\begin{aligned}
&\bigsqcup\left\{\alpha^J_\sigma(F(\{(f\ d, f'\ d')\}))\mid d\ (\gamma^J_\sigma\ a)\ d'\right\}\\
&\quad=\ \alpha^J_\sigma(F(\bigcup\left\{\{(f\ d, f'\ d')\}\mid d\ (\gamma^J_\sigma\ a)\ d'\right\}))\\
&\quad=\ \alpha^J_\sigma(F\left\{(f\ d, f'\ d')\mid d\ (\gamma^J_\sigma\ a)\ d'\right\}).
\end{aligned}$$

But by the Lemma this is just $\alpha^J_{\sigma\to\tau}(F\ \{(f, f')\})\ a$.

2. As we have shown, $abs^J_{\sigma\times\tau}((d_1, d_2), (d'_1, d'_2))$ is uniquely determined as the pair (a_1, a_2) such that for any (b_1, b_2), we have $(d_1, d_2)\ (\gamma^J_{\sigma\times\tau}(b_1, b_2))\ (d'_1, d'_2)$ iff $(a_1, a_2) \sqsubseteq (b_1, b_2)$. Now $d_1\ (\gamma^J_\sigma\ b_1)\ d'_1$ iff $abs^J_\sigma(d_1, d'_1) \sqsubseteq b_1$, and similarly for abs^J_τ. Hence $(d_1, d_2)\ (\gamma^J_{\sigma\times\tau}(b_1, b_2))\ (d'_1, d'_2)$ iff $(abs^J_\sigma(d_1, d'_1), abs^J_\tau(d_2, d'_2)) \sqsubseteq (b_1, b_2)$, since $\gamma^J_{\sigma\times\tau}(b_1, b_2) = (\gamma^J_\sigma\ b_1) \times (\gamma^J_\tau\ b_2)$.

□

It is interesting to compare our abstraction maps with those of [BHA86]. The immediate difference is that the [BHA86] maps are from D^{s}_σ to D^J_σ, rather than from $D^{\mathrm{s}}_\sigma \times D^{\mathrm{s}}_\sigma$ to D^J_σ. At the 'top-level' we only really need a map of the former type, since we are interested in best interpretations for constants. Thus for the purposes of comparison we could define the maps $abs'_\sigma : D^{\mathrm{s}}_\sigma \to D^J_\sigma$ by $abs'_\sigma(d) = abs_\sigma(d, d)$. But the difference between these maps and the abstraction maps of [BHA86] is still very marked. In particular, the [BHA86] maps are continuous whereas in general, ours are not even monotone. An example is given in the next chapter. In a sense this could be construed as an advantage, since a recent result due to Samuel Kamin ([Kam91]) shows that the head-strictness property of [WH87] cannot be discovered via abstract interpretation based on monotone abstraction maps with finite range. In Chapter 5 we present an abstract interpretation using pers which is able to detect head-strictness and the associated abstraction maps are indeed non-monotone (the analysis is essentially the one described in [Hun90a], which pre-dates Kamin's result).

4.3 Non-Injective Concretisation Maps and Expected Forms

The example analyses of Chapter 5 show the condition that the concretisation maps preserve meets to be met quite naturally. Thus best interpretations are guaranteed to exist for the constants and are given by the abstraction maps defined above. Note that there is no way of automating the construction of a totally correct (guaranteed to terminate) algorithm for computing $abs^J_\sigma(c^{\mathrm{s}}, c^{\mathrm{s}})$, so we have to rely on human ingenuity. Nonetheless, in practice there does usually seem to be a fairly obvious construction for a correct interpretation for each constant, and if it's not immediately evident we can use the definition of the abs^J maps provided by Proposition 4.2.9 to guide us to it, as Burn shows in [Bur87, Bur91]. However, these 'obvious' interpretations for the constants are what Nielsen calls *expected forms* ([Nie85]) and in general are not the same as the best interpretations. We give an example of this below, which illustrates the typical cause of the difference between the expected form and the best interpretation: in general the concretisation maps are not injective at the higher types. We would normally assume the *base type* concretisation maps to be injective, since this corresponds to defining abstract lattices with no redundant points but, as is demonstrated by the constancy analysis described in Chapter 5, injectiveness of the concretisation maps is not an inherited property.

Suppose that for each pair of types σ, τ, the set of constants includes $\mathbf{apply}_{\sigma,\tau} : (\sigma \to \tau) \to \sigma \to \tau$, with the obvious standard interpretation given by

$$\mathbf{apply}^{\mathrm{s}}_{\sigma,\tau}\ f\ d = f\ d.$$

The natural candidate for the abstract interpretation of $\mathbf{apply}_{\sigma,\tau}$ simply mimics the standard interpretation:

$$\mathbf{apply}^J_{\sigma,\tau}\ h\ a = h\ a.$$

It should be clear that this will always be *correct* since γ^J is logical, but is it *best*?

For the rest of this section we will assume that the γ^J_σ preserve meets, each having left adjoint α^J_σ.

Lemma 4.3.1 *Let $\sigma, \tau \in \mathcal{T}$. Let $h \in [D^J_\sigma \to D^J_\tau]$ and let $a \in D^J_\sigma$. Then:*

$$\alpha^J_{\sigma\to\tau}(\gamma^J_{\sigma\to\tau}\ h)\ a = \alpha^J_\tau(F\left\{(f\ d, f'\ d') \mid f\ (\gamma^J_{\sigma\to\tau}\ h)\ f', d\ (\gamma^J_\sigma\ a)\ d'\right\}).$$

Proof Since $F \circ U = id$, we have

$$\begin{array}{rl}
\multicolumn{2}{l}{\alpha^J_{\sigma\to\tau}(\gamma^J_{\sigma\to\tau}\ h)\ a} \\
= & \alpha^J_{\sigma\to\tau}(F(U(\gamma^J_{\sigma\to\tau}\ h)))\ a \\
= & \alpha^J_\tau(F\left\{(f\ d, f'\ d') \mid (f, f') \in U(\gamma^J_{\sigma\to\tau}\ h), d\ (\gamma^J_\sigma\ a)\ d'\right\},
\end{array}$$

where the last step is by Lemma 4.2.8. □

For any adjoint pair $f \dashv g$, an easy argument shows that g is injective if and only if $f \circ g = id$, so a corollary of this lemma is that if $\gamma^J_{\sigma\to\tau}$ is injective then $\mathbf{apply}^J_{\sigma,\tau}$ is indeed best, since for any h and a:

$$\begin{aligned}
&\alpha^J_{(\sigma\to\tau)\to\sigma\to\tau}(F\left\{(\mathbf{apply}^{\mathrm{s}}_{\sigma,\tau}, \mathbf{apply}^{\mathrm{s}}_{\sigma,\tau})\right\})\ h\ a \\
&= \alpha^J_\tau(F\left\{(\mathbf{apply}^{\mathrm{s}}_{\sigma,\tau}\ f\ d, \mathbf{apply}^{\mathrm{s}}_{\sigma,\tau}\ f'\ d') \mid f\ (\gamma^J_{\sigma\to\tau}\ h)\ f', d\ (\gamma^J_\sigma\ a)\ d'\right\}) \\
&= \alpha^J_\tau(F\left\{(f\ d, f'\ d') \mid (f, f') \in U(\gamma^J_{\sigma\to\tau}\ h), d\ (\gamma^J_\sigma\ a)\ d'\right\} \\
&= \alpha^J_{\sigma\to\tau}(\gamma^J_{\sigma\to\tau}\ h)\ a \\
&= h\ a.
\end{aligned}$$

Now assume that γ^J_τ is injective (for example τ could be a base type) but suppose that $\gamma^J_{\sigma\to\tau}$ is not (this is possible, as shown by the constancy analysis of the next chapter). Hence there are functions h_1, h_2 such that $\gamma^J_{\sigma\to\tau}\ h_1 = \gamma^J_{\sigma\to\tau}\ h_2 = P$, with some a such that $h_1\ a \neq h_2\ a$. Let app be the best interpretation for $\mathbf{apply}_{\sigma,\tau}$, i.e., $app = abs^J_{(\sigma\to\tau)\to\sigma\to\tau}(\mathbf{apply}^{\mathrm{s}}_{\sigma,\tau}, \mathbf{apply}^{\mathrm{s}}_{\sigma,\tau})$. By Lemma 4.2.8 it follows that

$$app\ h_1\ a = app\ h_2\ a = \alpha^J_\tau(F\left\{(f\ d, f'\ d') \mid f\ P\ f', d\ (\gamma^J_\sigma\ a)\ d'\right\}), \tag{4.3.2}$$

so $\mathbf{apply}^J_{\sigma,\tau}$ is certainly not best, since $\mathbf{apply}^J_{\sigma,\tau}\ h_1\ a = h_1\ a \neq h_2\ a = \mathbf{apply}^J_{\sigma,\tau}\ h_2\ a$. We will show that using app as the interpretation for $\mathbf{apply}_{\sigma,\tau}$ leads to a more accurate analysis than the obvious $\mathbf{apply}^J_{\sigma,\tau}$, under the following assumption: there are closed terms e_1, e_2 and e_a with abstract interpretations h_1, h_2 and a, respectively. To make the comparison in a straightforward way, we will assume that there is an alternative apply constant $\mathbf{altapply}_{\sigma,\tau} : (\sigma\to\tau)\to\sigma\to\tau$, such that $\mathbf{altapply}^{\mathrm{s}}_{\sigma,\tau} = \mathbf{apply}^{\mathrm{s}}_{\sigma,\tau}$ but $\mathbf{altapply}^J_{\sigma,\tau} = app$. To show that using the best interpretation leads to a more accurate analysis, we show that there are terms $e : \tau$ and $e' : \tau$ which differ only in that e uses $\mathbf{altapply}_{\sigma,\tau}$ where e' uses $\mathbf{apply}_{\sigma,\tau}$, but that $\gamma^J_\tau[\![e]\!]^J$ is strictly smaller than $\gamma^J_\tau[\![e']\!]^J$.

Consider the following pers:

- $P_1 = \gamma^J_\tau[\![\mathbf{altapply}_{\sigma,\tau}\ e_1\ e_a]\!]^J = \gamma^J_\tau(app\ h_1\ a)$
- $P_2 = \gamma^J_\tau[\![\mathbf{altapply}_{\sigma,\tau}\ e_2\ e_a]\!]^J = \gamma^J_\tau(app\ h_2\ a)$
- $P'_1 = \gamma^J_\tau[\![\mathbf{apply}_{\sigma,\tau}\ e_1\ e_a]\!]^J = \gamma^J_\tau(h_1\ a)$
- $P'_2 = \gamma^J_\tau[\![\mathbf{apply}_{\sigma,\tau}\ e_2\ e_a]\!]^J = \gamma^J_\tau(h_2\ a)$

Since app is best for both $\mathbf{apply}_{\sigma,\tau}$ and $\mathbf{altapply}_{\sigma,\tau}$, and since γ^J_τ is monotone, it follows that $P_1 \leq P'_1$ and $P_2 \leq P'_2$. Furthermore, $P'_1 \neq P'_2$ since γ^J_τ is injective and $h_1\ a \neq h_2\ a$. But by (4.3.2), $P_1 = P_2$, hence either P_1 is strictly less than P'_1 or P_2 is strictly less than P'_2.

The assumption that was necessary to show that the best interpretation for $\mathbf{apply}_{\sigma,\tau}$ has a practical advantage over the obvious one deserves closer scrutiny. Given that $\gamma^J_{\sigma\to\tau}$ is not injective, we could argue that $D^J_{\sigma\to\tau}$ contains too many points, since at least two of them (h_1 and h_2) represent the same property. It is natural to ask whether all these points are *denotable* under the abstract interpretation, in particular, do the expressions e_1 and e_2 assumed above actually exist? In fact, it would be interesting to go further than this and look for a a result akin to *full abstraction* ([Plo77]). Take a program to be a closed term of base type and take concretisation as a notion of observation, i.e., say that programs $e, e' : \iota$ are *observably equal* if and only if $\gamma^J_\iota[\![e]\!]^J = \gamma^J_\iota[\![e']\!]^J$. Then assuming that all constants other than $\mathbf{altapply}_{\sigma,\tau}$ are interpreted by their 'expected forms', does there exist a program context $C[]$ such that $C[\mathbf{apply}_{\sigma,\tau}]$ and $C[\mathbf{altapply}_{\sigma,\tau}]$ are not observably equal? The glaring obstacle to answering such a question is that we have no good definition of what an expected form *is*. One possibility, suggested by the work of [Nie84, Nie85], would be to attempt to characterise expected forms in terms of their behaviour on *irreducible elements* (see Chapter 8). This subject remains to be investigated.

4.4 Pers Subsume Sets

It is intuitively clear that the use of pers as program properties subsumes the use of sets. This is simply demonstrated by encoding each subset of a domain $X \subseteq D$ as the per $P(X)$, where

$$d \; P(X) \; d' \iff d \in X \textit{ and } d' \in X.$$

Thus the per $P(X)$ satisfies $|P(X)| = X$ and all $d \in X$ belong to the same equivalence class of $P(X)$. Note that $X \in \mathcal{P}_H(D)$ iff $P(X) \in \mathrm{CPER}(D)$. (An alternative is to encode X as the diagonal $\Delta(X)$, where $d \; \Delta(X) \; d' \iff d = d' \in X$. See [Hun90b].)

Let (K, R) be a set-based (binary logical relation) abstract interpretation with associated concretisation map γ. Let (K, R') be the per-based abstract interpretation with the same D^K_σ and c^K_τ, with associated logical concretisation map γ' induced from the base-types by

$$\gamma'_\iota \; a = P(\gamma_\iota \; a).$$

Then it is straightforward to show (induction on σ) that for all σ and for all $b \in D^K_\sigma$:

$$\gamma'_\sigma \ b = P(\gamma_\sigma \ b).$$

It follows that (K, R) is correct if and only if (K, R') is correct. Thus any set-based analysis can trivially be presented as a per-based one: the converse does not appear to be true.

Chapter 5

Example Analyses

We describe two example abstract interpretations based on the framework developed in previous chapters. The first is what we call a constancy analysis which could form the basis of a higher-order binding time analysis, as discussed in Section 3.1. The second is a strictness analysis able to discover the head-strictness property of [WH87]. Because head-strictness only makes sense for functions over lists we must extend our language of types. In this chapter this is done in a rather ad-hoc way. In the next chapter a slightly more satisfactory approach is taken, in which the language of types is extended to include recursive types. Unfortunately, our method for inducing abstract interpretations for these types does not generalise the strictness analysis of this chapter. The point is discussed further at the end of the next chapter.

5.1 A Constancy Analysis

In Chapter 3 we introduced the property of *constancy* to illustrate the weakness of the binary logical relations framework. In this section we present an analysis for reasoning about constancy which exploits the extra power of the ternary logical relations framework. This analysis is based on one developed jointly with David Sands, described in [HS91]. It can be seen as generalisation of the basic analysis of [Lau89] to the higher-order case, although there are a number of important aspects of that work which we do not address: in particular, domain factorisation, polymorphism and the construction of a 'global' analysis to propagate the local constancy information throughout a program.

We postpone giving an example of the analysis 'in action' until the next chapter, where we show how it can be extended to programs using recursively defined types.

5.1.1 The Abstract Domains and Concretisation Maps

Recall that for a function $f : D \to E$, constancy can be expressed as

$$f : All_D \Rightarrow Id_E.$$

It is thus natural to define an interpretation $\mathbf{C}$ (C for Constancy) with two-point base type interpretations

$$D^{\mathrm{C}}_{\iota} = \mathbf{2} = \begin{array}{c}\mathrm{D}\\ | \\ \mathrm{S}\end{array},$$

where as a mnemonic convenience we name the elements of $\mathbf{2}$ as D (Dynamic) and S (Static) instead of 1 and 0. The intention is that D and S should correspond to *All* and *Id* (of the appropriate types) respectively. This correspondence is established by the concretisation maps $\gamma^{\mathrm{C}}_{\iota} : D^{\mathrm{C}}_{\iota} \to \mathrm{CPER}(D^{\mathrm{S}}_{\iota})$, where

$$\gamma^{\mathrm{C}}_{\iota}\ a = \begin{cases} All_{\iota} & \text{if } a = \mathrm{D} \\ Id_{\iota} & \text{if } a = \mathrm{S}. \end{cases}$$

The logical concretisation map $\gamma^{\mathrm{C}} = \{\gamma^{\mathrm{C}}_{\sigma}\}_{\sigma \in \mathcal{T}}$ is induced via the Logical Concretisation Map Theorem (3.4.2). The associated logical relation R^{C}: $\mathbf{S}^2 \leftrightarrow \mathbf{C}$ is that induced by

$$\begin{aligned} (d, d')\ R^{\mathrm{C}}_{\iota}\ \mathrm{D} & \text{ for all } d, d' \in D^{\mathrm{S}}_{\iota} \\ (d, d')\ R^{\mathrm{C}}_{\iota}\ \mathrm{S} & \iff d = d' \end{aligned}$$

Each $\gamma^{\mathrm{C}}_{\sigma}$ is:

- CP;
- meet-preserving.

By Propositions 3.4.3 and 4.2.5 it is sufficient only to verify these at the base types. This is trivially done. It is also trivially shown to be the case that the base type concretisation maps are injective but, as we show in Subsection 5.1.3, this property is not inherited at the function types.

Since γ^{C} preserves meets, we know that each $\gamma^{\mathrm{C}}_{\sigma}$ has a left adjoint $\alpha^{\mathrm{C}}_{\sigma}$ and hence that best interpretations exist for the constants. The base type abstraction maps are easily verified to be given by:

$$abs^{\mathrm{C}}_{\iota}(d, d') = \begin{cases} \mathrm{S} & \text{if } d = d' \\ \mathrm{D} & \text{otherwise.} \end{cases}$$

$$D_\iota^{\mathrm{C}} = \begin{array}{c}\mathrm{D}\\ |\\ \mathrm{S}\end{array} \quad \gamma_\iota^{\mathrm{C}}\ \mathrm{D} = All_\iota \quad \gamma_\iota^{\mathrm{C}}\ \mathrm{S} = Id_\iota$$

$$\begin{aligned}
&\mathbf{n}^{\mathrm{C}} = \mathrm{S}\\
&\mathbf{true}^{\mathrm{C}} = \mathbf{false}^{\mathrm{C}} = \mathrm{S}\\
&\mathbf{iszero}^{\mathrm{C}}\ a = a\\
&\mathbf{plus}^{\mathrm{C}}\ a\ b = a \sqcup b\\
&\mathbf{minus}^{\mathrm{C}} = \mathbf{mult}^{\mathrm{C}} = \mathbf{plus}^{\mathrm{C}}\\
&\mathbf{if}_\sigma^{\mathrm{C}}\ a\ b_1\ b_2 = \begin{cases} \top_\sigma^{\mathrm{C}} & \text{if } a = \mathrm{D}\\ b_1 \sqcup b_2 & \text{if } a = \mathrm{S}\end{cases}\\
&\mathbf{Y}_\sigma^{\mathrm{C}}\ f = \bigsqcup_{i\in\omega} f^i\ \bot_\sigma^{\mathrm{C}}
\end{aligned}$$

Figure 5.1: An Abstract Interpretation for Constancy Analysis

The simplest function-type abstraction maps are shown by Proposition 4.2.9 to be given by:

$$abs_{\iota\to\iota}^{\mathrm{C}}(f, f')a = \bigsqcup \{ abs_\iota^{\mathrm{C}}(f\ d, f'\ d') \mid abs_\iota^{\mathrm{C}}(d, d') \sqsubseteq a \}\,.$$

It is then straightforward to show that $abs_{\iota\to\iota}^{\mathrm{C}}(f, f') = \lambda\, a \in \mathbf{2}\,.\,\mathrm{S}$ iff f and f' are equal and constant, and similarly that $abs_{\iota\to\iota}^{\mathrm{C}}(f, f') = \lambda\, a \in \mathbf{2}\,.\,\mathrm{D}$ iff f and f' are not equal. The only possibility left is that $abs_{\iota\to\iota}^{\mathrm{C}}(f, f') = \lambda\, a \in \mathbf{2}\,.\,a$ iff f and f' are equal and not constant. It follows that the derived map $f \mapsto abs_{\iota\to\iota}^{\mathrm{C}}(f, f)$ mentioned in the previous chapter (Subsection 4.2.2) is not monotone since, for example, $(\lambda\, n \in \mathbf{Z}\,.\,\bot) \mapsto \lambda\, a \in \mathbf{2}\,.\,\mathrm{S}$ and $(\lambda\, n \in \mathbf{Z}\,.\,101) \mapsto \lambda\, a \in \mathbf{2}\,.\,\mathrm{S}$, but $(\lambda\, n \in \mathbf{Z}\,.\,n = \bot \to \bot, 101) \mapsto \lambda\, a \in \mathbf{2}\,.\,a$.

5.1.2 The Interpretations of Constants

The interpretation $\mathbf{C}$, including the interpretations of the constants, is shown in Figure 5.1. Recall, for a constant $c : \sigma$, that c^{C} is correct if and only if $c^{\mathrm{S}} : \gamma_\sigma^{\mathrm{C}}\ c^{\mathrm{C}}$. We show below that each c^{C} is correct, and hence, by the Correctness Theorem (4.1.7), that the interpretation $\mathbf{C}$ is correct.

Nullary Constants

The interpretations of the nullary constants $\mathbf{true}, \mathbf{false}, \mathbf{0}, \mathbf{1}, \ldots$ are all S. From the fact that $\gamma_{bool}^{\mathrm{C}}\ \mathrm{S} = Id_{bool}$, $\gamma_{int}^{\mathrm{C}}\ \mathrm{S} = Id_{int}$ and $|Id_D| = D$ for any D, correctness of the nullary constants is immediate. Clearly, these interpretations are all best since they are all least.

Arithmetic Operators and Test for Zero

For the test for zero, since $\perp\ All_{int}\ 0$, we must have $\perp\ \gamma^{\mathrm{H}}_{bool}(\mathbf{iszero}^{\mathrm{H}}\ \mathrm{D})\ tt$, so $\mathbf{iszero}^{\mathrm{H}}\ \mathrm{D}$ has to be D. Like all functions, $\mathbf{iszero}^{\mathrm{S}} : Id \Rightarrow Id$, so it is correct to take $\mathbf{iszero}^{\mathrm{S}}\ \mathrm{S} = \mathrm{S}$. Thus $\mathbf{iszero}^{\mathrm{H}}$ is correct (and best).

To show correctness for the binary arithmetic operators, we must show that, taking $f : \mathbf{Z} \to \mathbf{Z} \to \mathbf{Z}$ to be any of $\mathbf{plus}^{\mathrm{S}}$, $\mathbf{minus}^{\mathrm{S}}$, $\mathbf{mult}^{\mathrm{S}}$, for all $a, b \in \mathbf{2}$:

$$f : \gamma^{\mathrm{C}}_{int}\ a \Rightarrow \gamma^{\mathrm{C}}_{int}\ b \Rightarrow \gamma^{\mathrm{C}}_{int}(a \sqcup b). \tag{5.1.1}$$

Now if either $a = \mathrm{D}$ or $b = \mathrm{D}$ then $a \sqcup b = \mathrm{D}$ and (5.1.1) is trivially satisfied, (for any f, not just $\mathbf{plus}^{\mathrm{S}}$, $\mathbf{minus}^{\mathrm{S}}$ and $\mathbf{mult}^{\mathrm{S}}$), since $\gamma^{\mathrm{C}}_{int}\ \mathrm{D} = All_{int}$. Otherwise, $a = b = (a \sqcup b) = \mathrm{S}$. But $\gamma^{\mathrm{C}}_{int}\ \mathrm{S} = Id_{int}$ and $f : Id_{int} \Rightarrow Id_{int} \Rightarrow Id_{int}$ clearly holds (again, for any f).

We can see that these interpretations are also best: suppose towards a contradiction that $\mathbf{plus}^{\mathrm{C}}$ is not best. Then there must be a, b such that $a \sqcup b = \mathrm{D}$ and $\mathbf{plus}^{\mathrm{S}} : \gamma^{\mathrm{C}}_{int}\ a \Rightarrow \gamma^{\mathrm{C}}_{int}\ b \Rightarrow Id_{int}$. But either a or b must be D, so $\mathbf{plus}^{\mathrm{S}}$ must be constant in one of its arguments, a contradiction. The same argument obviously applies to **minus** and **mult**.

Conditionals

To justify the interpretation used for the conditionals we will use the abstraction maps. We require the following result:

Lemma 5.1.2 *Let $\sigma \in \mathcal{T}$ and let $a \in D^{\mathrm{C}}_{\sigma}$. Then there is some $d \in |\gamma^{\mathrm{C}}_{\sigma}\ a|$ such that $abs^{\mathrm{C}}_{\sigma}(\perp^{\mathrm{S}}_{\sigma}, d) = \top^{\mathrm{C}}_{\sigma}$.*

Proof (sketch) First define the family of sets $\{M_\sigma\}_{\sigma\in\mathcal{T}}$ with each $M_\sigma \subseteq D^{\mathrm{S}}_{\sigma}$, by:

- $M_\iota = D^{\mathrm{S}}_{\iota} \setminus \perp$;
- $M_{\sigma\times\tau} = M_\sigma \times M_\tau$;
- $M_{\sigma\to\tau} = \{\lambda\, d \in D^{\mathrm{S}}_{\sigma} \,.\, m \mid m \in M_\tau\}$.

Then it can be shown that for all $\sigma \in \mathcal{T}$, for all $a \in D^{\mathrm{C}}_{\sigma}$ and $d \in M_\sigma$:

- M_σ is not empty;

- $M_\sigma \subseteq |\gamma^{\mathrm{C}}_\sigma\ a|$;
- $\bot^{\mathrm{S}}_\sigma\ (\gamma^{\mathrm{C}}_\sigma\ a)\ d \Rightarrow \gamma^{\mathrm{C}}_\sigma\ a = All_\sigma$.

The first of these is obvious from the definition. The second and third are by easy inductions on σ. It can further be shown that in addition to being top-preserving, each $\gamma^{\mathrm{C}}_\sigma$ is *top-reflecting*, which is to say that $\gamma^{\mathrm{C}}_\sigma\ a = All_\sigma \Rightarrow a = \top^{\mathrm{C}}_\sigma$. Then for any $d \in M_\sigma$

$$\begin{array}{ll} & abs^{\mathrm{C}}_\sigma(\bot^{\mathrm{S}}_\sigma, d) \sqsubseteq a \\ \Rightarrow & \bot^{\mathrm{S}}_\sigma\ (\gamma^{\mathrm{C}}_\sigma\ a)\ d \\ \Rightarrow & \gamma^{\mathrm{C}}_\sigma\ a = All_\sigma \\ \Rightarrow & a = \top^{\mathrm{C}}_\sigma. \end{array}$$

□

Omitting most of the superscripts and subscripts to keep things readable, we have

$$\begin{array}{l} abs^{\mathrm{C}}(\mathbf{if}^{\mathrm{S}}_\sigma, \mathbf{if}^{\mathrm{S}}_\sigma)\ \mathrm{D}\ b\ c \\ \quad = \ \bigsqcup \{ abs(\mathbf{if}^{\mathrm{S}}_\sigma\ x\ y\ z, \mathbf{if}^{\mathrm{S}}_\sigma\ x'\ y'\ z') \mid abs(x,x') \sqsubseteq \mathrm{D}, abs(y,y') \sqsubseteq b, abs(z,z') \sqsubseteq c\}\,. \end{array}$$

Since D is top, $abs^{\mathrm{C}}_{bool}(x,x') \sqsubseteq \mathrm{D}$ for all $(x,x') \in \mathbf{B}$. In particular, $abs^{\mathrm{C}}_{bool}(\bot, tt) \sqsubseteq \mathrm{D}$. Thus the set on the right hand side contains $abs^{\mathrm{C}}_\sigma(\bot^{\mathrm{S}}_\sigma, z')$ for all $z' \in |\gamma^{\mathrm{C}}_\sigma\ b|$, so by the Lemma, the set contains $\top^{\mathrm{C}}_\sigma$. Thus $abs^{\mathrm{C}}(\mathbf{if}^{\mathrm{S}}_\sigma, \mathbf{if}^{\mathrm{S}}_\sigma)\ \mathrm{D}\ b\ c = \top^{\mathrm{C}}_\sigma$, which agrees with our choice for $\mathbf{if}^{\mathrm{C}}_\sigma$.

Using the fact that $\gamma^{\mathrm{C}}_{bool}\ \mathrm{S} = Id_{bool}$, we have:

$$\begin{array}{l} abs^{\mathrm{C}}(\mathbf{if}^{\mathrm{S}}_\sigma, \mathbf{if}^{\mathrm{S}}_\sigma)\ \mathrm{S}\ b\ c \\ \quad = \ \bigsqcup \{ abs^{\mathrm{C}}_\sigma(\bot^{\mathrm{S}}_\sigma, \bot^{\mathrm{S}}_\sigma), abs^{\mathrm{C}}_\sigma(y,y'), abs^{\mathrm{C}}_\sigma(z,z') \mid abs^{\mathrm{C}}_\sigma(y,y') \sqsubseteq b, abs^{\mathrm{C}}_\sigma(z,z') \sqsubseteq c\}\,. \end{array}$$

By adjointness considerations this is equivalent to

$$abs^{\mathrm{C}}(\mathbf{if}^{\mathrm{S}}_\sigma, \mathbf{if}^{\mathrm{S}}_\sigma)\ \mathrm{S}\ b\ c \quad = \quad \alpha^{\mathrm{C}}_\sigma(\gamma^{\mathrm{C}}_\sigma\ b) \sqcup \alpha^{\mathrm{C}}_\sigma(\gamma^{\mathrm{C}}_\sigma\ c).$$

Now for σ such that $\gamma^{\mathrm{C}}_\sigma$ is injective, the right hand side is just $b \sqcup c$. So in particular, for the base types we have shown that the $\mathbf{if}^{\mathrm{C}}_\iota$ are best. In general, as is shown below, $\gamma^{\mathrm{C}}_\sigma$ is not injective and we only have $\alpha^{\mathrm{C}}_\sigma(\gamma^{\mathrm{C}}_\sigma\ b) \sqcup \alpha^{\mathrm{C}}_\sigma(\gamma^{\mathrm{C}}_\sigma\ c) \sqsubseteq b \sqcup c$, so the $\mathbf{if}^{\mathrm{C}}_\sigma$ are correct but not necessarily best.

Recursion Combinators

C is a least fixed point interpretation so correctness of $\mathbf{Y}^{\mathrm{C}}_\sigma$ is given by Corollary 4.1.9.

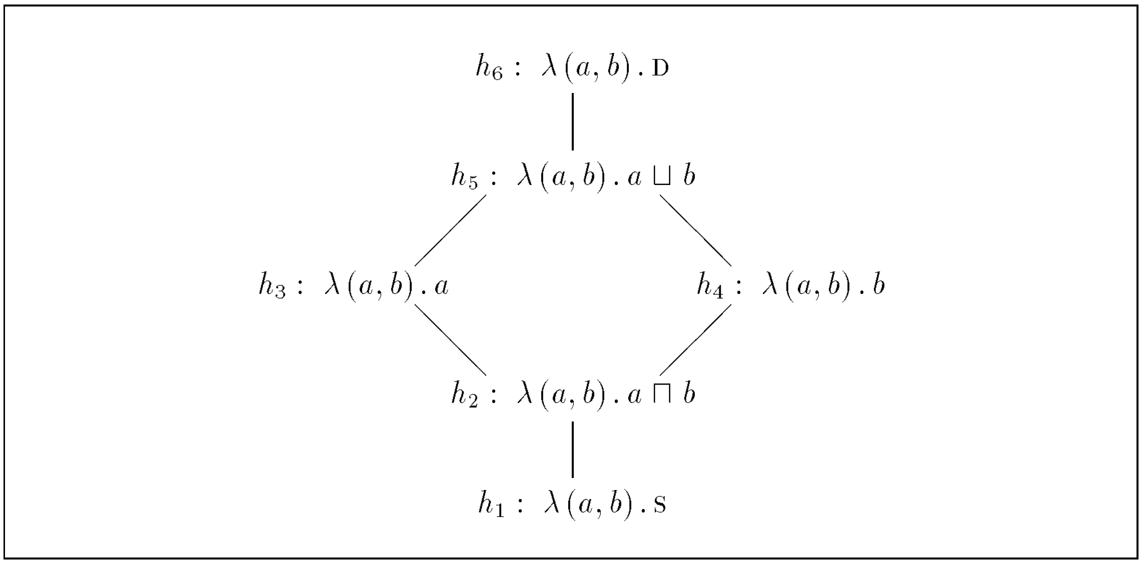

Figure 5.2: The Lattice $[\mathbf{2} \times \mathbf{2} \to \mathbf{2}]$

5.1.3 Non-Injective Concretisation Maps

As remarked above, the $\gamma^{\mathrm{C}}_{\sigma}$ maps are not all injective. To show this we consider the example of $\sigma = int \times int \to int$ (the choice of int is not significant, any base type would do). The abstract lattice interpretation for this type is $[\mathbf{2} \times \mathbf{2} \to \mathbf{2}]$, shown in Figure 5.2. We will show that $\gamma^{\mathrm{C}}_{\sigma}\ h_1 = \gamma^{\mathrm{C}}_{\sigma}\ h_2$.

Using the Logical Concretisation Map Theorem, and dropping type subscripts for the sake of readability, we see that

$$\gamma^{\mathrm{C}}\ h_2 = P_1 \wedge P_2 \wedge P_3 \wedge P_4,$$

where

$$\begin{aligned} P_1 &= All \times All \Rightarrow All \\ P_2 &= All \times Id \Rightarrow Id \\ P_3 &= Id \times All \Rightarrow Id \\ P_4 &= Id \times Id \Rightarrow Id. \end{aligned}$$

Using Proposition 3.2.4 we can see that $P_1 = All$ and $P_2, P_3 \le P_4$. Hence

$$\gamma^{\mathrm{C}}\ h_2 = P_2 \wedge P_3.$$

Similarly, we have

$$\gamma^{\mathrm{C}}\ h_1 = Q_1 \wedge Q_2 \wedge Q_3 \wedge Q_4,$$

where

$$\begin{aligned} Q_1 &= All \times All \Rightarrow Id \\ Q_2 &= All \times Id \Rightarrow Id \\ Q_3 &= Id \times All \Rightarrow Id \\ Q_4 &= Id \times Id \Rightarrow Id. \end{aligned}$$

Again using Proposition 3.2.4, we find that $Q_1 \leq Q_2, Q_3, Q_4$. Thus

$$\gamma^{\mathrm{C}}\ h_1 = Q_1.$$

Since $Q_1 \leq P_2, P_3$, we know that $\gamma^{\mathrm{C}}\ h_1 \leq \gamma^{\mathrm{C}}\ h_2$. Note that $P_4 = Q_4 = Id$, so that $P_2, P_3, Q_1 \leq Id$. Any per P with this property is completely determined by $|P|$, so to show that $\gamma^{\mathrm{C}}\ h_1 = \gamma^{\mathrm{C}}\ h_2$ it remains to show that $f \in |\gamma^{\mathrm{C}}\ h_2|$ implies $f \in |\gamma^{\mathrm{C}}\ h_1|$. Now let $f \in |\gamma^{\mathrm{C}}\ h_2|$, and let $n_1, n_2, n_1', n_2' \in \mathbf{Z}$. Then $f(n_1, n_2) = f(n_1', n_2)$, since $f : All \times Id \Rightarrow Id$, and $f(n_1', n_2) = f(n_1', n_2')$, since $f : Id \times All \Rightarrow Id$. Thus $f(n_1, n_2) = f(n_1', n_2')$. Hence $f : All \times All \Rightarrow Id$. We conclude that $\gamma^{\mathrm{C}}_{int \times int \rightarrow int}\ h_1 = \gamma^{\mathrm{C}}_{int \times int \rightarrow int}\ h_2$ and hence that $\gamma^{\mathrm{C}}_{int \times int \rightarrow int}$ is not injective.

The crux of the above argument is in the last few lines which reveal the basic cause of non-injectiveness for $\gamma^{\mathrm{C}}_{int \times int \rightarrow int}$: a binary function is separately constant in each argument if and only if it is jointly constant in both its arguments. By mapping (D, S) to S and mapping (S, D) to S, the function h_2 has already completely determined a property (per). What it does to (D, D) is really not important. Thus there is an inherent redundancy in using the full space of monotone functions to describe properties of functions of more than one argument. (It is not hard to see that the restriction of $\gamma^{\mathrm{C}}_{int \times int \rightarrow int}$ to $\{h_3, \ldots, h_6\}$ is injective, so the only redundancy in this particular lattice is caused by having both h_1 and h_2.) In [CC79] it is suggested that such redundancy be eliminated by quotienting out the superfluous points: the suggested quotient map is simply $\alpha^{\mathrm{C}}_\sigma \circ \gamma^{\mathrm{C}}_\sigma$, which does the job admirably. The problem with that suggestion in this setting is that we have to cope with the full type structure of $\mathcal{T}$, which means infinitely many σ, and in general we simply don't know how to compute $\alpha^{\mathrm{C}}_\sigma \circ \gamma^{\mathrm{C}}_\sigma$. What is required is an alternative characterisation of those abstract functions which are superfluous.

As yet we don't have such a characterisation, but we conjecture that one will be found by adapting Berry's work on stable functions. We note that the function h_2 is distinguished from the other functions in $[\mathbf{2} \times \mathbf{2} \rightarrow \mathbf{2}]$ by the fact that it doesn't preserve binary joins. However, this property alone does not give us the characterisation we seek, since *all* functions in $[\mathbf{2} \rightarrow [\mathbf{2} \rightarrow \mathbf{2}]]$ preserve binary joins, but the curried version of h_2 is superfluous there just as h_2 is in $[\mathbf{2} \times \mathbf{2} \rightarrow \mathbf{2}]$. The answer may be to adapt [Ber78] and work with bidomains (perhaps more properly, bilattices) using the *coconsistently additive (ca)* functions[1] (dual to *consistently multiplicative*) and the associated ca ordering. This gives rise to a lattice for $int \times int \to int$ which is just $[\mathbf{2} \times \mathbf{2} \to \mathbf{2}]$ without h_2, and a lattice for $int \to int \to int$ which is just $[\mathbf{2} \to [\mathbf{2} \to \mathbf{2}]]$ without the curried version of h_2. More generally it gives rise to a cartesian closed category of finite lattices in which the interpretation $\mathbf{C}$ could be redefined. We do not know whether this is the characterisation we seek. A possible analogue of Berry's Function (used to demonstrate that stability $\neq$ sequentiality), is suggested by the fact that under the ca ordering, the function space $[\mathbf{2} \to \mathbf{2}]$ is isomorphic to $\mathbf{B}^{\mathrm{OP}}$:

$$\begin{tikzcd}
 & T : \lambda a \,.\, \mathrm{D} \arrow[dl, dash] \arrow[dr, dash] & \\
B : \lambda a \,.\, \mathrm{S} & & I : \lambda a \,.\, a
\end{tikzcd}$$

We define the function $q : [\mathbf{2} \to \mathbf{2}]^3 \to [\mathbf{2} \to \mathbf{2}]$ to be the greatest monotonic function such that

$$\begin{aligned}
q(I, B, T) &= I \\
q(B, T, I) &= I \\
q(T, I, B) &= I.
\end{aligned}$$

Then q and $\lambda (f, g, h) \,.\, I$ are distinct functions which are both ca, but we do not know whether $\gamma^{\mathrm{C}}(q) = \gamma^{\mathrm{C}}(\lambda (f, g, h) \,.\, I)$.

5.1.4 Constancy in a Strict Language

The standard semantics assumed for $\Lambda_{\mathcal{T}}$ is non-strict. We could model a strict functional language in $\Lambda_{\mathcal{T}}$ by providing strict apply combinators $\mathbf{sapply}_{\sigma,\tau} : (\sigma \to \tau) \to \sigma \to \tau$ with standard interpretations:

$$\mathbf{sapply}^{\mathrm{S}}_{\sigma,\tau}\ f\ d \;=\; \begin{cases} \bot & \text{if } d = \bot \\ f\ d & \text{otherwise.} \end{cases}$$

A strict language semantics could then be simulated by replacing all applications $[\![e_1\ e_2]\!]$ by $[\![\mathbf{sapply}\ e_1\ e_2]\!]$ except where non-strictness is needed (such as the application of a conditional to its second and third arguments). Unfortunately, the resulting analysis would be rather uninteresting. Since we have formalised constancy of f by

$$f : All \Rightarrow Id,$$

[1] A function f is ca if for all pairs a, b which are bounded below, $f(a \sqcup b) = f(a) \sqcup f(b)$.

the only constant strict function is the constant bottom function. It follows that, for example, any correct choice for $\mathbf{sapply}^{\mathrm{C}}_{int,int}$ will be such that $\mathbf{sapply}^{\mathrm{C}}_{int,int}\ h\ \mathrm{D} = \mathrm{D}$, for all $h \in D^{\mathrm{C}}_{int \to int}$. Thus no term $[\![\mathbf{sapply}_{int,int}\ e]\!]$ denoting a strict function will be found to be constant. It is possible to formulate a modified definition of constancy, more appropriate for strict languages, as follows. Define the per All'_D by

$$d\ All'_D\ d' \iff (d \neq \bot\ and\ d' \neq \bot) \vee d = d' = \bot.$$

Say that a strict function $f : D \to E$ is *almost constant* if

$$f : All'_D \Rightarrow Id_E.$$

Thus a function is almost constant if its restriction to non-$\bot$ arguments is constant. Now using $\mathbf{C}$ we could determine that $\lambda\,\mathtt{x}\,.\,((\lambda\,\mathtt{y}\,.\,\mathbf{3})\ (e\ \mathtt{x}))$ denoted a constant function, regardless of what e might be. But to determine that the term $\lambda\,\mathtt{x}\,.\,(\mathbf{sapply}\ (\lambda\,\mathtt{y}\,.\,\mathbf{3})\ (\mathbf{sapply}\ e\ \mathtt{x}))$ denoted an *almost* constant function, we would have to determine that e satisfied the condition

$$[\![\mathbf{sapply}\ e]\!]^{\mathrm{S}} : All' \Rightarrow All'.$$

Unfortunately, this amounts to determining that $[\![\mathbf{sapply}\ e]\!]^{\mathrm{S}}$ is a *total* function (one mapping non-$\bot$ arguments to non-$\bot$ results) and, as the termination analysis of [Abr90] demonstrates, an analysis based on abstract interpretation which is able to detect totality is likely to be so poor as to be virtually useless. In practice, binding time analyses for strict languages analyse programs as though the semantics were actually non-strict ([Jon88]) and these analyses may be viewed as being based on some form of partial correctness criterion.

5.2 List Types

The extended language of types is as follows:

$$\sigma, \tau \in \mathcal{T}^l\ ::= \iota \mid \sigma_1 \times \sigma_2 \mid \sigma_1 \to \sigma_2 \mid list(\sigma).$$

We do not extend the language of terms, relying instead on higher-order constants to operate on lists. This is not very satisfactory, but it does reduce the amount of work to be done in adapting the existing framework to incorporate the new types.

5.2.1 Interpretations

The definition of an interpretation is modified as follows: an interpretation I is now a pair

$$(\left\{D^I_\sigma\right\}_{\sigma\in\mathcal{T}^l}, \left\{K^I_\sigma\right\}_{\sigma\in\mathcal{T}^l}),$$

where the K^I_σ assign meanings to constants, as before, and the D^I_σ are subject to the following conditions:

$$\begin{aligned} D^I_{\sigma_1\times\sigma_2} &= D^I_{\sigma_1} \times D^I_{\sigma_2} \\ D^I_{\sigma\to\tau} &= [D^I_{\sigma_1} \to D^I_{\sigma_2}]. \end{aligned}$$

Note that the interpretations of types are no longer induced by the interpretations of the base types, since complete freedom is allowed for the interpretation of list types. In practice we expect some uniformity in the interpretation of list types. One way of imposing this would be to require some functor L^I over the category of domains, equipped with some additional structure (e.g. that of a monad), such that $D^I_{list(\sigma)} = L^I(D^I_\sigma)$. We have not done this since we have yet to identify a structure which is both useful and of sufficient generality to incorporate the standard interpretation and the abstract interpretations of interest.

The definitions of environments and the semantic valuation functions $[\![\_]\!]^I$ remain unaltered.

5.2.2 Logical Relations

Logical relations over $\mathcal{T}^l$ are defined exactly as before. As in the case of type interpretations, this means that a logical relation is no longer uniquely determined by its base type instances.

Concretisation maps are derived from logical relations in the same way as before, but to ensure that these maps always return complete pers requires more effort: see Subsection 5.2.3 below. The Logical Concretisation Map Theorem clearly remains valid, and the definition of logical concretisation map is retained, though of course a logical concretisation map is no longer induced by its base type members.

Correctness of an abstract interpretation is defined in the same way as before (Definition 4.1.4). The Ternary Logical Relations Theorem (3.3.2) remains valid because the proof is by induction over the structure of terms, which are as before. Hence the Correctness Theorem also remains valid.

Remark It may seem as though we are getting something for nothing: we have added list types without making any requirement that logical relations respect them, yet we are guaranteed that alternative interpretations of terms of list type will always be related. The reason is that the only way we have to construct such terms is via constants of list type, whose interpretations are related *by hypothesis.* Thus the extra work which we might expect to be involved in proving correctness for list types, is simply postponed until we define and prove correct the interpretations of these constants.

5.2.3 Inherited Properties

Inheritance of properties is profoundly affected by the addition of list types. In fact, since logical relations are no longer induced from their base type instances, we can only expect trivial properties to be inherited. This puts a bit of a hole in the framework. As a partial repair, we adopt the following definition.

Definition 5.2.1 *A property* $P = \{P_\sigma\}_{\sigma \in \mathcal{T}^l}$ *of* $\mathcal{T}^l$*-indexed families of relations, is* conditionally inherited *if for all ternary logical relations* $R : I \times I' \leftrightarrow J$*:* if *the condition*

$$\forall \sigma \in \mathcal{T}^l.\, P_\sigma(R_\sigma) \Rightarrow P_{list(\sigma)}(R_{list(\sigma)})$$

holds, then

$$(\forall \iota \in \mathcal{T}_0.\, P_\iota(R_\iota)) \Rightarrow \forall \sigma \in \mathcal{T}^l.\, P_\sigma(R_\sigma).$$

It is routinely verified that Lemma 3.5.2 remains valid for logical relations over $\mathcal{T}^l$ if 'inherited' is replaced throughout by 'conditionally inherited'. Furthermore, the proofs of Proposition 3.4.3 and Proposition 4.2.5 can be easily modified to show that in the current setting, the property CP and preservation of meets by (hence the existence of left adjoints for) the concretisation maps are conditionally inherited. Of course, to make use of the fact that a property is conditionally inherited when reasoning about any given R, we will have to show both that the property holds for the R_ι and that the condition specified by Definition 5.2.1 is actually met.

5.2.4 List Constants and the Standard Interpretation

The constants we assume for lists are, for each $\sigma, \tau \in \mathcal{T}^l$:

- $\mathbf{nil}_\sigma : list(\sigma)$;

$$D^{\mathrm{s}}_{list(\sigma)} = list(D^{\mathrm{s}}_{\sigma})$$

$$\begin{aligned}
&\mathbf{nil}^{\mathrm{s}}_{\sigma} = [\,] \\
&\mathbf{cons}^{\mathrm{s}}_{\sigma}\ x\ xs = x : xs \\
&\mathbf{case}^{\mathrm{s}}_{\sigma,\tau}\ l\ f\ d = \begin{cases} \perp^{\mathrm{s}}_{\tau} & \text{if } l = \perp \\ f\ x\ xs & \text{if } l = x : xs \\ d & \text{if } l = [\,] \end{cases}
\end{aligned}$$

Figure 5.3: Extensions to the Standard Interpretation

- $\mathbf{cons}_{\sigma} : \sigma \to list(\sigma) \to list(\sigma)$;
- $\mathbf{case}_{\sigma,\tau} : list(\sigma) \to (\sigma \to list(\sigma) \to \tau) \to \tau \to \tau$.

For any domain D, the domain $list(D)$ is defined to be the usual domain of partial, finite and infinite lists of elements in D, obtained as an initial solution to the domain equation

$$list(D) \cong \mathbf{1} + (D \times list(D))$$

where $+$ is separated sum. (See [SP82] and the next chapter.) We will write the isomorphic images of $in_1(\cdot)$ and $in_2(d, l)$ in $list(D)$ as $[\,]$ and $d : l$ respectively.

Extensions to the standard interpretation $\mathbf{S}$ of Chapter 2 to incorporate list types and the new constants are shown in Figure 5.3.

5.3 A Head-Strictness Analysis

In [WH87] Phil Wadler and John Hughes described a strictness analysis technique for first-order functional languages. Two features distinguish this technique from strictness analyses in the [Myc81, BHA86] style:

1. it uses a non-standard *backwards* semantics;
2. it uses *projections* to formalise strictness properties rather than Scott-closed sets.

The way projections are used to describe strictness properties is via the equation (3.1.2) introduced in Chapter 3:

$$\beta \circ f = \beta \circ f \circ \alpha.$$

By using projections in this way, Wadler and Hughes were able to capture a novel form of strictness, known as *head-strictness*. This is defined in terms of the projection H_D:

Definition 5.3.1 *For each domain D, the projection $\mathrm{H}_D : list(D) \to list(D)$ is defined by*

$$\begin{array}{lcl} \mathrm{H}_D(\perp) & = & \perp \\ \mathrm{H}_D([\,]) & = & [\,] \\ \mathrm{H}_D(d:l) & = & \begin{cases} \perp & \text{if } d = \perp_D \\ d : \mathrm{H}_D(l) & \text{otherwise.} \end{cases} \end{array}$$

For $\sigma \in \mathcal{T}^l$ we will write H_σ to mean $\mathrm{H}_{D^{\mathrm{S}}_\sigma}$.

A function $f : list(A) \to B$ is said to be *head-strict* if $f = f \circ \mathrm{H}_A$, which put into form (3.1.2) is:

$$\mathrm{Id}_B \circ f = \mathrm{Id}_B \circ f \circ \mathrm{H}_A.$$

Intuitively, a head-strict function is one which never 'looks at' a tail of its argument without looking at the head of that tail. An example is given at the end of this section. By Proposition 3.1.5, it is equivalent to define a head-strict function as one for which

$$f : (\ker \mathrm{H}_A) \Rightarrow Id_B.$$

This enables us to construct an analysis for head-strictness based on complete pers. There would appear to be no equivalent definition of head-strictness using sets in the form $f(X) \subseteq Y$, so the [BHA86, Abr90] frameworks are unable to capture head-strictness. (In [Bur90], Burn investigates what common ground there *is* between [BHA86] and [WH87].)

The interest of head-strictness, as suggested in [WH87], is as follows. Let $e_1 : list(\sigma) \to \tau$ and $e_2 : list(\sigma)$ be closed terms. Suppose we were able to establish that the function $[\![e_1]\!]^{\mathrm{S}}$ satisfied the property

$$[\![e_1]\!]^{\mathrm{S}} = [\![e_1]\!]^{\mathrm{S}} \circ \mathrm{H}_\sigma,$$

so that $[\![e_1\ e_2]\!]^{\mathrm{S}} = [\![e_1]\!]^{\mathrm{S}}(\mathrm{H}_\sigma([\![e_2]\!]^{\mathrm{S}}))$. We could then modify the code generation for the sub-expression e_2 to implement every call to $\mathbf{cons}_\sigma$ which contributed to the output of e_2 by a left-strict cons operation, thus avoiding the need to build and maintain closures and increasing the opportunities for parallelism.

It is not quite so easy to describe ordinary strictness in terms of projections. The approach taken in [WH87] involves lifting all the domains and functions involved:

given a continuous map $f : A \to B$, the *lifting* of f is $f_\perp : A_\perp \to B_\perp$, given by

$$f_\perp\ x = \begin{cases} \perp & \text{if } x = \perp \\ \mathit{lift}(f\ a) & \text{if } x = \mathit{lift}(a). \end{cases}$$

For each domain A the projection $\mathrm{Str}_A : A_\perp \to A_\perp$ is defined by

$$\mathrm{Str}_A\ x = \begin{cases} \perp & \text{if } x = \perp \\ \perp & \text{if } x = \mathit{lift}(\perp_A) \\ \mathit{lift}(a) & \text{if } x = \mathit{lift}(a), a \neq \perp_A. \end{cases}$$

It is then easily seen that $f : A \to B$ is strict if and only if

$$\mathrm{Str}_B \circ f_\perp = \mathrm{Str}_B \circ f_\perp \circ \mathrm{Str}_A.$$

Here we avoid the need for lifting by exploiting the fact that unlike projections, pers can be partial: we can use the per *Bot* to describe strictness, since $f : A \to B$ is strict if and only if

$$f : \mathit{Bot}_A \Rightarrow \mathit{Bot}_B.$$

However, one of the most pleasing aspects of the [WH87] approach is the natural way in which projections over the lifted domains can be combined to capture a rich family of strictness properties over recursively defined types. Our approach does not seem to generalise in the same way. We return to this point at the end of the next chapter.

5.3.1 The Abstract Domains and Concretisation Maps

We define an interpretation **H** (H for Head-strictness) with the following lattice interpretations for base types and list types:

$$D_\iota^{\mathbf{H}} = \mathbf{3} = \begin{array}{c} \mathrm{D} \\ | \\ \mathrm{S} \\ | \\ \mathrm{B} \end{array} \qquad D_{list(\sigma)}^{\mathbf{H}} = \mathbf{4} = \begin{array}{c} \mathrm{D} \\ | \\ \mathrm{H} \\ | \\ \mathrm{S} \\ | \\ \mathrm{B} \end{array}$$

Product and function type interpretations are induced in the usual way.

The intention is that D and S be interpreted as in the constancy analysis, that B should correspond to *Bot* and that H should correspond to $\ker \mathrm{H}$. Thus for base types we define

$$\gamma^{\mathbf{H}}_{\iota}\ \mathrm{B} = Bot_{\iota} \quad \gamma^{\mathbf{H}}_{\iota}\ \mathrm{S} = Id_{\iota} \quad \gamma^{\mathbf{H}}_{\iota}\ \mathrm{D} = All_{\iota},$$

and for list types we define

$$\gamma^{\mathbf{H}}_{list(\sigma)}\ \mathrm{B} = Bot_{list(\sigma)} \quad \gamma^{\mathbf{H}}_{list(\sigma)}\ \mathrm{S} = Id_{list(\sigma)} \quad \gamma^{\mathbf{H}}_{list(\sigma)}\ \mathrm{H} = \ker \mathrm{H}_{\sigma} \quad \gamma^{\mathbf{H}}_{list(\sigma)}\ \mathrm{D} = All_{list(\sigma)}.$$

This is extended to the logical concretisation map $\gamma^{\mathbf{H}} = \{\gamma^{\mathbf{H}}_{\sigma}\}_{\sigma\in\mathcal{T}^l}$ via the Logical Concretisation Map Theorem. The associated logical relation is called $R^{\mathbf{H}}$. Note that our interpretation of $list(\sigma)$ is rather crude: we are always restricted to the same four properties, regardless of what σ is. Each $\gamma^{\mathbf{H}}_{\sigma}$ is:

- CP;
- meet preserving;
- strict.

Recall from Section 3.5 that to say that $\gamma^{\mathbf{H}}_{\sigma}$ is strict is equivalent to saying that $R^{\mathbf{H}}_{\sigma}$ is strict and $\perp$-reflecting, and that $\gamma^{\mathbf{H}}_{\sigma}$ is CP implies that $R^{\mathbf{H}}_{\sigma}$ is strict. Thus it remains to show that each $\gamma^{\mathbf{H}}_{\sigma}$ is CP and meet preserving and that each $R^{\mathbf{H}}_{\sigma}$ is $\perp$-reflecting. Furthermore, if $\gamma^{\mathbf{H}}_{\sigma}$ preserves meets then it preserves $\top$, which implies that $R^{\mathbf{H}}_{\sigma}$ is $\top$-universal. It follows that as a corollary of Propositions 3.4.3 and 4.2.5, and Lemma 3.5.2 (modified as described in Subsection 5.2.3), the combination of $\gamma^{\mathbf{H}}_{\sigma}$ being CP, meet preserving and strict, is conditionally inherited. Thus we must show two things:

1. each $\gamma^{\mathbf{H}}_{\iota}$ is CP, meet preserving and strict;
2. for each $\sigma \in \mathcal{T}^l$, if $\gamma^{\mathbf{H}}_{\sigma}$ is CP then so is $\gamma^{\mathbf{H}}_{list(\sigma)}$, and similarly for meet preservation and strictness.

It is immediate from the definitions that both these conditions are met (in fact, $\gamma^{\mathbf{H}}_{list(\sigma)}$ is CP, meet preserving and strict independently of any properties of $\gamma^{\mathbf{H}}_{\sigma}$).

5.3.2 Testing For Strictness and Head-Strictness

To test a function definition for head-strictness we will show there to be an element $\mathrm{S}_{\tau} \in D^{\mathbf{H}}_{\tau}$ for each $\tau \in \mathcal{T}^l$, such that $\gamma^{\mathbf{H}}_{\tau}\ \mathrm{S}_{\tau} = Id_{\tau}$. Then given correct interpretations for the constants we can analyse a closed term $e : list(\sigma) \to \tau$ for head-strictness by checking whether:

$$[\![e]\!]^{\mathbf{H}}\ \mathrm{H} = \mathrm{S}_{\tau}.$$

The family $\{\mathrm{S}_\tau\}_{\tau \in T^l}$ is inductively defined by:

- $\mathrm{S}_\iota = \mathrm{S}$
- $\mathrm{S}_{\sigma\times\tau} = (\mathrm{S}_\sigma, \mathrm{S}_\tau)$
- $\mathrm{S}_{list(\sigma)} = \mathrm{S}$
- $\mathrm{S}_{\sigma\to\tau} = \lfloor \mathrm{S}_\sigma, \mathrm{S}_\tau \rfloor$

The only non-obvious case is $\mathrm{S}_{\sigma\to\tau}$. This is justified by the conjunction of Proposition 3.4.5, which says that $\gamma^{\mathbf{H}}_{\sigma\to\tau}\ \lfloor \mathrm{S}_\sigma, \mathrm{S}_\tau \rfloor = \gamma^{\mathbf{H}}_\sigma\ \mathrm{S}_\sigma \Rightarrow \gamma^{\mathbf{H}}_\tau\ \mathrm{S}_\tau$, and Proposition 3.2.4, which says that $Id_\sigma \Rightarrow Id_\tau = Id_{\sigma\to\tau}$.

To test a function definition for strictness we use the fact that each $\gamma^{\mathbf{H}}_\sigma$ is strict. So for $f : \gamma^{\mathbf{H}}_{\sigma_1\to\cdots\to\sigma_n\to\tau}\ h$:

$$h\ \mathrm{S}_{\sigma_1} \cdots \mathrm{S}_{\sigma_{i-1}}\ \perp\ \mathrm{S}_{\sigma_{i+1}} \cdots \mathrm{S}_{\sigma_n} = \perp$$

implies that

$$f : Id_{\sigma_1} \Rightarrow \cdots \Rightarrow Id_{\sigma_{i-1}} \Rightarrow Bot \Rightarrow Id_{\sigma_{i+1}} \Rightarrow \cdots \Rightarrow Id_{\sigma_n} \Rightarrow Bot,$$

which says that f is strict in its ith argument.

5.3.3 The Interpretations of Constants

The interpretations of the constants are shown in Figure 5.4. Most superscripts and subscripts are omitted. We do not prove correctness of all the constants in detail, concentrating only on the more interesting and less obvious cases.

Arithmetic Operators

These are justified essentially as in the constancy analysis together with the fact that they are each strict in both arguments. For example, $\mathbf{plus}^{\mathrm{S}} \perp n = \perp = \mathbf{plus}^{\mathrm{S}} \perp n'$ for any $n, n' \in \mathbf{Z}$, hence $\mathbf{plus}^{\mathrm{S}} : Bot \Rightarrow P \Rightarrow Bot$ for any P.

Conditionals

Consider $\mathbf{if}^{\mathbf{H}}_\sigma\ a\ b_1\ b_2$. For $a = \mathrm{B}$ the definition is justified by the fact that $\mathbf{if}^{\mathrm{S}}_\sigma$ is strict in its first argument. For $a = \mathrm{S}$ the argument is as for $\mathbf{if}^{\mathrm{C}}_\sigma$ above. For $a = \mathrm{D}$

$$\mathbf{n}^{\mathrm{H}} = \mathrm{S}$$

$$\mathbf{true}^{\mathrm{H}} = \mathbf{false}^{\mathrm{H}} = \mathrm{S}$$

$$\mathbf{iszero}^{\mathrm{H}}\ a = a$$

$$\mathbf{plus}^{\mathrm{H}}\ a\ b = \begin{cases} \mathrm{B} & \text{if } a = \mathrm{B} \text{ or } b = \mathrm{B} \\ a \sqcup b & \text{otherwise} \end{cases}$$

$$\mathbf{minus}^{\mathrm{H}} = \mathbf{mult}^{\mathrm{H}} = \mathbf{plus}^{\mathrm{H}}$$

$$\mathbf{if}_{\sigma}^{\mathrm{H}}\ a\ b_1\ b_2 = \begin{cases} \bot & \text{if } a = \mathrm{B} \\ b_1 \sqcup b_2 & \text{if } a = \mathrm{S} \\ \bot & \text{if } a = \mathrm{D} \text{ and } b_1 = \bot \text{ and } b_2 = \bot \\ \top & \text{if } a = \mathrm{D}, \text{ otherwise} \end{cases}$$

$$\mathbf{nil}_{\sigma}^{\mathrm{H}} = \mathrm{S}$$

$$\mathbf{cons}_{\sigma}^{\mathrm{H}}\ a\ b = \begin{cases} \mathrm{S} & \text{if } b \sqsubseteq \mathrm{S}, a \sqsubseteq \mathrm{S}_{\sigma} \\ \mathrm{H} & \text{if } b \not\sqsubseteq \mathrm{S}, a = \bot \\ b & \text{if } b \not\sqsubseteq \mathrm{S}, a \neq \bot, a \sqsubseteq \mathrm{S}_{\sigma} \\ \mathrm{D} & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\mathbf{case}_{\sigma,\tau}^{\mathrm{H}}\ \mathrm{B}\ h\ b &= \bot \\
\mathbf{case}_{\sigma,\tau}^{\mathrm{H}}\ \mathrm{S}\ h\ b &= b \sqcup (h\ \mathrm{S}_{\sigma}\ \mathrm{S}) \\
\mathbf{case}_{\sigma,\tau}^{\mathrm{H}}\ \mathrm{H}\ h\ b &= \begin{cases} b \sqcup (h\ \mathrm{S}_{\sigma}\ \mathrm{H}) & \text{if } h \perp \mathrm{S} = \bot \\ \top & \text{otherwise} \end{cases} \\
\mathbf{case}_{\sigma,\tau}^{\mathrm{H}}\ \mathrm{D}\ h\ b &= \begin{cases} \bot & \text{if } b = \bot \text{ and } h = \bot \\ \top & \text{otherwise} \end{cases}
\end{aligned}$$

$$\mathbf{Y}_{\sigma}^{\mathrm{H}}\ f = \bigsqcup_{i \in \omega} f^i\ \bot$$

Figure 5.4: Interpretations Of Constants for Head-Strictness Analysis

and $b_1 = b_2 = \mathrm{B}$ the definition is justified by the fact that $\mathbf{if}^{\mathrm{S}}_{\sigma}$ is *jointly* strict in its second and third arguments, i.e.,

$$\mathbf{if}^{\mathrm{S}}_{\sigma}\ d\ \bot\ \bot = \bot,$$

for any $d \in \mathbf{B}$. In the remaining cases $\mathbf{if}^{\mathbf{H}}_{\sigma}\ a\ b_1\ b_2 = \top$, which is correct since $\gamma^{\mathbf{H}}_{\sigma}(\top) = All_{\sigma}$. (John Hughes has pointed out that this definition is open to improvement: for example, for any $a, b \in D^{\mathbf{H}}_{\sigma}$ it would be correct to take

$$\mathbf{if}_{\sigma\times\tau}\ \mathrm{D}\ (a, \bot)\ (b, \bot) = (\top, \bot).$$

Although we have not done so, it is not too hard to see how the definition could be modified to take account of this observation.)

List Constructors

The correctness of $\mathbf{cons}^{\mathbf{H}}_{\sigma}$ is demonstrated by showing for all a and b, that $d\ (\gamma^{\mathbf{H}}_{\sigma}\ a)\ d'$ *and* $l\ (\gamma^{\mathbf{H}}_{list(\sigma)}\ b)\ l'$ implies that $d : l\ (\gamma^{\mathbf{H}}_{list(\sigma)}(\mathbf{cons}^{\mathbf{H}}_{\sigma}\ a\ b))\ d' : l'$. This can be done by a case analysis following the four clauses in the definition of $\mathbf{cons}^{\mathbf{H}}_{\sigma}$. We consider the first two.

Recall that $\gamma^{\mathbf{H}}_{\tau}\ \mathrm{S}_{\tau} = Id_{\tau}$ and $\gamma^{\mathbf{H}}_{\tau}$ is monotone for all τ. Suppose that $b \sqsubseteq \mathrm{S}$ and $a \sqsubseteq \mathrm{S}_{\sigma}$. Then $l\ (\gamma^{\mathbf{H}}_{list(\sigma)}\ b)\ l'$ *and* $d\ (\gamma^{\mathbf{H}}_{\sigma}\ a)\ d'$ implies that $l = l'$ and $d = d'$, and hence that $d : l\ (\gamma^{\mathbf{H}}_{list(\sigma)}\ \mathrm{S})\ d' : l'$.

Now suppose that $b \not\sqsubseteq \mathrm{S}$ but that $a = \bot$. Since $\gamma^{\mathbf{H}}_{\sigma}$ is strict, we know that $d\ (\gamma^{\mathbf{H}}_{\sigma}\ \bot)\ d' \iff d = d' = \bot$. Now for any l, l' we have $\mathrm{H}_{\sigma}(\bot : l) = \bot = \mathrm{H}_{\sigma}(\bot : l')$. Thus $d\ (\gamma^{\mathbf{H}}_{\sigma}\ \bot)\ d'$ *and* $l\ (\gamma^{\mathbf{H}}_{list(\sigma)}\ b)\ l'$ implies that $d : l\ (\ker \mathrm{H}_{\sigma})\ d' : l'$.

Case Constants

We consider only the second and third clauses in the definition of $\mathbf{case}^{\mathbf{H}}_{\sigma,\tau}$, the other two being routine.

For the second clause, assume that $f\ (\gamma^{\mathbf{H}}_{\sigma\to list(\sigma)\to\tau}\ h)\ f'$ and $d\ (\gamma^{\mathbf{H}}_{\tau}\ b)\ d'$. Since $\gamma^{\mathbf{H}}_{list(\sigma)}\ \mathrm{S} = Id_{list(\sigma)}$, we must show that $(\mathbf{case}^{\mathrm{S}}_{\sigma,\tau}\ l\ f\ d)$ and $(\mathbf{case}^{\mathrm{S}}_{\sigma,\tau}\ l\ f'\ d')$ are related by $\gamma^{\mathbf{H}}_{\tau}(b \sqcup (h\ \mathrm{S}_{\sigma}\ \mathrm{S}))$, for all l. There are three cases for l:

1. $l = \bot$. Then $\mathbf{case}^{\mathrm{S}}_{\sigma,\tau}\ \bot\ f\ d = \bot = \mathbf{case}^{\mathrm{S}}_{\sigma,\tau}\ \bot\ f'\ d'$,and $\bot\ P\ \bot$ for any complete per P.

2. $l = [\,]$. Then $\mathbf{case}^{\mathrm{S}}_{\sigma,\tau}\ [\,]\ f\ d = d$ and $\mathbf{case}^{\mathrm{S}}_{\sigma,\tau}\ [\,]\ f'\ d' = d'$. Since $\gamma^{\mathbf{H}}_{\tau}$ is monotone,

$d\ (\gamma^{\mathbf{H}}_{\tau}\ b)\ d'$ implies that $d\ (\gamma^{\mathbf{H}}_{\tau}(b \sqcup (h\ \mathrm{S}_{\sigma}\ \mathrm{S})))\ d'$.

3. $l = x : xs$. Then $\mathbf{case}^{\mathrm{S}}_{\sigma,\tau}\ (x : xs)\ f\ d = f\ x\ xs$ and $\mathbf{case}^{\mathrm{S}}_{\sigma,\tau}\ (x : xs)\ f'\ d' = f'\ x\ xs$. Now by the Logical Concretisation Map Theorem, $f\ (\gamma^{\mathbf{H}}_{\sigma \to list(\sigma) \to \tau}\ h)\ f'$ implies that
$$f\ (\gamma^{\mathbf{H}}_{\sigma}\ \mathrm{S}_{\sigma} \Rightarrow \gamma^{\mathbf{H}}_{list(\sigma)}\ \mathrm{S} \Rightarrow \gamma^{\mathbf{H}}_{\tau}(h\ \mathrm{S}_{\sigma}\ \mathrm{S}))\ f'.$$
Thus, $(f\ x\ xs)\ (\gamma^{\mathbf{H}}_{\tau}(h\ \mathrm{S}_{\sigma}\ \mathrm{S}))\ (f'\ x\ xs)$, since $x\ Id_{\sigma}\ x$ and $xs\ Id_{list(\sigma)}\ xs$. Again, monotonicity of $\gamma^{\mathbf{H}}_{\tau}$ finishes the job.

For the third clause, assume that $l\ (\gamma^{\mathbf{H}}\ \mathrm{H})\ l'$ (which is to say that $\mathrm{H}\ l = \mathrm{H}\ l'$), that $f\ (\gamma^{\mathbf{H}}\ h)\ f'$ and that $d\ (\gamma^{\mathbf{H}}\ b)\ d'$. The interesting case is when $h \perp \mathrm{S} = \perp$, so assume this holds. We must show that $(\mathbf{case}^{\mathrm{S}}_{\sigma,\tau}\ l\ f\ d)$ and $(\mathbf{case}^{\mathrm{S}}_{\sigma,\tau}\ l'\ f'\ d')$ are related by $\gamma^{\mathbf{H}}_{\tau}(b \sqcup (h\ \mathrm{S}_{\sigma}\ \mathrm{H}))$. A simple case analysis based on the definition of H shows there to be six ways in which $\mathrm{H}\ l = \mathrm{H}\ l'$ can be true. We consider each of these in turn.

1. $l = l' = \perp$. See case 1 in the argument for the second clause above.

2. $l = l' = [\,]$. See case 2 in the argument for the second clause.

3. $l = x : xs$ and $l' = x : xs'$, with $\mathrm{H}\ xs = \mathrm{H}\ xs'$. Essentially as for case 3 in the argument for the second clause, except that instead of $xs\ Id_{list(\sigma)}\ xs$, we only have $xs\ (\ker \mathrm{H}_{\sigma})\ xs'$.

4. $l = \perp : xs$ and $l' = \perp$. Since $h \perp \mathrm{S} = \perp$ and $f : \gamma^{\mathbf{H}}\ h$, we know that f is strict in its first argument, so $\mathbf{case}^{\mathrm{S}}_{\sigma,\tau}\ (\perp : xs)\ f\ d = f \perp xs = \perp = \mathbf{case}^{\mathrm{S}}_{\sigma,\tau} \perp f'\ d'$, and $\perp P \perp$ for any complete per P.

5. $l' = \perp : xs$ and $l = \perp$. Symmetric with the previous case.

6. $l = \perp : xs$ and $l' = \perp : xs'$. Essentially as for the previous two cases.

Recursion Combinators

H is a least fixed point interpretation so correctness of $\mathbf{Y}^{\mathbf{H}}_{\sigma}$ is given by Corollary 4.1.9.

5.3.4 An Example Analysis for Head Strictness

Let the functions $tbz^{\mathrm{S}} \in [list(\mathbf{Z}) \rightarrow list(\mathbf{Z})]$ and $tbz^{\mathbf{H}} \in [\mathbf{4} \rightarrow \mathbf{4}]$ be the standard and **H** interpretations, respectively, of the following term:

$$\begin{array}{l} \mathbf{Y}\ \lambda\, \mathtt{tbz}\,.\,\lambda\, \mathtt{l}\,.\,\mathbf{case}\ \mathtt{l} \\ \qquad\qquad \lambda\, \mathtt{n}\,.\,\lambda\, \mathtt{ns}\,.\,\mathbf{if}\ (\mathbf{iszero}\ \mathtt{n}) \\ \qquad\qquad\qquad\qquad \mathbf{nil} \\ \qquad\qquad\qquad\qquad \mathbf{cons}\ \mathtt{n}\ (\mathtt{tbz}\ \mathtt{ns}) \\ \qquad\qquad \mathbf{nil} \end{array}$$

The standard interpretation tbz^{S} is the function which returns its argument truncated just before the first zero element. Intuitively, it is clear that this function is head-strict, because whenever it looks at a tail of its argument it also looks at the head of that tail to see if it's zero. To use the abstract interpretation $tbz^{\mathbf{H}}$ to confirm this we must show that

$$tbz^{\mathbf{H}}\ \mathrm{H} = \mathrm{S}.$$

Since **H** is a least fixed point interpretation, $tbz^{\mathbf{H}}$ is the limit of the sequence $\{tbz^{\mathbf{H}}_0, tbz^{\mathbf{H}}_1, \ldots\}$, generated by

$$\begin{array}{rcl} tbz^{\mathbf{H}}_0\ \ b & = & \mathrm{B} \\ tbz^{\mathbf{H}}_{n+1}\ b & = & \mathbf{case}^{\mathbf{H}}\ b \\ & & \quad \lambda\, a \in \mathbf{2}\,.\,\lambda\, b' \in \mathbf{4}\,.\,\mathbf{if}^{\mathbf{H}}\ a\ \mathrm{S}\ (\mathbf{cons}^{\mathbf{H}}\ a\ (tbz^{\mathbf{H}}_n\ b')) \\ & & \quad \mathrm{S} \end{array}$$

(we use the fact that $\mathbf{iszero}^{\mathbf{H}}\ a = a$ and $\mathbf{nil}^{\mathbf{H}} = \mathrm{S}$). This sequence must converge to its limit after a finite number of terms since the lattice $[\mathbf{4} \rightarrow \mathbf{4}]$ is of finite height.

In an automated analysis the limit could be calculated explicitly by tabulating successive iterates until two were found to be equal (see Chapters 7–9). Since that would be rather tedious to do by hand, we will be more economical and show that $tbz^{\mathbf{H}}\ \mathrm{H} = \mathrm{S}$ by showing (by induction on n) that $tbz^{\mathbf{H}}_{n+1}\ \mathrm{H} = \mathrm{S}$ for all n. For $n = 0$ we have $tbz^{\mathbf{H}}_1\ \mathrm{H} = \mathbf{case}^{\mathbf{H}}\ \mathrm{H}\ h\ \mathrm{S}$, where

$$\begin{array}{rcl} h & = & \lambda\, a \in \mathbf{2}\,.\,\lambda\, b' \in \mathbf{4}\,.\,\mathbf{if}^{\mathbf{H}}\ a\ \mathrm{S}\ (\mathbf{cons}^{\mathbf{H}}\ a\ (tbz^{\mathbf{H}}_0\ b')) \\ & = & \lambda\, a \in \mathbf{2}\,.\,\lambda\, b' \in \mathbf{4}\,.\,\mathbf{if}^{\mathbf{H}}\ a\ \mathrm{S}\ (\mathbf{cons}^{\mathbf{H}}\ a\ \mathrm{B}). \end{array}$$

Now $h\ \text{B}\ \text{S} = \mathbf{if}^{\mathbf{H}}\ \text{B}\ \text{S}\ (\mathbf{cons}^{\mathbf{H}}\ \text{B}\ \text{B}) = \text{B}$, and

$$\begin{aligned} h\ \text{S}\ \text{H} &= \mathbf{if}^{\mathbf{H}}\ \text{S}\ \text{S}\ (\mathbf{cons}^{\mathbf{H}}\ \text{S}\ \text{B}) \\ &= \text{S} \sqcup (\mathbf{cons}^{\mathbf{H}}\ \text{S}\ \text{B}) \\ &= \text{S} \sqcup \text{S} \\ &= \text{S}. \end{aligned}$$

Hence $\mathbf{case}^{\mathbf{H}}\ \text{H}\ h\ \text{S} = \text{S} \sqcup \text{S} = \text{S}$. For $n = k+1$ we have $tbz^{\mathbf{H}}_{k+2}\ \text{H} = \mathbf{case}^{\mathbf{H}}\ \text{H}\ h'\ \text{S}$, where

$$h' = \lambda\, a \in \mathbf{2}\,.\,\lambda\, b' \in \mathbf{4}\,.\,\mathbf{if}^{\mathbf{H}}\ a\ \text{S}\ (\mathbf{cons}^{\mathbf{H}}\ a\ (tbz^{\mathbf{H}}_{k+1}\ b')).$$

Then $h'\ \text{B}\ \text{S} = \mathbf{if}^{\mathbf{H}}\ \text{B}\ \text{S}\ (\mathbf{cons}^{\mathbf{H}}\ \text{B}\ (tbz^{\mathbf{H}}_{k+1}\ \text{S})) = \text{B}$, and

$$\begin{aligned} h'\ \text{S}\ \text{H} &= \mathbf{if}^{\mathbf{H}}\ \text{S}\ \text{S}\ (\mathbf{cons}^{\mathbf{H}}\ \text{S}\ (tbz^{\mathbf{H}}_{k+1}\ \text{H})) \\ &= \text{S} \sqcup (\mathbf{cons}^{\mathbf{H}}\ \text{S}\ \text{S}) \quad \text{(by induction hypothesis)} \\ &= \text{S} \sqcup \text{S} \\ &= \text{S}. \end{aligned}$$

Hence $\mathbf{case}^{\mathbf{H}}\ \text{H}\ h'\ \text{S} = \text{S} \sqcup \text{S} = \text{S}$.

Chapter 6

Recursive Types

In this chapter we consider the implications of enriching our language with recursively defined types. We are able to give meaning to recursively described properties for recursively defined types, adapting the method of [Lau89] to the world of pers and higher-order functions. However, for recursive types which use $\rightarrow$ in an unrestricted way, we are unable (because of contravariance problems) to give a construction for finite lattices of such properties suitable for use in abstract interpretation.

6.1 Enriching the Languages of Types and Terms

We extend the language of types by introducing the unit type, sum types and recursive types. To simplify the exposition we allow only one type variable in recursive definitions and stratify the type system to distinguish between bodies of recursive definitions, which may contain free occurrences of the type variable, and ordinary types, which are always closed.

$$\sigma \in \mathcal{T}^{\mu} \ ::= \iota \mid \sigma_1 \times \sigma_2 \mid \sigma_1 \rightarrow \sigma_2 \mid 1 \mid \sigma_1 + \sigma_2 \mid \mu\alpha \,.\, \sigma^{\alpha}$$

$$\sigma^{\alpha} \in \mathcal{T}^{\alpha} \ ::= \sigma \mid \sigma_1^{\alpha} \times \sigma_2^{\alpha} \mid \sigma_1^{\alpha} \rightarrow \sigma_2^{\alpha} \mid \sigma_1^{\alpha} + \sigma_2^{\alpha} \mid \alpha$$

The extended language of terms $\Lambda_{\mathcal{T}^{\mu}}$ is shown in Figure 6.1 (the syntax is taken from [Gun91]). The typing rules are as for $\Lambda_{\mathcal{T}}$ together with the additional rules shown in Figure 6.2.

$$
\begin{aligned}
e \in \Lambda_{\mathcal{T}^\mu} \;::=\; & x \mid c \mid \lambda\, x\,.\, e \mid e_1\; e_2 \mid () \mid \\
& (e_1, e_2) \mid \mathbf{fst}(e) \mid \mathbf{snd}(e) \mid \\
& \mathbf{inl}_{\sigma_1,\sigma_2}(e) \mid \mathbf{inr}_{\sigma_1,\sigma_2}(e) \mid \mathbf{case}\ e\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_1\ \mathbf{or}\ \mathbf{inr}(y) \Rightarrow e_2 \mid \\
& \mathbf{fold}_{\sigma^\alpha}(e) \mid \mathbf{unfold}_{\sigma^\alpha}(e)
\end{aligned}
$$

Figure 6.1: The Language of Terms $\Lambda_{\mathcal{T}^\mu}$

$$() : 1$$

$$\frac{e : \sigma_1}{\mathbf{inl}_{\sigma_1,\sigma_2}(e) : \sigma_1 + \sigma_2} \qquad \frac{e : \sigma_2}{\mathbf{inr}_{\sigma_1,\sigma_2}(e) : \sigma_1 + \sigma_2}$$

$$\frac{e : \sigma_1 + \sigma_2 \qquad e_1 : \tau \qquad e_2 : \tau}{\mathbf{case}\ e\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_1\ \mathbf{or}\ \mathbf{inr}(y) \Rightarrow e_2 : \tau}\ \text{ if } x \in Var_{\sigma_1}, y \in Var_{\sigma_2}$$

$$\frac{e : \sigma^\alpha[\alpha \mapsto \mu\alpha\,.\,\sigma^\alpha]}{\mathbf{fold}_{\sigma^\alpha}(e) : \mu\alpha\,.\,\sigma^\alpha} \qquad \frac{e : \mu\alpha\,.\,\sigma^\alpha}{\mathbf{unfold}_{\sigma^\alpha}(e) : \sigma^\alpha[\alpha \mapsto \mu\alpha\,.\,\sigma^\alpha]}$$

Figure 6.2: The Additional Typing Rules for $\Lambda_{\mathcal{T}^\mu}$

6.2 The Interpretation of Unit and Sum Types

In the following sections we outline a method for interpreting recursive types and interpreting terms over such types. For now we just deal with the interpretation of the unit type and sum types. Our notion of interpretation for $\Lambda_{\mathcal{T}^\mu}$ is far less general than that for $\Lambda_{\mathcal{T}}$: we define the standard interpretation and abstract interpretation in essentially different ways.

The types 1 and $\sigma_1 + \sigma_2$ have standard interpretations given by

$$\begin{aligned} D^{\mathrm{s}}_1 &= \mathbf{1} \\ D^{\mathrm{s}}_{\sigma_1+\sigma_2} &= D^{\mathrm{s}}_{\sigma_1} + D^{\mathrm{s}}_{\sigma_2}, \end{aligned}$$

where the $+$ on the right hand side is separated sum.

Any finite interpretation J used for abstract interpretation is assumed to be defined as for $\mathcal{T}$ for function types and product types, and on the unit type and sum types by:

$$\begin{aligned} D^{J}_1 &= \mathbf{1} \\ D^{J}_{\sigma_1+\sigma_2} &= (D^{J}_{\sigma_1} \times D^{J}_{\sigma_2})^{\top}. \end{aligned}$$

The use of product as an abstract interpretation for sum is an idea used in [Nie84] where it is motivated by appeal to the isomorphism $\wp(A+B) \cong \wp(A) \times \wp(B)$. In this setting the most straightforward motivation we can give is to consider the construction described below for concretisation maps at the sum types.

For the concretisation maps $\gamma^J = \left\{\gamma^J_\sigma\right\}_{\sigma\in\mathcal{T}^\mu}$ used in an abstract interpretation J, it is required that each γ^J_σ be a monotone map from D^J_σ to $\mathrm{CPER}(D^{\mathrm{s}}_\sigma)$ and that $\gamma^J_{\sigma\times\tau}$ and $\gamma^J_{\sigma\to\tau}$ are 'logical' as in the Logical Concretisation Map Theorem (3.4.2). It is further required that the definition of γ^J_1 and $\gamma^J_{\sigma_1+\sigma_2}$ be:

$$\begin{aligned} \gamma^J_1(\cdot) &= \mathit{Bot}_{\mathbf{1}} \\ \gamma^J_{\sigma_1+\sigma_2}\ a &= \begin{cases} \mathit{All}_{\sigma_1+\sigma_2} & \text{if } a = \top \\ (\gamma^J_{\sigma_1}\ a_1) + (\gamma^J_{\sigma_2}\ a_2) & \text{if } a = \mathit{colift}(a_1, a_2), \end{cases} \end{aligned}$$

where the sum of two pers $P + Q$ is defined to be the per with graph

$$\{(\bot,\bot)\} \cup \{(\mathit{in}_1(a), \mathit{in}_1(a')) \mid a\ P\ a'\} \cup \{(\mathit{in}_2(b), \mathit{in}_2(b')) \mid b\ Q\ b'\}.$$

What we need now is a way of inducing finite lattices of useful properties over the recursive types. The approach taken in [Lau89] is to base the description of a finite family of properties for a recursive type on the syntax of the type definition. The properties themselves are recursively described and "treat every level of recursion identically". The idea is illustrated by the following example: let $bintree^{\alpha}$ be the $\mathcal{T}^{\alpha}$ type $int + (\alpha \times \alpha)$ and let $bintree$ be the type $\mu\alpha \,.\, bintree^{\alpha}$. Assuming a standard interpretation of this type as a domain of binary trees with integers at the leaves, one obvious property is just $All_{bintree}$. Two other properties of interest are the per Q which relates any two trees having the same structure, and the per S which only relates trees having the same structure *and* the same integers at their leaves. These pers are naturally described recursively:

$$\begin{aligned} Q &= All_{\mathbf{Z}} + (Q \times Q) \\ S &= Id_{\mathbf{Z}} + (S \times S). \end{aligned}$$

Of course S should just be $Id_{bintree}$. In the following sections we show how this example is generalised and how such recursive descriptions can be given meaning. Because our types include function types and $\_ \Rightarrow \_$ is anti-monotone in its first argument, we cannot use a simple least fixed point semantics as in [Lau89]. Instead we use the category-theoretic framework of [SP82]. A further difference between our treatment and Launchbury's is that within our framework we make a clear distinction between the abstract lattices and the lattices of properties to which they correspond, whereas in [Lau89] (and in [WH87]) no such distinction is made. Since concretisation maps are not guaranteed to be injective in the higher-order case, we really have no choice *but* to make the distinction. In fact it is not completely obvious that the syntactic descriptions of projections which [Lau89]'s inference rules assign to a recursive type, and the projections which those descriptions denote, are in one-one correspondence either: the question of whether syntactically distinct descriptions can describe the same projection is not actually discussed, and a disadvantage of de-emphasising the distinction between syntax and semantics is that it becomes rather hard to pose such questions.

6.3 Domains and O-Categories

Let **Dom** be the category of domains and continuous maps and let $\mathbf{Dom}_{\perp}$ be the sub-category of domains and *strict* continuous maps. Both these categories are examples of [SP82]'s **O**-categories when the hom-sets are ordered in the usual (pointwise) way. For any **O**-category **K**, the category of embedding projection pairs $\mathbf{K}^{ep}$

has the same objects as $\mathbf{K}$ and embedding-projection pairs as morphisms, i.e., a morphism $\phi : A \to B$ in $\mathbf{K}^{ep}$ is a pair (ϕ^e, ϕ^p) where $\phi^e : A \to B$ and $\phi^p : B \to A$ are morphisms in $\mathbf{K}$ such that

1. $\phi^p \circ \phi^e = id_A$;

2. $\phi^e \circ \phi^p \sqsubseteq id_B$.

The identity on A in $\mathbf{K}^{ep}$ is just (id_A, id_A). Composition of embedding-projection pairs $\phi : A \to B$ and $\psi : B \to C$ is given by

$$\psi \circ \phi = (\psi^e \circ \phi^e, \phi^p \circ \psi^p).$$

For $\mathbf{Dom}$ and $\mathbf{Dom}_\perp$ we have $\mathbf{Dom}^{ep} = \mathbf{Dom}_\perp^{ep}$.

To interpret a recursively defined type $\mu\alpha\,.\,\sigma^\alpha$, the parameterised type σ^α is interpreted as a functor $F_{\sigma^\alpha} : \mathbf{Dom}^{ep} \to \mathbf{Dom}^{ep}$. The standard interpretation $D^{\mathrm{s}}_{\mu\alpha\,.\,\sigma^\alpha}$ is then obtained as an *initial* solution to the equation

$$F_{\sigma^\alpha}(D) \cong D,$$

using the techniques of [SP82].

6.3.1 Functors on $\mathbf{Dom}_\perp$ and $\mathbf{Dom}^{ep}$

The bi-functors $\times$, $+$ and $\to$ over $\mathbf{Dom}_\perp$ are defined on objects to be, respectively, cartesian product, separated sum and continuous function space. On morphisms they are given by

$$\begin{array}{lcl}
(f \times g)(x_1, x_2) & = & (f(x_1), g(x_2)) \\[2ex]
(f + g)(x) & = & \begin{cases} \perp & \text{if } x = \perp \\ in_1(f(x_1)) & \text{if } x = in_1(x_1) \\ in_2(g(x_2)) & \text{if } x = in_2(x_2) \end{cases} \\[2ex]
(f \to g)(h) & = & g \circ h \circ f,
\end{array}$$

for $f : A \to B$, $g : C \to D$, $h \in [B \to C]$. Note that $\to$ is contravariant in its first argument. In addition to these functors we have the identity functor $id_{\mathbf{Dom}_\perp}$ and for each domain A the constant functor K_A, where $K_A(B) = A$ and $K_A(f) = id_A$ for any object B and morphism f. Using the technique described in [SP82], all these functors can be converted into functors on $\mathbf{Dom}^{ep}$ in such a way that each of them (including $\rightarrow$) becomes covariant. On objects they are as before, on morphisms they become:

$$\begin{array}{lcl}
\phi \times \psi & = & (\phi^e \times \psi^e, \phi^p \times \psi^p) \\[2ex]
\phi + \psi & = & (\phi^e + \psi^e, \phi^p + \psi^p) \\[2ex]
(\phi \rightarrow \psi)^e(f) & = & \psi^e \circ f \circ \phi^p \\
(\phi \rightarrow \psi)^p(g) & = & \psi^p \circ g \circ \phi^e \\[2ex]
id_{\mathbf{Dom}^{ep}}(\phi) & = & \phi \\[2ex]
K_A(\phi) & = & (id_A, id_A).
\end{array}$$

Given functors $F : \mathbf{C} \rightarrow \mathbf{C}_1$ and $G : \mathbf{C} \rightarrow \mathbf{C}_2$, their pairing $\langle F, G\rangle : \mathbf{C} \rightarrow \mathbf{C}_1 \times \mathbf{C}_2$ is defined in the obvious way. Each parameterised type $\sigma^\alpha \in \mathcal{T}^\alpha$ is interpreted as a functor $F_{\sigma^\alpha} : \mathbf{Dom}^{ep} \rightarrow \mathbf{Dom}^{ep}$ as follows:

$$\begin{array}{lcl}
F_\sigma & = & K_{D^{\mathbf{S}}_\sigma} \\[2ex]
F_{\sigma_1^\alpha \times \sigma_2^\alpha} & = & \times \circ \left\langle F_{\sigma_1^\alpha}, F_{\sigma_2^\alpha} \right\rangle \\[2ex]
F_{\sigma_1^\alpha \rightarrow \sigma_2^\alpha} & = & \rightarrow \circ \left\langle F_{\sigma_1^\alpha}, F_{\sigma_2^\alpha} \right\rangle \\[2ex]
F_{\sigma_1^\alpha + \sigma_2^\alpha} & = & + \circ \left\langle F_{\sigma_1^\alpha}, F_{\sigma_2^\alpha} \right\rangle \\[2ex]
F_\alpha & = & id_{\mathbf{Dom}^{ep}}.
\end{array}$$

6.4 A Category of Complete Pers

One likely choice of category in which to interpret recursive descriptions of pers would be the one defined by Abadi and Plotkin in [AP90], since that work shows how recursive types can be interpreted as pers. However, viewed as objects in their category the pers *Bot* and *All* are *isomorphic* which does not seem to be appropriate in the current setting. The basic category we use is an adaptation of [SP82]'s category of *ω-complete relations*. Nielson also uses a similar category (the category **SIM** of [Nie89]), the essential difference between Nielson's use and ours being that we do not attempt to induce a *logical relation* for recursive types, but settle less ambitiously for a way of understanding recursively described pers over such types.

Definition 6.4.1 *The category* **CPD** *has as objects pairs* (D, P)*, where* D *is a domain and* $P \in \mathrm{CPER}(D)$*. By an abuse of notation we will often refer to such a pair by* P *alone, writing* $D(P)$ *for the domain component when necessary. A* **CPD** *morphism* $f : P \to Q$ *is a strict continuous map* $f : D(P) \to D(Q)$ *such that* $f : P \Rightarrow Q$*. Identities and compositions are inherited from* $\mathbf{Dom}_\perp$*.*

Verifying that **CPD** is well defined (in particular that if $f : P \Rightarrow Q$ and $g : Q \Rightarrow R$ then $g \circ f : P \Rightarrow R$) and that ordering the hom-sets of **CPD** by the usual pointwise ordering on functions makes **CPD** an **O**-category, is straightforward. We can thus form the category $\mathbf{CPD}^{ep}$ in which morphisms $\phi : P \to Q$ are embedding-projection pairs from $D(P)$ to $D(Q)$ in $\mathbf{Dom}^{ep}$ such that $\phi^e : P \Rightarrow Q$ and $\phi^p : Q \Rightarrow P$.

Since **CPD** is derived from $\mathbf{Dom}_\perp$ by adding extra structure to the objects and requiring morphisms to preserve the additional structure, it is natural to define the forgetful functor $U : \mathbf{CPD} \to \mathbf{Dom}_\perp$ such that $U(P) = D(P)$ and $U(f) = f$. We will also denote the forgetful functor $\mathbf{CPD}^{ep} \to \mathbf{Dom}^{ep}$ by U, relying on context to determine which is intended.

6.4.1 Functors on CPD and CPD^{ep}

The functors $\times$, $+$ and $\to$ over $\mathbf{Dom}_\perp$ can be extended to functors over **CPD**. Their operation on morphisms is as before. On objects they are defined as follows:

$$\begin{aligned} (A, P) \times (B, Q) &= (A \times B, P \times Q) \\ (A, P) + (B, Q) &= (A + B, P + Q) \\ (A, P) \to (B, Q) &= ([A \to B], P \Rightarrow Q). \end{aligned}$$

The following result verifies that these functors are well defined.

Lemma 6.4.2 *Let* $f : P \to Q$ *and* $g : R \to S$ *be* **CPD** *morphisms. Then the following hold:*

1. $(f \times g) : (P \times R) \Rightarrow (Q \times S)$;
2. $(f + g) : (P + R) \Rightarrow (Q + S)$;
3. $(f \to g) : (Q \Rightarrow R) \Rightarrow (P \Rightarrow S)$.

Proof

1. If $(a, c)\ (P \times R)\ (a', c')$ then $a\ P\ a'$ and $c\ R\ c'$. By assumption, $f(a)\ Q\ f(a')$ and $g(c)\ S\ g(c')$, hence $(f \times g)(a, c)\ (P \times R)\ (f \times g)(a', c')$.

2. Suppose $x\ (P + R)\ x'$. By the definition of $+$ on pers, there are three cases to consider. The first case, $x = x' = \perp$, is clear since $(f + g)$ is strict and so are the pers. The second case is that $x = in_1(a)$ and $x' = in_1(a')$ with $a\ P\ a'$. Then $(f + g)(x) = in_1(f(a))$ and $(f + g)(x') = in_1(f(a'))$. By assumption $f(a)\ Q\ f(a')$, hence $(f + g)(x)\ (Q + S)\ (f + g)(x')$. The third case, $x = in_2(c)$ and $x' = in_2(c')$, is similar.

3. Suppose that $h\ (Q \Rightarrow R)\ h'$. We must show that $g \circ h \circ f\ (P \Rightarrow S)\ g \circ h' \circ f$. So assume that $a\ P\ a'$. Then $f(a)\ Q\ f(a')$, hence $h(f(a))\ R\ h'(f(a'))$, hence $g(h(f(a)))\ S\ g(h'(f(a')))$.

$\square$

Again, we also have the identity functor and the constant functors on **CPD**. Using [SP82]'s technique, all these functors can be converted to covariant functors over $\mathbf{CPD}^{ep}$, with the same definition on objects and acting as the corresponding $\mathbf{Dom}^{ep}$ functors on morphisms.

Assuming finite lattices D^J_σ and concretisation maps γ^J_σ satisfying the requirements outlined in Section 6.2, we define for each parameterised type σ^α, a set $\mathcal{P}_{\sigma^\alpha}$ of formal expressions denoting functors on $\mathbf{CPD}^{ep}$:

$$\begin{array}{lcl}
\mathcal{P}_\sigma & = & \left\{\widehat{a} \mid a \in D^J_\sigma\right\} \\
\mathcal{P}_{\sigma_1^\alpha \times \sigma_2^\alpha} & = & \left\{p_1 \times p_2 \mid p_1 \in \mathcal{P}_{\sigma_1^\alpha}, p_2 \in \mathcal{P}_{\sigma_2^\alpha}\right\} \\
\mathcal{P}_{\sigma_1^\alpha \to \sigma_2^\alpha} & = & \left\{p_1 \to p_2 \mid p_1 \in \mathcal{P}_{\sigma_1^\alpha}, p_2 \in \mathcal{P}_{\sigma_2^\alpha}\right\} \\
\mathcal{P}_{\sigma_1^\alpha + \sigma_2^\alpha} & = & \left\{\widehat{\top}\right\} \cup \left\{p_1 + p_2 \mid p_1 \in \mathcal{P}_{\sigma_1^\alpha}, p_2 \in \mathcal{P}_{\sigma_2^\alpha}\right\} \\
\mathcal{P}_\alpha & = & \{\alpha\}\,.
\end{array}$$

Each $p \in \mathcal{P}_{\sigma^\alpha}$ is interpreted as a functor $F_p : \mathbf{CPD}^{ep} \to \mathbf{CPD}^{ep}$ as follows:

$$\begin{array}{lcl}
\mathcal{P}_\sigma & : & F_{\widehat{a}} = K_{(D^{\mathbf{S}}_\sigma, \gamma^J_\sigma\ a)} \\
\mathcal{P}_{\sigma_1^\alpha \times \sigma_2^\alpha} & : & F_{p_1 \times p_2} = \times \circ \langle F_{p_1}, F_{p_2} \rangle \\
\mathcal{P}_{\sigma_1^\alpha \to \sigma_2^\alpha} & : & F_{p_1 \to p_2} = \to \circ \langle F_{p_1}, F_{p_2} \rangle \\
\mathcal{P}_{\sigma_1^\alpha + \sigma_2^\alpha} & : & \left\{ \begin{array}{lcl} F_{\widehat{\top}}(A, P) & = & (F_{\sigma_1^\alpha + \sigma_2^\alpha}(A), All_{F_{\sigma_1^\alpha + \sigma_2^\alpha}(A)}) \\ F_{\widehat{\top}}(\phi) & = & F_{\sigma_1^\alpha + \sigma_2^\alpha}(\phi) \\ F_{p_1 + p_2} & = & + \circ \langle F_{p_1}, F_{p_2} \rangle \end{array} \right. \\
\mathcal{P}_\alpha & : & F_\alpha = id_{\mathbf{CPD}^{ep}}.
\end{array}$$

The definition of $F_{\widehat{\top}}$ for the sum case is valid because $\psi^e : All \Rightarrow All$ and $\psi^p : All \Rightarrow All$ for any ψ.

The following result shows that for $p \in \mathcal{P}_{\sigma^\alpha}$, the functor F_p is essentially F_{σ^α} combined with a map on pers.

Lemma 6.4.3 *Let $\sigma^\alpha \in \mathcal{T}^\alpha$ and let $p \in \mathcal{P}_{\sigma^\alpha}$. Let A be a domain, let $P \in$ CPER(A) and let ϕ be a morphism in $\mathbf{CPD}^{ep}$. Then:*

1. $U(F_p(A, P)) = F_{\sigma^\alpha}(A)$;
2. $U(F_p(\phi)) = F_{\sigma^\alpha}(\phi)$.

Proof Routine induction on the structure of σ^α. □

6.5 Solving Recursive Equations in $\mathbf{CPD}^{ep}$

To apply the techniques of [SP82] to solving 'equations'

$$F_p(D, P) \cong (D, P)$$

in $\mathbf{CPD}^{ep}$, it is required that:

1. the **CPD** functors $\times$, $+$, $\to$, $K_{(A,P)}$ and $id_{\mathbf{CPD}}$ are *locally continuous*;
2. all ω^{OP}-chains in **CPD** have limits.

Given complete pers P, Q and an ω-chain $\{f_n\}$ of continuous maps such that $f_n : P \Rightarrow Q$ for all n, it is easy to see that $\bigsqcup\{f_n\} : P \Rightarrow Q$. It follows that the functors are locally continuous just as their $\mathbf{Dom}_\perp$ counterparts are. The next result shows that the second requirement is also satisfied.

Lemma 6.5.1 *Let Δ be an ω^{op}-chain in* **CPD***:*

$$(D_0, P_0) \xleftarrow{f_0} (D_1, P_1) \xleftarrow{f_1} (D_2, P_2) \xleftarrow{f_2} \cdots$$

By applying the forgetful functor we obtain an ω^{op}-chain $U(\Delta)$ in $\mathbf{Dom}_\perp$:

$$D_0 \xleftarrow{f_0} D_1 \xleftarrow{f_1} D_2 \xleftarrow{f_2} \cdots$$

It is known that $\mathbf{Dom}_\perp$ has limits of all ω^{op}-chains, so let $\nu : A \to U(\Delta)$ be a limiting cone in $\mathbf{Dom}_\perp$ and define the relation $P(\Delta, \nu) : A \leftrightarrow A$ by

$$y \; P(\Delta, \nu) \; y' \iff \forall n.\, \nu_n(y) \; P_n \; \nu_n(y').$$

Then the cone $\nu : (A, P(\Delta, \nu)) \to \Delta$ is well defined and limiting in **CPD***.*

Proof We must show:

1. $P(\Delta, \nu)$ is a complete per on A;
2. $\forall n.\, \nu_n : P(\Delta, \nu) \Rightarrow P_n$;
3. if $\nu' : (A', P) \to \Delta$ is any other cone, then the unique mediating morphism $\alpha : A' \to A$ from $A' \xrightarrow{U(\nu')} U(\Delta)$ to $A \xrightarrow{\nu} U(\Delta)$ satisfies $\alpha : P \Rightarrow P(\Delta, \nu)$.

For 1, since each ν_n and each P_n is strict we have $\perp_A \; P(\Delta, \nu) \; \perp_A$. Now suppose that $\{y_m\}$ and $\{y'_m\}$ are ω-chains in A such that $\forall m.\, y_m \; P(\Delta, \nu) \; y'_m$, i.e., $\forall m. \forall n.\, \nu_n(y_m) \; P_n \; \nu_n(y'_m)$. For any n, since ν_n is continuous we have $\nu_n(\bigsqcup\{y_m\}) = \bigsqcup\{\nu_n(y_m)\}$, and similarly for $\{y'_m\}$. Then inductiveness of the P_n implies that $\forall n.\, \bigsqcup\{y_m\} \; P_n \bigsqcup\{y'_m\}$, hence $\bigsqcup\{y_m\} \; P(\Delta, \nu) \bigsqcup\{y'_m\}$.

2 is immediate by the construction of $P(\Delta, \nu)$.

For 3, firstly it is clear that $U(\nu') : A' \to U(\Delta)$ actually *is* a cone. Now suppose $z \; P \; z'$. Then $\nu'_n(z) \; P_n \; \nu'_n(z')$ for any n, since $\nu'_n : P \Rightarrow P_n$. But $\nu'_n = \nu_n \circ \alpha$, hence $\forall n.\, \nu_n(\alpha(z)) \; P_n \; \nu_n(\alpha(z'))$, hence $\alpha(z) \; P(\Delta, \nu) \; \alpha(z')$. □

The initial object of $\mathbf{Dom}^{ep}$ is the one-point domain $\mathbf{1}$ (note that this is indeed *initial* in $\mathbf{Dom}^{ep}$, while being *terminal* in **Dom**). For any domain A the unique morphism from $\mathbf{1}$ to A is denoted 0_A and is just $\cdot \mapsto \perp_A$. The initial object of $\mathbf{CPD}^{ep}$ is just $(\mathbf{1}, Bot_{\mathbf{1}})$ and for any (A, P) the unique morphism from $(\mathbf{1}, Bot_{\mathbf{1}})$ to (A, P) is again 0_A.

For a functor $F : \mathbf{Dom}^{ep} \to \mathbf{Dom}^{ep}$, define the ω-chain Δ_F to be:

$$\mathbf{1} \xrightarrow{0_A} F(\mathbf{1}) \xrightarrow{F(0_A)} F^2(\mathbf{1}) \xrightarrow{F^2(0_A)} \cdots$$

Similarly, for $F : \mathbf{CPD}^{ep} \to \mathbf{CPD}^{ep}$, define Δ_F to be:

$$(\mathbf{1}, Bot_{\mathbf{1}}) \xrightarrow{0_A} F(\mathbf{1}, Bot_{\mathbf{1}}) \xrightarrow{F(0_A)} F^2(\mathbf{1}, Bot_{\mathbf{1}}) \xrightarrow{F^2(0_A)} \cdots$$

For $\sigma^\alpha \in \mathcal{T}^\mu$ and $p \in \mathcal{P}_{\sigma^\alpha}$, we let Δ_{σ^α} abbreviate $\Delta_{F_{\sigma^\alpha}}$ and Δ_p abbreviate Δ_{F_p}. The Basic Lemma of [SP82] shows that an interpretation for a type $\mu\alpha\,.\,\sigma^\alpha$, i.e., an initial solution to

$$F_{\sigma^\alpha}(D) \cong D,$$

is given by a colimiting cone $D \xleftarrow{\nu} \Delta_{\sigma^\alpha}$ in $\mathbf{Dom}^{ep}$. Similarly, for each $p \in \mathcal{P}_{\sigma^\alpha}$, we can obtain an initial solution to

$$F_p(D, P) \cong (D, P)$$

as a colimit $(D, P) \xleftarrow{\nu} \Delta_p$ in $\mathbf{CPD}^{ep}$. By using the following lemma, we are able to construct the per solutions given the domain solution:

Lemma 6.5.2 *Let $\sigma^\alpha \in \mathcal{T}^\alpha$ and let $p \in \mathcal{P}_{\sigma^\alpha}$. Then $U(\Delta_p) = \Delta_{\sigma^\alpha}$.*

Proof A simple corollary of Lemma 6.4.3. □

Armed with this lemma we obtain the following result via [SP82]'s Theorem 2:

Proposition 6.5.3 *Let $\sigma^\alpha \in \mathcal{T}^\alpha$ and assume $D \xleftarrow{\nu} \Delta_{\sigma^\alpha}$ to be colimiting in $\mathbf{Dom}^{ep}$ with unique mediating isomorphism $\phi : F_{\sigma^\alpha}(D) \to D$. Then for each $p \in \mathcal{P}_{\sigma^\alpha}$, $(D, P(\Delta_p, \nu)) \xleftarrow{\nu} \Delta_p$ is colimiting in $\mathbf{CPD}^{ep}$ and the unique mediating isomorphism from $F_p(D, P(\Delta_p, \nu))$ to $(D, P(\Delta_p, \nu))$ is also ϕ.*

In what follows we assume for each $\sigma^\alpha \in \mathcal{T}^\alpha$ some chosen colimiting cone $D_{\sigma^\alpha} \xleftarrow{\nu_{\sigma^\alpha}} \Delta_{\sigma^\alpha}$. The mediating $\mathbf{Dom}^{ep}$ isomorphism from $F_{\sigma^\alpha}(D_{\sigma^\alpha})$ to D_{σ^α} is denoted ϕ_{σ^α}.

$$
\begin{array}{lcl}
[\![\_]\!]^{\mathrm{s}} : \Lambda_{\mathcal{T}^\mu} \to Env^{\mathrm{s}} \to \bigcup_{\sigma \in \mathcal{T}^\mu} D^{\mathrm{s}}_\sigma & & \\
[\![()]\!]^{\mathrm{s}}\rho & = & \cdot \\
[\![\mathbf{inl}_{\sigma_1,\sigma_2}(e)]\!]^{\mathrm{s}}\rho & = & in_1([\![e]\!]^{\mathrm{s}}\rho) \\
[\![\mathbf{inr}_{\sigma_1,\sigma_2}(e)]\!]^{\mathrm{s}}\rho & = & in_2([\![e]\!]^{\mathrm{s}}\rho) \\
\multicolumn{3}{l}{[\![\mathbf{case}\ e\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_1\ \mathbf{or}\ \mathbf{inr}(y) \Rightarrow e_2]\!]^{\mathrm{s}}\rho} \\
& = & \begin{cases} \perp & \text{if } [\![e]\!]^{\mathrm{s}}\rho = \perp \\ [\![e_1]\!]^{\mathrm{s}}\rho[x \mapsto d_1] & \text{if } [\![e]\!]^{\mathrm{s}}\rho = in_1(d_1) \\ [\![e_2]\!]^{\mathrm{s}}\rho[y \mapsto d_2] & \text{if } [\![e]\!]^{\mathrm{s}}\rho = in_2(d_2) \end{cases} \\
[\![\mathbf{fold}_{\sigma^\alpha}(e)]\!]^{\mathrm{s}}\rho & = & \mathbf{fold}^{\mathrm{s}}_{\sigma^\alpha}([\![e]\!]^{\mathrm{s}}\rho) \\
[\![\mathbf{unfold}_{\sigma^\alpha}(e)]\!]^{\mathrm{s}}\rho & = & \mathbf{unfold}^{\mathrm{s}}_{\sigma^\alpha}([\![e]\!]^{\mathrm{s}}\rho)
\end{array}
$$

Figure 6.3: The Standard Interpretation of Terms in $\Lambda_{\mathcal{T}^\mu}$

6.6 The Standard Interpretation of Terms and Recursive Types

The standard interpretation of a recursive type is taken to be:

$$D^{\mathrm{s}}_{\mu\alpha\,.\,\sigma^\alpha} = D_{\sigma^\alpha}.$$

The standard interpretation of terms involving **fold** and **unfold** is determined by defining $\mathbf{fold}^{\mathrm{s}} : F_{\sigma^\alpha}(D_{\sigma^\alpha}) \to D_{\sigma^\alpha}$ and $\mathbf{unfold}^{\mathrm{s}} : D_{\sigma^\alpha} \to F_{\sigma^\alpha}(D_{\sigma^\alpha})$ to be $\phi^e_{\sigma^\alpha}$ and $\phi^p_{\sigma^\alpha}$ respectively. The standard interpretation of terms in $\Lambda_{\mathcal{T}^\mu}$ is then as for $\Lambda_{\mathcal{T}}$ together with the additional clauses shown in Figure 6.3.

6.7 The Abstract Interpretation of Terms and Recursive Types

In Subsection 6.4.1, assuming a finite lattice interpretation D^J_σ and a concretisation map γ^J_σ for each $\sigma \in \mathcal{T}^\mu$, we defined, for each $\sigma^\alpha \in \mathcal{T}^\alpha$, a finite set of expressions $\mathcal{P}_{\sigma^\alpha}$ with each $p \in \mathcal{P}_{\sigma^\alpha}$ denoting a $\mathbf{CPD}^{ep}$ functor. In Section 6.4.1 we saw how we could go on to interpret each such functor as defining a complete per on $D_{\sigma^\alpha} = D^{\mathrm{s}}_{\mu\alpha\,.\,\sigma^\alpha}$, namely the per component $P(\Delta_p, \nu_{\sigma^\alpha})$ of an initial solution to the equation

$$F_p(D, P) \cong (D, P).$$

It is thus natural to want to take $D^J_{\mu\alpha\,.\,\sigma^\alpha}$ to be $\mathcal{P}_{\sigma^\alpha}$ and to define $\gamma^J_{\mu\alpha\,.\,\sigma^\alpha}$ by

$$\gamma^J_{\mu\alpha\,.\,\sigma^\alpha}\; p = P(\Delta_p, \nu_{\sigma^\alpha}).$$

The obvious definition of a partial order making each $\mathcal{P}_{\sigma^\alpha}$ into a complete lattice is as follows:

$$\begin{array}{lll}
\mathcal{P}_\sigma & : & \hat{a} \leq \hat{a}' \iff a \sqsubseteq a' \\
\mathcal{P}_{\sigma_1^\alpha \times \sigma_2^\alpha} & : & (p_1 \times p_2) \leq (p_1' \times p_2') \iff p_1 \leq p_1' \textit{ and } p_2 \leq p_2' \\
\mathcal{P}_{\sigma_1^\alpha \to \sigma_2^\alpha} & : & (p_1 \to p_2) \leq (p_1' \to p_2') \iff p_1' \leq p_1 \textit{ and } p_2 \leq p_2' \\
\mathcal{P}_{\sigma_1^\alpha + \sigma_2^\alpha} & : & \left\{ \begin{array}{lll} (p_1 + p_2) \leq (p_1' + p_2') & \iff & p_1 \leq p_1' \textit{ and } p_2 \leq p_2' \\ (p_1 + p_2) \leq \widehat{\top} & & \\ \widehat{\top} \leq \widehat{\top} & & \end{array} \right. \\
\mathcal{P}_\alpha & : & \alpha \leq \alpha.
\end{array}$$

With this ordering we will denote the meet and join operations on the $\mathcal{P}_{\sigma^\alpha}$ by $\wedge$ and $\vee$ respectively.

6.7.1 An Unsolved Monotonicity Problem

Not all the $\mathcal{P}_{\sigma^\alpha}$ lattices defined above are acceptable. To see why, consider the following example: let lam^α be the $\mathcal{T}^\alpha$ type $int + (\alpha \to \alpha)$ and let lam be the type $\mu\alpha\,.\,lam^\alpha$. Thus lam is a data type over which we might define an interpreter for an untyped lambda calculus. Suppose J at the base types to be as for the interpretation **C** of the previous chapter. Then $\mathcal{P}_{lam^\alpha}$ is:

$$\begin{tikzcd}
\widehat{\top} \arrow[d, dash] \\
\widehat{\mathrm{D}} + (\alpha \to \alpha) \arrow[d, dash] \\
\widehat{\mathrm{S}} + (\alpha \to \alpha)
\end{tikzcd}$$

With the suggested definition of γ^J_{lam}, it is clear that $\gamma^J_{lam}\ \widehat{\top} = P(\Delta_{\widehat{\top}}, \nu_{lam^\alpha}) = All_{lam}$. The other two points are recursive specifications of pers which we will write informally as

$$\begin{aligned}
\gamma^J_{lam}(\widehat{\mathsf{D}} + (\alpha \to \alpha)) &= Q &= All + (Q \Rightarrow Q) \\
\gamma^J_{lam}(\widehat{\mathsf{S}} + (\alpha \to \alpha)) &= S &= Id + (S \Rightarrow S).
\end{aligned}$$

It is easy to show (by a simple mathematical induction) that S is just Id_{lam}. It is not nearly so clear how to gain an intuitive understanding of the per Q. However, we can understand just enough of it to realise that something is awry with our chosen ordering on $\mathcal{P}_{lam^\alpha}$. Let ν be ν_{lam^α} and let $f : D_{lam^\alpha} \to D_{lam^\alpha}$ be a continuous function such that (suppressing the applications of the mediating isomorphism) $f(in_1(0)) = in_1(0)$ and $f(in_1(101)) = in_2(g)$ for some g. It should be clear that such a function exists. Now from the construction of $P(\Delta, \nu)$ we know that if $in_2(f) \in |Q|$, then

$$\nu^p_1 \circ f \circ \nu^e_1 : Q_1 \Rightarrow Q_1,$$

where $Q_1 = All_{\mathbf{Z}} + (Bot_{\mathbf{1}} \Rightarrow Bot_{\mathbf{1}})$ is the second approximation to Q. But $\nu^e_1(in_1(n)) = in_1(n)$ and $\nu^p_1(in_2(g) = in_2(0_{\mathbf{1}})$ for any $n \in DomZ$ and $g \in D_{lam^\alpha}$. Then since $0\ All\ 101$, $in_2(f) \in |Q|$ implies that $in_1(0)\ (All + (Bot \Rightarrow Bot))\ in_2(0_{\mathbf{1}})$, which is false by the definition of $+$ on pers.

What we have shown is that there are elements of D_{lam^α} which are not in $|Q|$. Thus $Id \leq Q$ does not hold, and so our suggested ordering on $\mathcal{P}_{lam^\alpha}$ results in a non-monotone concretisation map. Worse than that, the ordering on the pers All, Q and Id is:

$$\begin{tikzcd}
 & All \arrow[dl, dash] \arrow[dr, dash] & \\
Q & & Id
\end{tikzcd}$$

Thus there is *no* ordering on $\mathcal{P}_{lam^\alpha}$ making $\mathcal{P}_{lam^\alpha}$ a complete lattice and $\gamma^J_{\mu\alpha\,.\,lam^\alpha}$ monotone. The use of $\to$ in lam^α is the cause of the problem, as we can see if we define the following restricted language of types:

$$\sigma \in \mathcal{T}^{\overline{\mu}} ::= \iota \mid \sigma_1 \times \sigma_2 \mid \sigma_1 \to \sigma_2 \mid 1 \mid \sigma_1 + \sigma_2 \mid \mu\alpha\,.\,\sigma^\alpha$$

$$\sigma^\alpha \in \mathcal{T}^{\overline{\alpha}} ::= \sigma \mid \sigma^\alpha_1 \times \sigma^\alpha_2 \mid \sigma^\alpha_1 + \sigma^\alpha_2 \mid \alpha$$

Thus $\mathcal{T}^{\overline{\mu}}$ is the subset of $\mathcal{T}^\mu$ for which free occurrences of α never occur in the arguments of $\to$, so $\to$ is not 'active' in recursive definitions. For such types the ordering on $\mathcal{P}_{\sigma^\alpha}$ *does* result in monotone concretisation maps, as shown by the corollary to the following lemma.

Lemma 6.7.1 *Let $\sigma^\alpha \in \mathcal{T}^{\overline{\alpha}}$, let $p, p' \in \mathcal{P}_{\sigma^\alpha}$ and let P and P' be objects in* **CPD** *such that $D(P) = D(P')$. Then*

1. $p \leq p' \Rightarrow F_p(P) \leq F_{p'}(P)$;
2. $P \leq P' \Rightarrow F_p(P) \leq F_p(P')$.

Proof The first part is by an easy induction on σ^α (this part also goes through for $\sigma^\alpha \in \mathcal{T}^\alpha$). The second part amounts to the assertion that the constant functors and $+$ and $\times$ are all monotone when viewed as maps over lattices of pers, which is easily verified. The banning of $\rightarrow$ from $\mathcal{T}^{\overline{\alpha}}$ types is clearly essential since $\_ \Rightarrow \_$ is not monotone in its first argument. □

Corollary 6.7.2 *Let $\sigma^\alpha \in \mathcal{T}^{\overline{\alpha}}$ and let $p, p' \in \mathcal{P}_{\sigma^\alpha}$. Then*

$$p \leq p' \Rightarrow P(\Delta_p, \nu_{\sigma^\alpha}) \leq P(\Delta_{p'}, \nu_{\sigma^\alpha}).$$

Proof Follows in the obvious way from the Lemma by the construction of $P(\Delta, \nu)$. □

Remark Both the Lemma and the Corollary could be generalised by weakening the restriction on types in $\mathcal{T}^{\overline{\alpha}}$ so that $\rightarrow$ was allowed but only in such a way that all free occurrences of α were 'positive'. This is a well known syntactic condition which guarantees that the resulting map on pers is monotone, for example see [Ama91]. We do not do this since the resulting class of types does not seem to be significantly more useful than $\mathcal{T}^{\overline{\mu}}$.

The language of terms over $\Lambda_{\mathcal{T}^{\overline{\mu}}}$ is defined the same way as $\Lambda_{\mathcal{T}^\mu}$ but of course the well formed (i.e. well typed) terms are assumed only to have types in $\Lambda_{\mathcal{T}^{\overline{\mu}}}$.

So far we know of no other solution to the monotonicity problem than to restrict attention to $\Lambda_{\mathcal{T}^{\overline{\mu}}}$. Clearly, this is something of a disappointment. Since we are forced to restrict things in this way, we *could* have used more straightforward least fixed point semantics to give meaning to the recursively described pers we wish to use as properties over recursive types. On the other hand, if we hadn't developed a way of giving meaning to recursively described pers over the more general class of types, we would not have been able to reveal the monotonicity problem. We have every hope that a solution to the problem will be found, but for now we will have to settle for defining abstract interpretations in the restricted setting.

6.7.2 Abstract Interpretation for The Restricted Case

The type component of an abstract interpretation J for $\Lambda_{\mathcal{T}^{\overline{\mu}}}$ is induced from the following for each $\iota \in \mathcal{T}_0$:

- a finite lattice D^J_ι;
- a monotone concretisation map $\gamma^J_\iota : D^J_\iota \to \text{CPER}(D^{\text{s}}_\iota)$.

These are extended to $\sigma \in \mathcal{T}^{\overline{\mu}}$ as specified in Section 6.2 for product, function, unit and sum types, and by setting

$$\begin{aligned} D^J_{\mu\alpha\,.\,\sigma^\alpha} &= \mathcal{P}_{\sigma^\alpha} \\ \gamma^J_{\mu\alpha\,.\,\sigma^\alpha}\ p &= P(\Delta_p, \nu_{\sigma^\alpha}). \end{aligned}$$

It is straightforward to verify that each γ^J_σ is then monotone and top preserving. The relation R^J is defined by $(d, d')\ R^J\ a \iff d\ (\gamma^J_\sigma\ a)\ d'$. Applying the definition of logical relation for $\mathcal{T}$ without change to $\mathcal{T}^{\overline{\mu}}$, we can see from the proof of the Logical Concretisation Map Theorem that R^J is logical.

The term component of J consists of an interpretation c^J for each constant together with monotone functions

$$\begin{aligned} &\mathbf{unfold}^J_{\sigma^\alpha} : \mathcal{P}_{\sigma^\alpha} \to D^J_{\sigma^\alpha[\alpha \mapsto \mu\alpha\,.\,\sigma^\alpha]} \\ &\mathbf{fold}^J_{\sigma^\alpha} : D^J_{\sigma^\alpha[\alpha \mapsto \mu\alpha\,.\,\sigma^\alpha]} \to \mathcal{P}_{\sigma^\alpha}, \end{aligned}$$

for each $\sigma^\alpha \in \mathcal{T}^{\overline{\alpha}}$. The induced abstract semantic valuation function is as in the general definition of interpretations for $\Lambda_{\mathcal{T}}$, together with the additional clauses shown in Figure 6.4.

For the remainder of this section let J be an abstract interpretation as specified above.

6.7.3 Correctness

Correctness is defined as before for constants. The correctness condition for $\mathbf{unfold}^J_{\sigma^\alpha}$ and $\mathbf{fold}^J_{\sigma^\alpha}$ is just as for the constants, i.e., $\mathbf{unfold}^J_{\sigma^\alpha}$ is *correct* if $\mathbf{unfold}^{\text{s}}_{\sigma^\alpha} : \gamma^J\ \mathbf{unfold}^J_{\sigma^\alpha}$, and similarly for $\mathbf{fold}_{\sigma^\alpha}$.

Proposition 6.7.3 *If $c^{\text{s}} : \gamma^J_\tau\ c^J$ for all constants $c : \tau$, with $\tau \in \mathcal{T}^{\overline{\mu}}$, and if* $\mathbf{unfold}^J_{\sigma^\alpha}$ *and* $\mathbf{fold}^J_{\sigma^\alpha}$ *are correct for all $\sigma^\alpha \in \mathcal{T}^{\overline{\alpha}}$, then J is correct.*

$$
\begin{array}{lcl}
[\![\_]\!]^J : \Lambda_{\mathcal{T}^{\overline{\mu}}} \to Env^J \to \bigcup_{\sigma \in \mathcal{T}^{\overline{\mu}}} D^J_\sigma & & \\
[\![()]\!]^J \delta & = & \cdot \\
[\![\mathbf{inl}_{\sigma_1,\sigma_2}(e)]\!]^J \delta & = & \mathit{colift}([\![e]\!]^J \delta, \bot^J_{\sigma_2}) \\
[\![\mathbf{inr}_{\sigma_1,\sigma_2}(e)]\!]^J \delta & = & \mathit{colift}(\bot^J_{\sigma_1}, [\![e]\!]^J \delta) \\
[\![\mathbf{case}\ e\ \mathbf{of\ inl}(x) \Rightarrow e_1\ \mathbf{or\ inr}(y) \Rightarrow e_2]\!]^J \delta & & \\
 & = & \begin{cases} \top & \text{if } [\![e]\!]^J \delta = \top \\ [\![e_1]\!]^J \delta[x \mapsto a_1] & \\ \qquad \sqcup & \text{if } [\![e]\!]^J \delta = \mathit{colift}(a_1, a_2) \\ [\![e_2]\!]^J \delta[y \mapsto a_2] & \end{cases} \\
[\![\mathbf{fold}_{\sigma^\alpha}(e)]\!]^J \delta & = & \mathbf{fold}^J_{\sigma^\alpha}([\![e]\!]^J \delta) \\
[\![\mathbf{unfold}_{\sigma^\alpha}(e)]\!]^J \delta & = & \mathbf{unfold}^J_{\sigma^\alpha}([\![e]\!]^J \delta)
\end{array}
$$

Figure 6.4: The Abstract Interpretation of Terms in $\Lambda_{\mathcal{T}^{\overline{\alpha}}}$

Proof Under the assumption that the hypotheses of the Proposition hold, we will show for all types $\sigma \in \mathcal{T}^{\overline{\mu}}$ and for all expressions $e : \sigma$ in $\Lambda_{\mathcal{T}^{\overline{\mu}}}$, that

$$(\rho, \rho')\ R^J\ \delta \Rightarrow ([\![e]\!]^{\mathrm{s}} \rho, [\![e]\!]^{\mathrm{s}} \rho')\ R^J_\sigma\ ([\![e]\!]^J \delta).$$

The proof is by induction on the structure of e. The base cases and inductive cases for λ-abstraction, application, pairing and product projections are as in the proof of the Ternary Logical Relations Theorem (3.3.2). The case for $() : 1$ is trivial and the cases for $\mathbf{fold}(e)$ and $\mathbf{unfold}(e)$ are as for the case for application. This leaves $\mathbf{inl}(e)$, $\mathbf{inr}(e)$ and $\mathbf{case}\ e\ \mathbf{of\ inl}(x) \Rightarrow e_1\ \mathbf{or\ inr}(y) \Rightarrow e_2$. The first two follow more or less immediately from the induction hypothesis. For **case** expressions, if $[\![e]\!]^J \delta = \top$ then we are done, since γ^J is top preserving. Otherwise $[\![e]\!]^J \delta = \mathit{colift}(a_1, a_2)$. Now $\gamma^J(\mathit{colift}(a_1, a_2)) = (\gamma^J\ a_1) + (\gamma^J\ a_2)$ and by induction hypothesis $[\![e]\!]^{\mathrm{s}} \rho\ (\gamma^J\ \mathit{colift}(a_1, a_2))\ [\![e]\!]^{\mathrm{s}} \rho'$. There are then three possibilities.

1. $[\![e]\!]^{\mathrm{s}} \rho = [\![e]\!]^{\mathrm{s}} \rho' = \bot$. Then $[\![\mathbf{case}\ e\ \ldots]\!]^{\mathrm{s}} \rho = [\![\mathbf{case}\ e\ \ldots]\!]^{\mathrm{s}} \rho' = \bot$ and $\bot\ (\gamma^J([\![\mathbf{case}\ e\ \ldots]\!]^J \delta))\ \bot$, since complete pers are strict.
2. $[\![e]\!]^{\mathrm{s}} \rho = in_1(d)$ and $[\![e]\!]^{\mathrm{s}} \rho' = in_1(d')$, with $d\ (\gamma^J\ a_1)\ d'$. We must show that $([\![e_1]\!]^{\mathrm{s}} \rho[x \mapsto d])$ and $([\![e_1]\!]^{\mathrm{s}} \rho'[x \mapsto d'])$ are related by $\gamma^J(b_1 \sqcup b_2)$, where $b_1 = [\![e_1]\!]^J \delta[x \mapsto a_1]$ and $b_2 = [\![e_2]\!]^J \delta[y \mapsto a_2]$. But $d\ (\gamma^J\ a_1)\ d'$ implies that $(\rho[x \mapsto d], \rho'[x \mapsto d'])\ R^J\ \delta[x \mapsto a_1]$, so by induction hypothesis $([\![e_1]\!]^{\mathrm{s}} \rho[x \mapsto d])$ and

($[\![e_1]\!]^{\mathrm{s}}\rho'[x \mapsto d']$) are related by $\gamma^J\ b_1$, and monotonicity of γ^J does the rest.

3. $[\![e]\!]^{\mathrm{s}}\rho = in_2(d)$ and $[\![e]\!]^{\mathrm{s}}\rho' = in_2(d')$, with $d\ (\gamma^J\ a_2)\ d'$. Symmetrical with previous case.

□

6.7.4 Defining fold and unfold

It is fairly obvious how to obtain a correct definition for $\mathbf{unfold}^J_{\sigma^\alpha}$. Given $p \in \mathcal{P}_{\sigma^\alpha}$ and $b \in D^J_\tau$, we define $(subst\ p\ b) \in D^J_{\sigma^\alpha[\alpha\mapsto\tau]}$ as follows:

$$\begin{array}{llll}
\sigma & : & subst\ \widehat{a}\ b & = & a \\
\sigma_1^\alpha \times \sigma_2^\alpha & : & subst\ (p_1 \times p_2)\ b & = & (subst\ p_1\ b, subst\ p_2\ b) \\
\sigma_1^\alpha + \sigma_2^\alpha & : & subst\ p\ b & = & \begin{cases} \top & \text{if } p = \widehat{\top} \\ colift(subst\ p_1\ b, subst\ p_2\ b) & \text{if } p = p_1 + p_2 \end{cases} \\
\alpha & : & subst\ \alpha\ b & = & b.
\end{array}$$

As the following lemma shows, substituting a value b for free occurrences of α in p is a syntactic counterpart to applying the functor F_p to the per $\gamma^J\ b$.

Lemma 6.7.4 *Let $\sigma^\alpha \in \mathcal{T}^{\overline{\alpha}}$, let $p \in \mathcal{P}_{\sigma^\alpha}$. Let $\tau \in \mathcal{T}^{\overline{\mu}}$ and let $b \in D^J_{\mathcal{T}^{\overline{\mu}}}$. Then*

$$\gamma^J(subst\ p\ b) = F_p(D^{\mathrm{s}}_\tau, \gamma^J_\tau\ b)$$

(again we identify $F(A,P)$ with its per component).

Proof Routine induction on σ^α. □

Then the following result gives us a correct interpretation for $\mathbf{unfold}^J$.

Proposition 6.7.5 *For each $\sigma^\alpha \in \mathcal{T}^{\overline{\alpha}}$ it is correct to set*

$$\mathbf{unfold}^J_{\sigma^\alpha}\ p = subst\ p\ p.$$

Proof Let $p \in \mathcal{P}_{\sigma^\alpha}$. By the Lemma, $\gamma^J(subst\ p\ p) = F_p(D^{\mathrm{s}}_{\mu\alpha\,.\,\sigma^\alpha}, \gamma^J\ p) = F_p(D_{\sigma^\alpha}, P(\Delta_p, \nu_{\sigma^\alpha}))$. But by Proposition 6.5.3,

$$\mathbf{unfold}^{\mathrm{s}}_{\sigma^\alpha} : P(\Delta_p, \nu_{\sigma^\alpha}) \Rightarrow F_p(P(\Delta_p, \nu_{\sigma^\alpha}))$$

and $P(\Delta_p, \nu_{\sigma^\alpha})$ is just $\gamma^J\ p$. So for all $p \in D^J_{\mu\alpha\,.\,\sigma^\alpha}$

$$\mathbf{unfold}^{\mathrm{s}}_{\sigma^\alpha} : \gamma^J\ p \Rightarrow \gamma^J(subst\ p\ p).$$

Thus $\mathbf{unfold}^{\mathrm{S}}_{\sigma^\alpha} \in \bigcap_{p \in \mathcal{P}_{\sigma^\alpha}} |\gamma^J\ p \Rightarrow \gamma^J(\mathit{subst}\ p\ p)| = |\gamma^J(\lambda\, p \in \mathcal{P}_{\sigma^\alpha}\,.\,\mathit{subst}\ p\ p)|$. $\square$

Given the above definition for $\mathbf{unfold}^J_{\sigma^\alpha}$, the following result gives us a sufficient condition for $\mathbf{fold}^J_{\sigma^\alpha}$ to be correct:

Lemma 6.7.6 *Assume that* $\mathbf{unfold}^J_{\sigma^\alpha}$ *is defined as in the Proposition. Then*

$$\mathbf{unfold}^J_{\sigma^\alpha} \circ \mathbf{fold}^J_{\sigma^\alpha} \sqsupseteq \mathit{id}$$

implies that $\mathbf{fold}^J_{\sigma^\alpha}$ *is correct.*

Proof It is easy to see that because $(\mathbf{fold}^{\mathrm{S}}_{\sigma^\alpha}, \mathbf{unfold}^{\mathrm{S}}_{\sigma^\alpha}) : F_p(D_{\sigma^\alpha}, P(\Delta_p, \nu_{\sigma^\alpha})) \to (D_{\sigma^\alpha}, P(\Delta_p, \nu_{\sigma^\alpha}))$ is an isomorphism, we have the equivalences:

$$d\ \gamma^J(\mathbf{unfold}^J\ p)\ d' \iff d\ F_p(\gamma^J\ p)\ d' \iff (\mathbf{fold}^{\mathrm{S}}\ d)\ (\gamma^J\ p)\ (\mathbf{fold}^{\mathrm{S}}\ d').$$

Now let $a \in D^J_{\sigma^\alpha[\alpha \mapsto \mu\alpha\,.\,\sigma^\alpha]}$ and suppose that $d\ (\gamma^J\ a)\ d'$. Then $d\ \gamma^J(\mathbf{unfold}^J(\mathbf{fold}^J\ a))\ d'$, since $\mathbf{unfold}^J \circ \mathbf{fold}^J \sqsupseteq \mathit{id}$ and γ^J is monotone. But then by the equivalences noted above,

$$(\mathbf{fold}^{\mathrm{S}}\ d)\ \gamma^J(\mathbf{fold}^J\ a)\ (\mathbf{fold}^{\mathrm{S}}\ d'),$$

and we are done. $\square$

A definition of $\mathbf{fold}^J$ satisfying $\mathbf{unfold}^J \circ \mathbf{fold}^J \sqsupseteq \mathit{id}$ can be obtained using the method described in [Lau89]. The basic idea is as follows. Each $a \in D^J_{\sigma^\alpha[\alpha \mapsto \mu\alpha\,.\,\sigma^\alpha]}$ can be seen as some $p \in \mathcal{P}_{\sigma^\alpha}$ having n free occurrences of α, in which for $1 \le i \le n$, the ith occurrence of α has been replaced by some $p_i \in \mathcal{P}_{\sigma^\alpha}$. What is needed is a $q \in \mathcal{P}_{\sigma^\alpha}$ such that $a \sqsubseteq \mathit{subst}\ q\ q$, and this can be obtained by identifying p (this is done by [Lau89]'s `mask` function), identifying the p_i (this is done by [Lau89]'s `extract` function), and taking q to be $\bigvee\{p, p_1, \ldots, p_n\}$. As an example, consider the type of binary trees described in Section 6.2 and suppose that J is the extension to $\mathcal{T}^{\overline{\mu}}$ of the constancy interpretation $\mathbf{C}$ of the previous chapter. Then $\mathcal{P}_{\mathit{bintree}^\alpha}$ is:

$$\begin{array}{c} \widehat{\top} \\ | \\ \widehat{\mathrm{D}} + (\alpha \times \alpha) \\ | \\ \widehat{\mathrm{S}} + (\alpha \times \alpha) \end{array}$$

Take $a \in (D^J_{\mathit{int}} \times (D^J_{\mathit{bintree}} \times D^J_{\mathit{bintree}}))^\top$ to be $\mathit{colift}(\mathrm{S}, (\widehat{\mathrm{S}} + (\alpha \times \alpha), \widehat{\mathrm{D}} + (\alpha \times \alpha)))$. Then $p = \widehat{\mathrm{S}} + (\alpha \times \alpha)$, $p_1 = \widehat{\mathrm{S}} + (\alpha \times \alpha)$ and $p_2 = \widehat{\mathrm{D}} + (\alpha \times \alpha)$. Thus we would take

$\mathbf{fold}^J\ a = p \vee p_1 \vee p_2 = \widehat{\mathrm{D}} + (\alpha \times \alpha)$.

The details of a general definition for $\mathbf{fold}^J$ along these lines are somewhat tedious and we trust the idea is sufficiently clear without them.

6.7.5 Meet Preservation Is Not Inherited

It is interesting to ask whether the above described definitions for $\mathbf{unfold}^J$ and $\mathbf{fold}^J$ are best. The answer is that in general they cannot be best, since in general there *are* no best definitions for $\mathbf{unfold}^J$ and $\mathbf{fold}^J$. This is shown by demonstrating that there are $\sigma^\alpha \in \mathcal{T}^{\overline{\alpha}}$ for which $\gamma^J_{\mu\alpha\,.\,\sigma^\alpha}$ does not preserve meets.

Let t^α be the $\mathcal{T}^{\overline{\alpha}}$ type $int \times (\alpha + \alpha)$ and let t be the type $\mu\alpha\,.\,t^\alpha$. Then $\mathcal{P}_{t^\alpha}$ is:

$$\begin{tikzcd}[column sep=small] & a_4 \arrow[dl, dash] \arrow[dr, dash] & \\ a_2 \arrow[dr, dash] & & a_3 \arrow[dl, dash] \\ & a_1 & \end{tikzcd}$$

where $a_1 = \widehat{\mathrm{S}} \times (\alpha + \alpha)$, $a_2 = \widehat{\mathrm{S}} \times \widehat{\mathrm{T}}$, $a_3 = \widehat{\mathrm{D}} \times (\alpha + \alpha)$ and $a_4 = \widehat{\mathrm{D}} \times \widehat{\mathrm{T}}$. Applying γ^J_t to each of these points we obtain the pers

$$\begin{tikzcd}[column sep=small] & All \arrow[dl, dash] \arrow[dr, dash] & \\ P \arrow[dr, dash] & & Q \arrow[dl, dash] \\ & Id & \end{tikzcd}$$

where P and Q are informally described by:

$$\begin{aligned} P &= Id_{\mathbf{Z}} \times All_{t+t} \\ Q &= All_{\mathbf{Z}} \times (Q + Q). \end{aligned}$$

Now we can see that $P \wedge Q$ is the per informally described by $Id_{\mathbf{Z}} \times (Q + Q)$, which is *not* equal to Id. Thus γ^J_t does not preserve meets. If we take $a \in D^J_{t^\alpha[\alpha \mapsto t]}$ to be $(\mathrm{S}, \mathit{colift}(p, p))$ with $p = \widehat{\mathrm{D}} \times (\alpha + \alpha)$, then $\gamma^J\ a = P \wedge Q$. Thus a_2 and a_3 are both minimal values which would be correct for $\mathbf{fold}^J\ a$. It is interesting to note that the actual value given by the definition of $\mathbf{fold}^J$ sketched above, would be $a_1 \vee a_3 \vee a_3 = a_3$, which is optimal. We do not know whether this holds in general.

Since this example does not involve function types, it applies equally well in the first-order setting of [Lau89].

6.7.6 An Example

Assume $\mathbf{C}$ to be extended to $\mathcal{T}^{\overline{\mu}}$. Let zlist^{α} be the $\mathcal{T}^{\overline{\alpha}}$ type $1 + (\mathit{int} \times \alpha)$ and let zlist be the type $\mu\alpha\,.\,\mathit{zlist}^{\alpha}$. Thus $D^{\mathrm{S}}_{\mathit{zlist}}$ is just $\mathit{list}(\mathbf{Z})$ (up to isomorphism). The abstract domain $D^{\mathrm{C}}_{\mathit{zlist}}$ (i.e., $\mathcal{P}_{\mathit{zlist}^{\alpha}}$) is:

$$\begin{array}{c} \widehat{\top} \\ | \\ \widehat{\cdot} + (\widehat{\mathrm{D}} \times \alpha) \\ | \\ \widehat{\cdot} + (\widehat{\mathrm{S}} \times \alpha) \end{array}$$

In what follows we will write $\textsc{spine}(a)$ to mean $\widehat{\cdot} + (\widehat{a} \times \alpha)$. The point $\widehat{\top}$ describes 'dynamic' lists, $\textsc{spine}(\textsc{d})$ describes lists with 'static structure' but dynamic elements, and $\textsc{spine}(\textsc{s})$ describes completely static lists.

The 'unfolded' lattice $D^{\mathrm{C}}_{\mathit{zlist}^{\alpha}[\alpha \mapsto \mathit{zlist}]}$ is $(\mathbf{1} \times (\mathbf{2} \times \mathcal{P}_{\mathit{zlist}^{\alpha}}))^{\top}$ and for any $a \in \mathbf{2}$ and $b \in \mathcal{P}_{\mathit{zlist}^{\alpha}}$ we have:

$$\mathbf{fold}^{\mathrm{C}}(\mathit{colift}(\cdot, (a, b))) = \textsc{spine}(a) \sqcup b.$$

Let the functions $\mathit{map}^{\mathrm{S}} \in [[\mathbf{Z} \to \mathbf{Z}] \to [\mathit{list}(\mathbf{Z}) \to \mathit{list}(\mathbf{Z})]]$ and $\mathit{map}^{\mathrm{C}} \in [[\mathbf{2} \to \mathbf{2}] \to [\mathcal{P}_{\mathit{zlist}^{\alpha}} \to \mathcal{P}_{\mathit{zlist}^{\alpha}}]]$ be the standard and $\mathbf{C}$ interpretations, respectively, of the following term (of type $(\mathit{int} \to \mathit{int}) \to \mathit{zlist} \to \mathit{zlist}$):

$$\begin{array}{l} \mathbf{Y}\ \lambda\,\mathtt{map}\,.\,\lambda\,\mathtt{f}\,.\,\lambda\,\mathtt{l}\,.\,\mathbf{case}\ \mathbf{unfold}(\mathtt{l})\ \mathbf{of} \\ \qquad\qquad\qquad \mathbf{inl}(\mathtt{x}) \Rightarrow \mathbf{fold}(\mathbf{inl}(()))\ \mathbf{or} \\ \qquad\qquad\qquad \mathbf{inr}(\mathtt{y}) \Rightarrow \mathbf{fold}(\mathbf{inr}(\mathtt{f}\ \mathbf{fst}(\mathtt{y}), \mathtt{map}\ \mathtt{f}\ \mathbf{snd}(\mathtt{y}))) \end{array}$$

We can make this a bit more readable by writing $\mathtt{nil}$ for $\mathbf{fold}(\mathbf{inl}(()))$ and $\mathtt{cons}(e_1, e_2)$ for $\mathbf{fold}(\mathbf{inr}(e_1, e_2))$:

$$\begin{array}{l} \mathbf{Y}\ \lambda\,\mathtt{map}\,.\,\lambda\,\mathtt{f}\,.\,\lambda\,\mathtt{l}\,.\,\mathbf{case}\ \mathbf{unfold}(\mathtt{l})\ \mathbf{of} \\ \qquad\qquad\qquad \mathbf{inl}(\mathtt{x}) \Rightarrow \mathtt{nil}\ \mathbf{or} \\ \qquad\qquad\qquad \mathbf{inr}(\mathtt{y}) \Rightarrow \mathtt{cons}(\mathtt{f}\ \mathbf{fst}(\mathtt{y}), \mathtt{map}\ \mathtt{f}\ \mathbf{snd}(\mathtt{y})) \end{array}$$

The standard interpretation map^{S} is the function which maps a function over a list of integers. The abstract interpretation map^{C} is given as the limit of the sequence $\{map_0^{\mathrm{C}}, map_1^{\mathrm{C}}, \ldots\}$ generated by:

$$
\begin{array}{lcl}
map_0^{\mathrm{C}} \;\; h\ b & = & \textsc{spine}(\textsc{s}) \\
map_{n+1}^{\mathrm{C}}\ h\ b & = & \left\{ \begin{array}{ll} \hat{\top} & \text{if } \mathbf{unfold}^{\mathrm{C}}\ b = \top \\ \begin{array}{c} \mathbf{fold}^{\mathrm{C}}(\mathit{colift}(\cdot, (\textsc{s}, \textsc{spine}(\textsc{s})))) \\ \sqcup \\ \mathbf{fold}^{\mathrm{C}}(\mathit{colift}(\cdot, (h\ a, map_n^{\mathrm{C}}\ h\ b'))) \end{array} & \text{if } \mathbf{unfold}^{\mathrm{C}}\ b = \mathit{colift}(\cdot, (a, b')) \end{array} \right.
\end{array}
$$

Consider the cases for $\mathbf{unfold}^{\mathrm{C}}(b)$ with $b \in \mathcal{P}_{zlist^{\alpha}}$: either $b = \hat{\top}$, in which case $\mathbf{unfold}^{\mathrm{C}}(b) = \top$, or $b = \textsc{spine}(a)$, in which case $\mathbf{unfold}^{\mathrm{C}}(b) = \mathit{colift}(\cdot, (a, \textsc{spine}(a)))$. Thus the second iterate in the chain of approximations for map^{C} is

$$
map_1^{\mathrm{C}}\ h\ b \;=\; \left\{ \begin{array}{ll} \hat{\top} & \text{if } b = \hat{\top} \\ \begin{array}{c} \textsc{spine}(\textsc{s}) \sqcup \textsc{spine}(\textsc{s}) \\ \sqcup \\ \textsc{spine}(h\ a) \sqcup (map_0^{\mathrm{C}}\ h\ \textsc{spine}(a)) \end{array} & \text{if } b = \textsc{spine}(a), \end{array} \right.
$$

which simplifies to:

$$
map_1^{\mathrm{C}}\ h\ b \;=\; \left\{ \begin{array}{ll} \hat{\top} & \text{if } b = \hat{\top} \\ \textsc{spine}(h\ a) & \text{if } b = \textsc{spine}(a). \end{array} \right.
$$

The third iterate is

$$
map_2^{\mathrm{C}}\ h\ b \;=\; \left\{ \begin{array}{ll} \hat{\top} & \text{if } b = \hat{\top} \\ \begin{array}{c} \textsc{spine}(\textsc{s}) \sqcup \textsc{spine}(\textsc{s}) \\ \sqcup \\ \textsc{spine}(h\ a) \sqcup (map_1^{\mathrm{C}}\ h\ \textsc{spine}(a)) \end{array} & \text{if } b = \textsc{spine}(a), \end{array} \right.
$$

which simplifies to:

$$
map_2^{\mathrm{C}}\ h\ b \;=\; \left\{ \begin{array}{ll} \hat{\top} & \text{if } b = \hat{\top} \\ \textsc{spine}(h\ a) & \text{if } b = \textsc{spine}(a). \end{array} \right.
$$

So the sequence converges after the second iterate to $map^{\mathrm{C}} = map_1^{\mathrm{C}}$.

Thus for h such that $h\ \textsc{d} = \textsc{s}$ we have $map^{\mathrm{C}}\ h\ \textsc{spine}(\textsc{d}) = \textsc{spine}(h\ \textsc{d}) = \textsc{spine}(\textsc{s})$, so as we might hope, the constancy interpretation tells us that mapping a constant function over a list with static structure but dynamic elements, yields a completely static list.

6.8 Strictness Analysis

In Chapter 5 we described an analysis for detecting head-strictness, involving an ad hoc extension to $\mathcal{T}$ to include lists. The notion of head-strictness was originally defined in [WH87] using projections. Unlike [WH87] where all the domains were lifted and the projection Str was introduced in order to capture ordinary strictness. we used the per *Bot* in a rather simple way. (Note that our analysis would not be able to detect head-strictness if it were not also able to detect ordinary strictness: consider the third clause in the definition of $\mathbf{case^H}$.) However, there is a very important advantage to using the lifting method. In [WH87] and [HL91] it is shown how various interesting strictness properties over recursive types can be described in a rather natural way using the per Str. First the types are interpreted using coalesced sum and smash product, rather than separated sum and cartesian product (this can be seen as a consequence of lifting the domains): thus the domain of lists of integers is a solution to the equation

$$\mathit{list}(\mathbf{Z}) = \mathbf{1}_\perp \oplus (\mathbf{Z}_\perp \otimes \mathit{list}(\mathbf{Z})_\perp).$$

Then the projection H can be recursively described by:

$$\mathrm{H} = \mathit{Id}_{\mathbf{1}_\perp} \oplus (\mathrm{Str}_{\mathbf{Z}} \otimes \mathrm{H}_\perp),$$

where $\oplus$ and $\otimes$ are defined on projections in the obvious way. Another projection with a simple recursive definition is:

$$\mathrm{T} = \mathit{Id}_{\mathbf{1}_\perp} \oplus (\mathit{Id}_{\mathbf{Z}_\perp} \otimes \mathrm{Str} \circ \mathrm{T}_\perp).$$

Then the *tail-strict* functions can be characterised as those for which

$$\mathrm{Str} \circ f = \mathrm{Str} \circ f \circ (\mathrm{Str} \circ \mathrm{T}_\perp).$$

In fact T is capturing the same property as Wadler's abstract lattice point ∞, which denotes the Scott-closed set of all lists not ending in [] (the set *Inf* described in Chapter 2, Section 2.3).

Unfortunately these properties are *not* among those induced by applying the ideas of this chapter to an abstract interpretation which uses *Bot* to capture ordinary strictness. Suppose we define an abstract interpretation J for strictness analysis with:

$$D^J_{int} = \begin{array}{c} \mathrm{D} \\ | \\ \mathrm{S} \\ | \\ \mathrm{B} \end{array} \qquad\qquad \gamma^J_{int}\ a = \begin{cases} All & \text{if } a = \mathrm{D} \\ Id & \text{if } a = \mathrm{S} \\ Bot & \text{if } a = \mathrm{B}. \end{cases}$$

For the type *zlist* defined in Subsection 6.7.6, the lattice $\mathcal{P}_{zlist^\alpha}$ is:

$$\begin{array}{c} \hat{\top} \\ | \\ \hat{\cdot} + (\hat{\mathrm{D}} \times \alpha) \\ | \\ \hat{\cdot} + (\hat{\mathrm{S}} \times \alpha) \\ | \\ \hat{\cdot} + (\hat{\mathrm{B}} \times \alpha) \end{array}$$

The top point denotes the per *All*, the next one down denotes the per which relates all lists with the same structure and the next denotes *Id*. The bottom point denotes a per informally described by:

$$Q = Bot_{\mathbf{1}} + (Bot_{\mathbf{Z}} \times Q).$$

Just as *Bot* captures the set $\{\perp\}$, the per Q captures the set of all lists *all* of whose elements are $\perp$. While it is conceivable that this property could play a useful role in reasoning about termination properties, it is probably not nearly as useful as the pers ker H and ker T, which are conspicuous by their absence from our lattice of four properties.

Thus our way of capturing ordinary strictness does not seem to combine very well with the [Lau89] scheme for inducing finite lattices of properties over the recursive types. It may be that a better way of capturing strictness would be to extend the lifting technique of [WH87] to the setting of higher-order languages.

Chapter 7

The Frontiers Algorithm

In the preceding chapters we have discussed using non-standard interpretations of terms over finite lattices as a way of performing program analysis. The motivation for using finite lattices is that questions of the form, "$[\![\mathbf{Y}(\lambda\,\mathtt{h}\,.\,e)]\!]^J\ \delta\ a = b$?", with $\mathtt{h}, e : \sigma \to \tau$, are decidable. However, this fact alone does not tell us *how* such a question may be decided. Assuming that J is a least fixed point interpretation, the 'obvious' method is as follows. From the definitions, $[\![\mathbf{Y}(\lambda\,\mathtt{h}\,.\,e)]\!]^J\ \delta$ is the limit f of the ω-chain $\{f_n\}$ generated by:

$$\begin{array}{lcl} f_0 & = & \perp^J_{\sigma\to\tau} \\ f_{n+1} & = & [\![e]\!]^J\delta[\mathtt{f} \mapsto f_n]. \end{array}$$

Now assuming that we are given some way of computing the interpretation of each constant, and given the graph for f_n, it is possible to explicitly construct the graph of f_{n+1}, by computing $[\![e]\!]^J\delta[\mathtt{f} \mapsto f_n]\ a'$ for each $a' \in D^J_\sigma$. Since D^J_σ is finite, each such graph will clearly be finite. We can then compare the graph of f_{n+1} with that of f_n: if they are equal we have reached the least fixed point, if not we iterate this process until we do reach it. That the iteration will eventually terminate is guaranteed, since the sequence $f_0, f_1, f_2, \ldots$ is increasing and the lattice $[D^J_\sigma \to D^J_\tau]$ is finite, so in particular it is of finite *height*. Once we reach the fixed point we can simply 'look-up' the value of $f(a)$ in the graph.

Now this method may seem rather inefficient, since it involves calculating the whole graph of f, even though we only wanted the value of $f(a)$. Unfortunately, in general we know of no other method. In an implementation of the standard interpretation of course, this is not the way recursion is dealt with. But in the implementation of the standard interpretation, if the value of $f(d)$ is $\perp$, we don't expect an attempt to evaluate $f(d)$ to terminate. By contrast, we insist that the evaluation of $f(a)$ always terminates, even if $f(a) = \perp$. Indeed "$f(a) = \perp$?" is very often precisely the question we ask.

The search for ways to avoid calculating the whole graph of recursively defined abstract functions is still going on. In the first-order case, techniques such as *pending analysis* ([YH86]) (related to the idea of *minimal function graphs* ([JM86])), have been successfully employed in implementations of program analyses (such as that described in [Lau89]). This is in spite of the fact that even for first-order languages, the problem of deciding questions such as the one described above is known to be of exponential complexity ([Mey85, HY86]). However, it has proved impossible (so far) to adapt these methods to the higher-order case. A detailed examination of such techniques and a description of the problems posed by higher-order functions may be found in [Mar89]. For the time being at least, we have to accept that in general it *is* necessary to construct the whole graph of abstract functions with recursive definitions. The *frontiers algorithm* was developed by Clack and Peyton Jones ([CJ85, JC87]) as a way of exploiting the monotonicity of the abstract functions to make the construction of their graphs more efficient. In this chapter and the next we describe various extensions of the frontiers method.

7.1 Exploiting Monotonicity

The original work on frontiers was concerned with representations for monotone functions $f : \mathbf{2}^n \to \mathbf{2}$. Although the algorithm developed by Clack and Peyton Jones depended on the function space having this precise form, the principle behind the frontier representation itself applies generally to monotone functions $f : P \to \mathbf{2}$ for any finite poset P. Let f be such a function. The graph of f is the set:

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

It is obvious that this is far from being an optimal representation for f. For example, f could be represented just as well by the set

$$\{x \in P \mid f(x) = 1\}\,,$$

since $f(x) \neq 1 \iff f(x) = 0$. But this second representation is still open to improvement. Suppose that there are two elements $x, x' \in P$ such that $x' \sqsubseteq x$ and $f(x) = f(x') = 1$. Now since f is monotone, it would actually be sufficient to record explicitly only the fact that $f(x') = 1$, since monotonicity of f means that

$x' \sqsubseteq x \Rightarrow f(x') \sqsubseteq f(x)$. Taking this idea to its logical conclusion we arrive at the following candidate representation for f:

$$\{x \in P \mid f(x) = 1, (\forall x'. f(x') = 1 \textit{ and } x' \sqsubseteq x \Rightarrow x = x')\}.$$

In words, this is the set of minimal elements of the subset of P which f maps to 1. This is the set which Clack and Peyton Jones call the *minimum-1-frontier* for f. The *maximum-0-frontier* for f is defined dually as the maximal elements of the subset of P which f maps to 0. In principle, either representation would suffice but in practice, as we shall see in the next chapter (Subsection 8.4.2), it is natural to construct both representations at the same time and in the higher-order case it is also necessary to *keep* both representations.

As was mentioned above, the algorithm developed by Clack and Peyton Jones for establishing the frontier representations of a function depended on the function space having the form $[\mathbf{2}^n \rightarrow_m \mathbf{2}]$. The work of [MH87, Mar89] extended Clack and Peyton Jones's algorithm by removing that restriction, allowing the frontiers method to be applied to the representation of functions in $[L \rightarrow_m \mathbf{2}]$, and indirectly to functions in $[L \rightarrow_m L']$ for any finite lattices L, L'. The description of both the original frontiers algorithm and its subsequent extensions are rather complex. In [Hun89] and [HH91] a greatly simplified description of a frontiers algorithm was arrived at by making explicit what had been implicit in the previous work: the minimum-1-frontier for f represents an *upper set* – the set $f^{-1}\{1\}$ – and the maximum-0-frontier represents a *lower set* – the set $f^{-1}\{0\}$.

To explain more clearly the basis of our frontiers algorithm, we begin with the case of monotone functions $f : P \rightarrow \mathbf{2}$, for finite poset P. In the next chapter the algorithm is generalised to the case of $f \in L$, for finite distributive lattice L.

7.2 A Basic Frontiers Representation

Definition 7.2.1 *Let P be a poset and let $X \subseteq P$. Then $Max_P(X)$ and $Min_P(X)$ are defined to be, respectively, the maximal and minimal elements of X, i.e.:*

1. $Max_P(X) = \{x \in X \mid \forall x' \in X. x \sqsubseteq x' \Rightarrow x = x'\}$;

2. $Min_P(X) = \{x \in X \mid \forall x' \in X. x' \sqsubseteq x \Rightarrow x = x'\}$.

When P is clear from the context we just write $Max(X)$ and $Min(X)$.

Although the following result is fairly obvious, we use it so heavily that it is worth stating explicitly.

Lemma 7.2.2 *Let P be a poset and let $X \subseteq P$ be finite. Then*

1. $\downarrow X = \downarrow Max(X)$;

2. $\uparrow X = \uparrow Min(X)$.

Proof See Appendix. □

Corollary 7.2.3 *Let P be a finite poset. Then*

1. $X \in \mathcal{L}(P) \iff X = \downarrow Max(X)$;

2. $X \in \mathcal{U}(P) \iff X = \uparrow Min(X)$.

The frontiers algorithm(s) depend centrally on the representation of upper sets by their maximal elements and, dually, the representation of lower sets by their minimal elements. The use of sets of maximal and minimal elements as representations is motivated by the fact that they are *irredundant*, i.e., that they contain no comparable elements (which in this context would indeed imply a certain redundancy). For the case of finite posets we have a one-one correspondence between irredundant subsets and lower subsets, and a dual correspondence between irredundant subsets and upper subsets. We can formalise this by defining two orderings on the set of irredundant subsets.

Definition 7.2.4 *Let P be a poset. The lattices $(\mathcal{I}_u(P), \leq_u)$ and $(\mathcal{I}_l(P), \leq_l)$ have as elements the irredundant subsets of P, with order given respectively by:*

1. $X \leq_u X' \iff \forall x' \in X'. \exists x \in X. x \sqsubseteq x'$;

2. $Y \leq_l Y' \iff \forall y \in Y. \exists y' \in Y'. y \sqsubseteq y'$.

Then for finite poset P, using Lemma 7.2.2 and Corollary 7.2.3 it is straightforward to show that Min viewed as a map $\mathcal{U}(P) \to \mathcal{I}_u(P)$ is an isomorphism with inverse $X \mapsto \uparrow X$ and Max viewed as a map $\mathcal{L}(P) \to \mathcal{I}_l(P)$ is an isomorphism with inverse $Y \mapsto \downarrow Y$. Thus $X \leq_u X' \iff \uparrow X \supseteq \uparrow X'$ and $Y \leq_l Y' \iff \downarrow Y \subseteq \downarrow Y'$.

Let P be a finite poset and let $f : P \to \mathbf{2}$ be a monotone function. We will represent f using its maximum-0-frontier, $\mathbf{F\text{-}0}(f)$, and its minimum-1-frontier, $\mathbf{F\text{-}1}(f)$, defined as:

$$\begin{aligned} \mathbf{F\text{-}0}(f) &= Max\,\{x \in P \mid f(x) = 0\} \\ \mathbf{F\text{-}1}(f) &= Min\,\{x \in P \mid f(x) = 1\}. \end{aligned}$$

```
A0 := P
A1 := P
{ Invariant: (I1) and (I2) }
while (A0 ∩ A1) ≠ ∅
      choose x from A0 ∩ A1
      if f(x) = 0
        then A1 := A1 \ ↓x
        else  A0 := A0 \ ↑x
endwhile
```

Figure 7.1: Algorithm A: an algorithm to find $f^{-1}\{0\}$ and $f^{-1}\{1\}$

7.3 The Basic Frontiers Algorithm

As remarked above, the set $\{x \in P \mid f(x) = 0\}$ is lower and $\{x \in P \mid f(x) = 1\}$ is upper. This observation enables a simple derivation of an algorithm to calculate **F-0**(f) and **F-1**(f). We start with an algorithm, Algorithm A shown in Figure 7.1, which determines the whole of the sets $\{x \in P \mid f(x) = 0\}$ (the final value of A_0) and $\{x \in P \mid f(x) = 1\}$ (the final value of A_1). The components of the invariant of Figure 7.1 are:

(I1) $\forall x \in P.\ f(x) = 0 \Rightarrow x \in A_0$ *and* $f(x) = 1 \Rightarrow x \in A_1$;

(I2) A_0 is lower and A_1 is upper.

(Only the first of these is required for the proof of correctness. The second is useful in adapting Algorithm A to compute frontiers.) The semantics of "**choose** x **from** X" is not specified beyond requiring that it assigns some value from the finite set X to the variable x (to simplify what follows we assume that the choice made is a function of X). Although the choice of value does not affect the correctness of the algorithm, it can have a significant effect on the performance of the final version of the algorithm. This point is discussed in detail in [Mar89].

The proof that Algorithm A is correct is straightforward. The initial assignments clearly establish (I1) and (I2). That (I1) is an invariant of the loop follows from monotonicity of f. Recall from Section 1.5 that the complement of a lower set is upper (and vice versa) and that $\mathcal{U}(P)$ and $\mathcal{L}(P)$ are closed under intersection. That (I2) is an invariant then follows from the observation that $A_1 \setminus \downarrow x = A_1 \cap (P \setminus \downarrow x)$ and $A_0 \setminus \uparrow x = A_0 \cap (P \setminus \uparrow x)$. On termination we have $f(x) = 1 \Rightarrow x \in A_1$ by (I1), and $x \in A_1 \Rightarrow x \notin A_0$, from the condition of the loop. Thus we have $f(x) \neq$

B_0 := $Max(P)$
B_1 := $Min(P)$
{ *Invariant: (I1′)* }
while $(\downarrow B_0 \cap \uparrow B_1) \neq \emptyset$
  choose x **from** $\downarrow B_0 \cap \uparrow B_1$
  if $f(x) = 0$
   then B_1 := $Min(\uparrow B_1 \cap P \setminus \downarrow x)$
   else B_0 := $Max(\downarrow B_0 \cap P \setminus \uparrow x)$
endwhile

Figure 7.2: Algorithm B: a naïve algorithm to find $\mathbf{F\text{-}0}(f)$ and $\mathbf{F\text{-}1}(f)$

$0 \Rightarrow f(x) = 1 \Rightarrow x \notin A_0$ and $f(x) = 0 \Rightarrow x \in A_0$, i.e., $x \in A_0 \iff f(x) = 0$. Similarly, $x \in A_1 \iff f(x) = 1$. To show termination we show that $|A_0 \cap A_1|$ strictly decreases each time the body of the loop is executed. Let A_0' and A_1' be the values of A_0 and A_1 before the body of the loop is executed, let A_0'' and A_1'' be the values afterwards and let x be the value chosen from $A_0' \cap A_1'$ when the body is entered. It is clear that $A_0'' \subseteq A_0'$ and $A_1'' \subseteq A_1'$ and furthermore that $x \notin A_0'' \cap A_1''$ (consider the **then** branch of the conditional: $x \in \downarrow x$, hence $x \notin A_1' \setminus \downarrow x \subseteq A_1'$). Thus $|A_0'' \cap A_1''| \leq (|A_0' \cap A_1'| - 1)$.

Figure 7.2 shows a naïve algorithm, derived from Algorithm A, for calculating $\mathbf{F\text{-}0}(f)$ (the final value of B_0) and $\mathbf{F\text{-}1}(f)$ (the final value of B_1). A comparison of the two algorithms will reveal that Algorithm B is actually just mimicking Algorithm A: at each corresponding step in the execution of the algorithms, $B_0 = Max(A_0)$ and $B_1 = Min(A_1)$. (This fact hinges on Corollary 7.2.3 and the observation made above that $A_1 \setminus \downarrow x = A_1 \cap (P \setminus \downarrow x)$ and $A_0 \setminus \uparrow x = A_0 \cap (P \setminus \uparrow x)$.) The modified invariant for Algorithm B is:

(I1′) $\forall x \in P.\ f(x) = 0 \Rightarrow x \in \downarrow B_0$ *and* $f(x) = 1 \Rightarrow x \in \uparrow B_1$.

Correctness of Algorithm B is then immediate from that of Algorithm A. To achieve a realistic algorithm we must eliminate the use of $\downarrow$ and $\uparrow$. This is the crux of the development of a frontiers algorithm.

It is evident from the description of Algorithm B that key operations are calculating the minimal (maximal) elements of the intersection of two upper (lower) sets. Proposition 7.3.2 gives a method for doing this starting from the minimal (maximal) elements of the upper (lower) sets to be intersected. First we need two new binary operations on posets.

Definition 7.3.1 *Let P be a poset and let $x, y \in P$. The* minimal upper bounds *and* maximal lower bounds *of x and y are given respectively by:*

1. $x \mathbin{\tilde{\vee}} y = Min\,\{z \in P \mid x \sqsubseteq z \text{ and } y \sqsubseteq z\}$;

2. $x \mathbin{\tilde{\wedge}} y = Max\,\{z \in P \mid z \sqsubseteq x \text{ and } z \sqsubseteq y\}$.

Note that if P is a lattice then $x \mathbin{\tilde{\vee}} y = \{x \sqcup y\}$ and $x \mathbin{\tilde{\wedge}} y = \{x \sqcap y\}$.

Proposition 7.3.2 *Let P be a finite poset. Let $S_1, S_2 \in \mathcal{U}(P)$ and let $T_1, T_2 \in \mathcal{L}(P)$. Then*

1. $Min(S_1 \cap S_2) = Min\,(\bigcup\{x_1 \mathbin{\tilde{\vee}} x_2 \mid x_1 \in Min(S_1), x_2 \in Min(S_2)\})$;

2. $Max(T_1 \cap T_2) = Max\,(\bigcup\{y_1 \mathbin{\tilde{\wedge}} y_2 \mid y_1 \in Max(T_1), y_2 \in Max(T_2)\})$.

Proof See Appendix. □

Note that in the case that P is a lattice, 1 and 2 specialise to:

1. $Min(S_1 \cap S_2) = Min\,\{x_1 \sqcup x_2 \mid x_1 \in Min(S_1), x_2 \in Min(S_2)\}$;

2. $Max(T_1 \cap T_2) = Max\,\{y_1 \sqcap y_2 \mid y_1 \in Max(T_1), y_2 \in Max(T_2)\}$.

To exploit this result we make the following definition.

Definition 7.3.3 *Let P be a poset. Let $X_1, X_2, Y_1, Y_2 \subseteq P$. The operations $\cap_P^{min}$ and $\cap_P^{max}$ are defined by:*

1. $X_1 \cap_P^{min} X_2 = Min\,(\bigcup\{x_1 \mathbin{\tilde{\vee}} x_2 \mid x_1 \in X_1, x_2 \in X_2\})$;

2. $Y_1 \cap_P^{max} Y_2 = Max\,(\bigcup\{y_1 \mathbin{\tilde{\wedge}} y_2 \mid y_1 \in Y_1, y_2 \in Y_2\})$.

We can now replace the two assignments in the body of the while loop in Algorithm B by

- $B_1 := B_1 \cap_P^{min} (Min(P \setminus {\downarrow}x))$
- $B_0 := B_0 \cap_P^{max} (Max(P \setminus {\uparrow}x))$

In the case that $P = \mathbf{2}^n$, these updates can be seen as corresponding precisely to the 'shine down' and 'shine up' operations of the original frontiers algorithm ([JC87]). When implementing these assignments for a specific poset P we will have to show how $Min(P \setminus {\downarrow}x)$ and $Max(P \setminus {\uparrow}x)$ can be calculated. We postpone the details of this until Chapter 8, for now we just give the operations names:

Definition 7.3.4 *Let P be a poset and let $x \in P$. Then the* upper *and* lower complements of x *are defined respectively as:*

1. $Ucmp_P(x) = Min(P \setminus {\downarrow}x)$;
2. $Lcmp_P(x) = Max(P \setminus {\uparrow}x)$.

When P is clear from the context we will just write $Ucmp(x)$ and $Lcmp(x)$.

Remark Since there is some scope for confusion resulting from our terminology we had better spell out the fact that the *upper* complement of x is the set of *minimal* elements of the *upper* set which is the complement of ${\downarrow}x$. Similarly, the *lower* complement of x is the set of *maximal* elements of the *lower* set which is the complement of ${\uparrow}x$.

Finally, we must consider the test of the while loop and the choosing of an element from ${\downarrow}B_0 \cap {\uparrow}B_1$. Here we can fell two birds with one stone. Define the map $Edges : \mathcal{I}_l(P) \times \mathcal{I}_u(P) \rightarrow \wp(P)$ by

$$Edges(B_0, B_1) = \{x \in B_1 \mid \exists z \in B_0.\, x \sqsubseteq z\} \cup \{z \in B_0 \mid \exists x \in B_1.\, x \sqsubseteq z\}\,.$$

Now let $y \in P$. Then $y \in {\downarrow}B_0 \cap {\uparrow}B_1 \iff \exists x \in B_1. \exists z \in B_0.\, x \sqsubseteq y \sqsubseteq z$. From this it follows that:

1. ${\downarrow}B_0 \cap {\uparrow}B_1 = \emptyset \iff Edges(B_0, B_1) = \emptyset$;
2. $Edges(B_0, B_1) \subseteq {\downarrow}B_0 \cap {\uparrow}B_1$.

Thus the test of the while loop can be replaced by

while $Edges(B_0, B_1) \neq \emptyset$

and the command to choose a value for x can be replaced by

choose x **from** $Edges(B_0, B_1)$.

The final version of the algorithm, the Basic Frontiers Algorithm is shown in Figure 7.3.

B_0 := $Max(P)$
B_1 := $Min(P)$
{ *Invariant: (I1′)* }
while $Edges(B_0, B_1) \neq \emptyset$
  choose x **from** $Edges(B_0, B_1)$
  if $f(x) = 0$
   then B_1 := $B_1 \cap_P^{min} Ucmp(x)$
   else B_0 := $B_0 \cap_P^{max} Lcmp(x)$
endwhile

Figure 7.3: The Basic Frontiers Algorithm

7.4 The Algorithm as a Search

The best way to gain an intuitive understanding of how the frontiers algorithm works is to consider it as a search. Consider Figure 7.4 (a) in relation to Algorithm B of Figure 7.2. The parallel lines indicate the sets B_0 and B_1: think of them as being 'one element thick', reflecting the irredundancy of the sets. The set A_0 is everything below (and including) B_0 and the set A_1 is everything above (and including) B_1. The invariant guarantees that $A_0 \supseteq f^{-1}\{0\}$ and $A_1 \supseteq f^{-1}\{1\}$. The intersection of A_0 and A_1, the space in the middle, represents the *search space* of Algorithm B. This is the set of points for which membership of $f^{-1}\{0\}$ or $f^{-1}\{1\}$ has yet to be determined. The point x is the next point to be chosen from the search space.

Figure 7.4 (b) shows the situation after $f(x)$ has been found to evaluate to 1 and B_0 has been updated accordingly. Because f must map everything above x to 1, the search space has been cut down by 'chopping out' all those points above x.

The Basic Frontiers Algorithm as shown in Figure 7.3, can be understood in essentially the same way. The key difference with respect to the way the search space is traversed, is that x is always chosen from the edges of the search space.

Note that any way we can find of reducing the size of the search space could reduce the time taken to establish the frontiers for a function. When computing the frontiers for f_{n+1} we will already have the frontiers for f_n to hand. Since $f_n \sqsubseteq f_{n+1}$, we know that $\mathbf{F\text{-}0}(f_{n+1}) \leq_l \mathbf{F\text{-}0}(f_n)$, and so initialising B_0 to $\mathbf{F\text{-}0}(f_n)$ rather than $Max(P)$ will give a reduced initial search space while still establishing the invariant. In Chapter 9 we consider methods which can give us upper bounds on the fixed point of a function. The minimum–1–frontier for such an upper bound could be used to initialise B_1 when calculating the frontiers for each f_n.

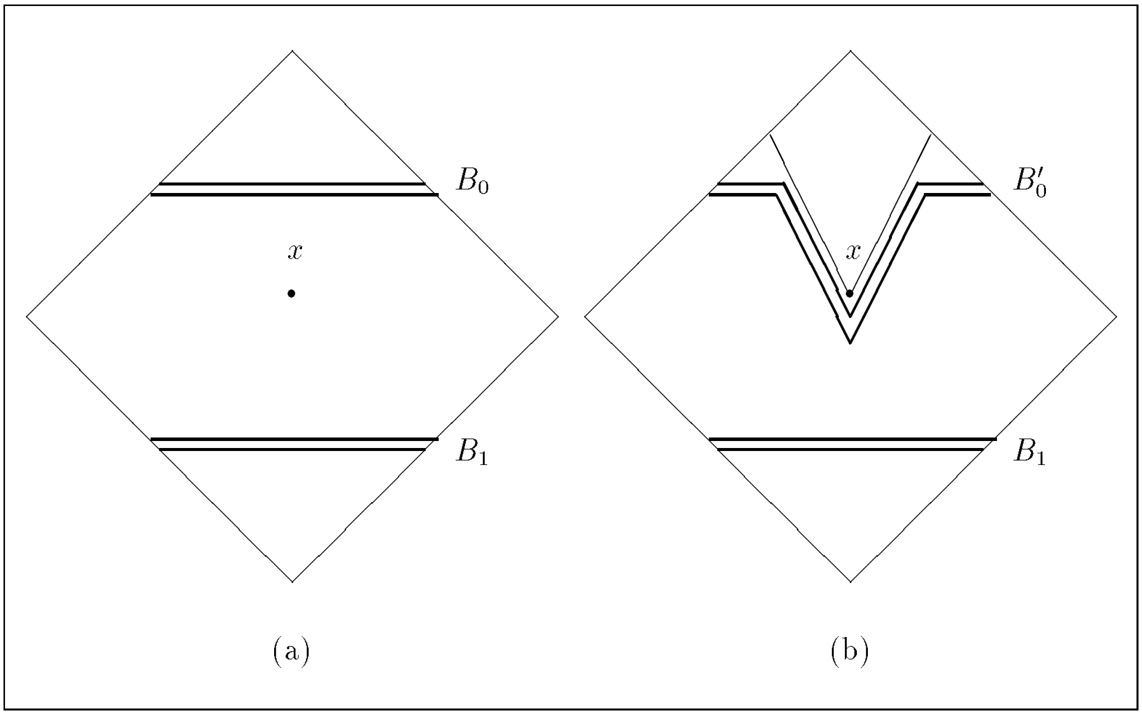

Figure 7.4: The Search Space Before and After Update

Chapter 8

The Generalised Frontiers Algorithm

In [Mar89], the extension of the frontiers representation for functions to cope with functions $f : L_1 \to L_2$ where $L_2 \neq \mathbf{2}$, is achieved by using the family of functions $\{f_a\}_{a \in L_2}$ where each $f_a : L_1 \to \mathbf{2}$ is defined thus:

$$f_a(x) = \begin{cases} 0 & \text{if } f(x) \sqsubseteq a \\ 1 & \text{otherwise} \end{cases}$$

It is clear that each f_a is monotone, and so can be represented using frontiers, and that for any x the value of $f(x)$ can be calculated using the family $\{f_a\}$. In [Hun89] and [HH91] we showed that by working with a restricted family of finite lattices, built up from $\mathbf{2}$ using product, function space and lifting, a more economical factorisation of f into a family of maps from L to $\mathbf{2}$ could be achieved. Here we take a rather different approach, exploiting the fact that the lattices used in all the abstract interpretations we have considered are *distributive*.

8.1 A Brief Review of the Theory of Finite Distributive Lattices

Although we did not state it explicitly, the use of frontiers to represent functions in $[P \to_m \mathbf{2}]$ described in the previous chapter, hinged on the following isomorphisms:

$$\mathcal{L}(P^{\mathrm{OP}}) \cong [P \to_m \mathbf{2}] \cong \mathcal{U}(P^{\mathrm{OP}}), \tag{8.1.1}$$

the first of these isomorphisms being given by

$$S \mapsto \lambda x \in P \,.\, \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } x \notin S \end{cases}$$

with inverse

$$f \mapsto f^{-1}\{1\}\,,$$

and the second isomorphism being dual. As we saw in the previous chapter, in the case that P is finite, the set $f^{-1}\{1\}$ can be represented by its minimal elements $Min(f^{-1}\{1\})$. We will arrive at a novel and powerful generalisation of this basic frontiers representation by viewing (8.1.1) as (in the case of finite P) a special case of a well known representation theorem for *finite distributive lattices.*

Definition 8.1.2 *Let L be a lattice. Then L is* distributive *if for all $x, y, z \in L$*

$$x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z).$$

A lattice L is distributive if and only if L^{OP} is distributive (see [DP90]), so it would be equivalent to define a distributive lattice as one for which

$$x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z).$$

The two point lattice **2** is clearly distributive. The following proposition shows that *all* the finite lattices used in the abstract interpretations considered in this thesis are distributive.

Proposition 8.1.3 *Let P be a poset and let L and K be distributive lattices. Then the following are all distributive lattices:*

1. $L \times K$;

2. $L_\perp$;

3. $L^\top$;

4. $[P \to_m L]$.

Proof The proofs are all simple. We prove 4 by way of example. For all $f, g, h \in [P \to_m L]$, for any $x \in P$:

$$\begin{array}{ll} (f \sqcap (g \sqcup h))\ x & \\ \quad = \ (f\ x) \sqcap ((g \sqcup h)\ x) & \\ \quad = \ (f\ x) \sqcap ((g\ x) \sqcup (h\ x)) & \\ \quad = \ ((f\ x) \sqcap (g\ x)) \sqcup ((f\ x) \sqcap (h\ x)) & \text{since } L \text{ is distributive} \\ \quad = \ ((f \sqcap g)\ x) \sqcup ((f \sqcap h)\ x) & \\ \quad = \ ((f \sqcap g) \sqcup (f \sqcap h))\ x. & \end{array}$$

$\square$

The representation theorem we shall be exploiting uses the the following class of elements:

Definition 8.1.4 *Let L be a lattice and let $x \in L$. Then x is* join irreducible *if*

1. *$x \neq \perp_L$ (in the case that $\perp_L$ exists);*

2. *for all $x', x'' \in L$, if $x = x' \sqcup x''$ then $x = x'$ or $x = x''$.*

The meet irreducible *elements of L are defined dually. The poset of join irreducible elements of L with order inherited from L, is written $\mathcal{J}(L)$. The poset of meet irreducible elements, again with order inherited from L, is written $\mathcal{M}(L)$. Thus $\mathcal{M}(L) = \mathcal{J}(L^{\mathrm{OP}})^{\mathrm{OP}}$.*

An extremely useful alternative characterisation of the irreducible elements for distributive lattices is given by the following result.

Lemma 8.1.5 *Let L be a distributive lattice and let $x \in L$ with $x \neq \perp_L$ (if $\perp_L$ exists). Then x is join irreducible if and only if for all non-empty finite $X \subseteq L$*

$$x \sqsubseteq \bigsqcup X \Rightarrow \exists x' \in X.\, x \sqsubseteq x'.$$

Dually, x is meet irreducible if and only if for all non-empty finite $X \subseteq L$

$$\bigsqcap X \sqsubseteq x \Rightarrow \exists x' \in X.\, x' \sqsubseteq x.$$

Proof See [DP90]. $\square$

The following representation theorem and its companion, Theorem 8.1.7, place the class of finite distributive lattices and the class of finite posets in one-one correspondence (up to isomorphism), via the maps $L \mapsto \mathcal{J}(L)$ and $P \mapsto \mathcal{L}(P)$. Proofs of both theorems are described in [DP90].

Theorem 8.1.6 (Birkhoff's Representation Theorem for Finite Distributive Lattices) *Let L be a finite distributive lattice. Then:*

1. $L \cong \mathcal{L}(\mathcal{J}(L))$;
2. $L \cong \mathcal{U}(\mathcal{M}(L))$.

Note that 2 follows by duality from 1. The isomorphisms are respectively:

1. $x \mapsto \{j \in \mathcal{J}(L) \mid j \sqsubseteq x\}$, *with inverse* $X \mapsto \bigsqcup X$;
2. $x \mapsto \{m \in \mathcal{M}(L) \mid x \sqsubseteq m\}$, *with inverse* $Y \mapsto \bigsqcap Y$.

Theorem 8.1.7 *Let P be a finite poset. Then:*

1. $P \cong \mathcal{J}(\mathcal{L}(P))$;
2. $P \cong \mathcal{M}(\mathcal{U}(P))$.

Now recall from Section 1.5 that for any poset P, the lattices $\mathcal{L}(P)$ and $\mathcal{U}(P)$ are isomorphic. Thus by Theorem 8.1.6, if L is a finite distributive lattice, then $\mathcal{L}(\mathcal{M}(L)) \cong \mathcal{U}(\mathcal{M}(L)) \cong L \cong \mathcal{L}(\mathcal{J}(L))$, hence $\mathcal{L}(\mathcal{M}(L)) \cong \mathcal{L}(\mathcal{J}(L))$. But clearly, if $A \cong B$ then $\mathcal{J}(A) \cong \mathcal{J}(B)$, so by Theorem 8.1.7, $\mathcal{M}(L) \cong \mathcal{J}(\mathcal{L}(\mathcal{M}(L))) \cong \mathcal{J}(\mathcal{L}(\mathcal{J}(L))) \cong \mathcal{J}(L)$. Thus $\mathcal{M}(L)$ and $\mathcal{J}(L)$ are isomorphic. In the Appendix we show that this isomorphism is established by the following pair of maps, which play a central role in our frontiers algorithm:

Definition 8.1.8 *Let L be a finite distributive lattice. Then the maps $JtoM : \mathcal{J}(L) \to \mathcal{M}(L)$ and $MtoJ : \mathcal{M}(L) \to \mathcal{J}(L)$ are defined as follows:*

1. $JtoM(j) = \bigsqcup(L \setminus \uparrow j)$;
2. $MtoJ(m) = \bigsqcap(L \setminus \downarrow m)$

(where $\uparrow j$ and $\downarrow m$ are both calculated in L).

The key property of this pair of maps, formalised by the following result, is the rather surprising fact that for any join irreducible element j of a finite distributive lattice L, the sets $\uparrow j$ and $\downarrow JtoM(j)$ partition L. Because $JtoM$ and $MtoJ$ form an isomorphism, it is equivalent to observe that for any meet irreducible element m, the sets $\downarrow m$ and $\uparrow MtoJ(m)$ also partition L.

Proposition 8.1.9 *Let L be a finite distributive lattice, let $j \in \mathcal{J}(L)$ and let $m \in \mathcal{M}(L)$. Then:*

1. $L \setminus \uparrow j = \downarrow JtoM(j)$;

2. $L \setminus \downarrow m = \uparrow MtoJ(m)$,

where each $\uparrow$ and $\downarrow$ is calculated in L.

Proof See Appendix. □

8.2 A Generalised Frontiers Representation

Our claim that the use of lower and upper subsets of P to represent functions in $[P \to_m \mathbf{2}]$ is a special case of Birkhoff's Representation Theorem is based on the fact that $\mathcal{J}([P \to_m \mathbf{2}]) \cong P^{\mathrm{OP}}$. To see that this is so we need the following result, which characterises the join and meet irreducible elements of finite distributive lattices of the form $[P \to_m L]$ in terms of the step and co-step functions (the step and co-step functions were defined in Chapter 3, Definition 3.4.4).

Theorem 8.2.1 *Let P be a finite poset and let L be a finite distributive lattice. Then:*

1. $\mathcal{J}([P \to_m L]) = \{\lceil x, j \rceil \mid x \in P, j \in \mathcal{J}(L)\}$;

2. $\mathcal{M}([P \to_m L]) = \{\lfloor x, m \rfloor \mid x \in P, m \in \mathcal{M}(L)\}$.

Proof See Appendix. □

The corollary to the following lemma then gives us the isomorphism we seek.

Lemma 8.2.2 *Let P be a poset and let L be a distributive lattice.*

1. *For any $x, x' \in P$ and $j, j' \in \mathcal{J}(L)$:*

$$\lceil x, j \rceil \sqsubseteq \lceil x', j' \rceil \iff x' \sqsubseteq x \text{ and } j \sqsubseteq j'.$$

2. *For any $x, x' \in P$ and $m, m' \in \mathcal{M}(L)$:*

$$\lfloor x, m \rfloor \sqsubseteq \lfloor x', m' \rfloor \iff x' \sqsubseteq x \text{ and } m \sqsubseteq m'.$$

Proof We prove the first directly. The second is dual.

Let $x, x', x'' \in P$ and let $j, j' \in \mathcal{J}(L)$. For the implication from right to left, assume $x' \sqsubseteq x$ and $j \sqsubseteq j'$. Then there are two cases: if $x \sqsubseteq x''$, then by assumption $x' \sqsubseteq x''$, and so $\lceil x, j \rceil (x'') = j \sqsubseteq j' = \lceil x', j' \rceil (x'')$; if $x \not\sqsubseteq x''$, then $\lceil x, j \rceil (x'') = \bot$. For the implication from left to right, assume $\lceil x, j \rceil \sqsubseteq \lceil x', j' \rceil$. We have $j = \lceil x, j \rceil (x) \sqsubseteq \lceil x', j' \rceil (x)$. Since j is join irreducible we know that $j \neq \bot$, so $\lceil x', j' \rceil (x) \neq \bot$. Hence $x' \sqsubseteq x$ and $\lceil x', j' \rceil (x) = j' \sqsupseteq j$. □

Corollary 8.2.3 *Let P be a finite poset. Let L be finite distributive lattice. Then:*

1. $\mathcal{J}([P \to_m L]) \cong P^{\mathrm{OP}} \times \mathcal{J}(L)$;
2. $\mathcal{M}([P \to_m L]) \cong P^{\mathrm{OP}} \times \mathcal{M}(L)$.

Now $\mathcal{J}(\mathbf{2})$ is just $\{1\}$, so we have $\mathcal{J}([P \to_m \mathbf{2}]) \cong P^{\mathrm{OP}}$. Thus the isomorphisms (8.1.1) on which the frontier representation of $f \in [P \to_m \mathbf{2}]$ is based, can be seen as special cases of Birkhoff's Representation Theorem.

Let L be a finite distributive lattice and let $f \in L$ (we will be particularly interested in the case that L is a function space, but what follows is quite general). The *meet-frontier* and *join-frontier* for f are defined respectively as:

1. $\mathbf{F}^{\wedge}(f) = Min\,\{m \in \mathcal{M}(L) \mid f \sqsubseteq m\}$;
2. $\mathbf{F}^{\vee}(f) = Max\,\{j \in \mathcal{J}(L) \mid j \sqsubseteq f\}$.

Thus $\uparrow\mathbf{F}^{\wedge}(f)$ is the representation of f in $\mathcal{U}(\mathcal{M}(L))$ given by Theorem 8.1.6, and $\downarrow\mathbf{F}^{\vee}(f)$ is the dual representation of f in $\mathcal{L}(\mathcal{J}(L))$. It is clear that for any subset X of a finite lattice, $\bigsqcap Min(X) = \bigsqcap X$ and $\bigsqcup Max(X) = \bigsqcup X$. Hence, by Theorem 8.1.6, $\bigsqcap \mathbf{F}^{\wedge}(f) = f = \bigsqcup \mathbf{F}^{\vee}(f)$. Since for finite posets the irredundant sets and the upper sets are in one-one correspondence, it also follows from Theorem 8.1.6 that $\mathbf{F}^{\wedge}(f)$ and $\mathbf{F}^{\vee}(f)$ are the *unique* irredundant sets satisfying these equations.

In what follows, we make extensive use of the intersection operations $\cap^{min}_{\mathcal{M}(L)}$ and $\cap^{max}_{\mathcal{J}(L)}$. It is helpful to bear in mind the following result, which shows that is often possible to understand these operations as implementing meets and joins.

Proposition 8.2.4 *Let L be a finite distributive lattice and let $f, f' \in L$. Then:*

1. $\mathbf{F}^{\wedge}(f) \cap^{min}_{\mathcal{M}(L)} \mathbf{F}^{\wedge}(f') = \mathbf{F}^{\wedge}(f \sqcup f')$;
2. $\mathbf{F}^{\vee}(f) \cap^{max}_{\mathcal{J}(L)} \mathbf{F}^{\vee}(f') = \mathbf{F}^{\vee}(f \sqcap f')$.

Proof Consider the first case (the second is dual): $\uparrow\mathbf{F}^{\wedge}(f)$ is the representation of f in $\mathcal{U}(\mathcal{M}(L))$ given by Birkhoff's Representation Theorem, and similarly for f'. Then, since joins in $\mathcal{U}(\mathcal{M}(L))$ are given by intersections, the representation of $f \sqcup f'$ in $\mathcal{U}(\mathcal{M}(L))$ is just:

$$\uparrow\mathbf{F}^{\wedge}(f) \cap \uparrow\mathbf{F}^{\wedge}(f').$$

But by Proposition 7.3.2, $\mathbf{F}^{\wedge}(f) \cap^{min}_{\mathcal{M}(L)} \mathbf{F}^{\wedge}(f')$ is precisely the set of minimal elements of this set. $\square$

The correspondence with minimum-1-frontiers and maximum-0-frontiers for $f \in [P \to_m \mathbf{2}]$ is between $\mathbf{F}^{\wedge}(f)$ and $\mathbf{F\text{-}0}(f)$ on the one hand and between $\mathbf{F}^{\vee}(f)$ and $\mathbf{F\text{-}1}(f)$ on the other. Thus

$$\mathbf{F}^{\wedge}(f) = \{\lfloor x, 0\rfloor \mid x \in \mathbf{F\text{-}0}(f)\}$$

and

$$\mathbf{F}^{\vee}(f) = \{\lceil x, 1\rceil \mid x \in \mathbf{F\text{-}1}(f)\}.$$

Note that $\mathbf{F}^{\wedge}(f)$ is defined as a set of *minimal* elements but $\mathbf{F\text{-}0}(f)$ is defined as a set of *maximal* elements. The reason for this reversal in order is simply that $\lfloor x, 0\rfloor \sqsubseteq \lfloor x', 0\rfloor \iff x' \sqsubseteq x$ (Lemma 8.2.2).

8.3 The Generalised Frontiers Algorithm

In this section we develop an algorithm, shown in Figure 8.1, which constructs $\mathbf{F}^{\vee}(f)$ and $\mathbf{F}^{\wedge}(f)$ in a way which is analogous to the construction of the minimum-1 and maximum-0 frontiers by the Basic Frontiers Algorithm.

Let $J \subseteq \mathcal{J}(L)$ and $M \subseteq \mathcal{M}(L)$ be irredundant sets such that $\sqcap M \sqsubseteq f$ and $f \sqsubseteq \bigsqcup J$ (without knowing f we can achieve this by setting $M = \mathbf{F}^{\wedge}(\bot_L)$ and $J = \mathbf{F}^{\vee}(\top_L)$). These inequalities may be expressed as $\uparrow\mathbf{F}^{\wedge}(f) \subseteq \uparrow M$ and $\downarrow\mathbf{F}^{\vee}(f) \subseteq \downarrow J$. While maintaining irredundancy and the inequalities as invariants, we wish to progressively refine J and M (as subsets of $\mathcal{J}(L)$ and $\mathcal{M}(L)$) until $\bigsqcup J \sqsubseteq \sqcap M$. When this point is reached we will have irredundant sets $J \subseteq \mathcal{J}(L)$ and $M \subseteq \mathcal{M}(L)$ such that $\bigsqcup J = \sqcap M = f$, in other words $J = \mathbf{F}^{\vee}(f)$ and $M = \mathbf{F}^{\wedge}(f)$.

We define an operation $\mathit{Disag} : \mathcal{I}_l(\mathcal{J}(L)) \times \mathcal{I}_u(\mathcal{M}(L)) \to \wp(\mathcal{J}(L) + \mathcal{M}(L))$ to play the role of the *Edges* operation in the Basic Frontiers Algorithm.

$$\mathit{Disag}(J, M) = \{j \in J \mid \exists m \in M. j \not\sqsubseteq m\} + \{m \in M \mid \exists j \in J. j \not\sqsubseteq m\},$$

M := $\mathbf{F}^{\wedge}(\bot_L)$
J := $\mathbf{F}^{\vee}(\top_L)$
{ *Invariant:* $\bigsqcap M \sqsubseteq f \sqsubseteq \bigsqcup J$ *and* M *and* J *irredundant.* }
while $Disag(J, M) \neq \emptyset$
  choose y **from** $Disag(J, M)$
  case y **of**
    $in_1(j)$: **if** $j \sqsubseteq f$
      then M := $Raise(M, JtoM(j))$
      else J := $Lower(J, j)$
    $in_2(m)$: **if** $f \sqsubseteq m$
      then J := $Lower(J, MtoJ(m))$
      else M := $Raise(M, m)$
  endcase
endwhile

Figure 8.1: The Generalised Frontiers Algorithm

where + here is simply disjoint union of the underlying sets. The set $Disag(J, M)$ may be thought of as a set of 'disagreements' between J and M, the two putative descriptions of f. By the definition of join, $\bigsqcup J \sqsubseteq m \iff \forall j \in J. j \sqsubseteq m$ and similarly $j \sqsubseteq \bigsqcap M \iff \forall m \in M. j \sqsubseteq m$. From this observation it follows that

$$\bigsqcup J \sqsubseteq \bigsqcap M \iff Disag(J, M) = \emptyset,$$

thus $Disag(J, M) = \emptyset$ is the condition for termination of our algorithm.

Now suppose $in_1(j) \in Disag(J, M)$. There are two cases to consider.

1. If $j \sqsubseteq f$ then $\mathbf{F}^{\wedge}(f) \subseteq \uparrow j$ (since $m \sqsupseteq f \sqsupseteq j$ for every $m \in \mathbf{F}^{\wedge}(f)$) so M can be updated to take the new value

$$Min(\uparrow M \cap \uparrow j)$$

(where $\uparrow M$ is calculated in $\mathcal{M}(L)$). Note that $\uparrow M \cap \uparrow j$ is guaranteed to be strictly smaller than $\uparrow M$ since $in_1(j) \in Disag(J, M)$ implies that $\exists m \in M. j \not\sqsubseteq m$. By Proposition 8.1.9 we have that $x \in \uparrow j \iff x \not\sqsubseteq JtoM(j)$, from which it is easy to see that the above is equal to

$$Min(\uparrow M \cap (\mathcal{M}(L) \setminus \downarrow JtoM(j))).$$

Now $Min\uparrow M = M$, since M is irredundant and, by Definition 7.3.4, $Min(\mathcal{M}(L)\setminus$

$\downarrow JtoM(j)) = Ucmp_{\mathcal{M}(L)}(JtoM(j))$.[1] Thus, using Proposition 7.3.2 and Definition 7.3.3 we arrive at the following assignment to update M:

$$M := M \cap^{min}_{\mathcal{M}(L)} Ucmp_{\mathcal{M}(L)}(JtoM(j)).$$

2. If $j \not\sqsubseteq f$ then $\mathbf{F}^{\vee}(f) \subseteq \mathcal{J}(L) \setminus \uparrow j$ (since $j' \sqsubseteq f$ for every $j' \in \mathbf{F}^{\vee}(f)$ and so $j' \in \mathbf{F}^{\vee}(f) \Rightarrow j \not\sqsubseteq j'$). Thus J can be updated to take the value

$$Max(\downarrow J \cap (\mathcal{J}(L) \setminus \uparrow j)).$$

The set $\downarrow J \cap (\mathcal{J}(L) \setminus \uparrow j)$ will be strictly smaller than $\downarrow J$ since $j \in J$ but $j \notin \mathcal{J}(L) \setminus \uparrow j$. A dual argument to that used in the previous case, leads to the assignment:

$$J := J \cap^{max}_{\mathcal{J}(L)} Lcmp_{\mathcal{J}(L)}(j).$$

Dually, if $in_2(m) \in Disag(J, M)$ the two cases are

1. $f \sqsubseteq m$, in which case J is updated: $J := J \cap^{max}_{\mathcal{J}(L)} Lcmp_{\mathcal{J}(L)}(MtoJ(m))$;

2. $f \not\sqsubseteq m$, in which case M is updated: $M := M \cap^{min}_{\mathcal{M}(L)} Ucmp_{\mathcal{M}(L)}(m)$.

We encapsulate the update operations on J and M in the following definitions:

1. $Raise : \mathcal{I}_u(\mathcal{M}(L)) \times \mathcal{M}(L) \to \mathcal{I}_u(\mathcal{M}(L))$
   $Raise(M, m) = M \cap^{min}_{\mathcal{M}(L)} Ucmp_{\mathcal{M}(L)}(m)$;

2. $Lower : \mathcal{I}_l(\mathcal{J}(L)) \times \mathcal{J}(L) \to \mathcal{I}_l(\mathcal{J}(L))$
   $Lower(J, j) = J \cap^{max}_{\mathcal{J}(L)} Lcmp_{\mathcal{J}(L)}(j)$.

The resulting algorithm is shown in Figure 8.1.

8.3.1 Implementing the Tests

It is worth considering how the tests for $j \sqsubseteq f$ and $f \sqsubseteq m$ are to be implemented, assuming that f is the abstract interpretation of some term. Suppose that f is $[\![\lambda\, x\, .\, e]\!]^J \delta$. Then deciding whether $\lceil a, j \rceil \sqsubseteq f$ reduces to deciding whether $j \sqsubseteq$

[1] Corollary 8.4.6 shows that this is just $\mathbf{F}^{\wedge}(j)$. Seen in this light, the update we arrive at amounts to replacing M by $\mathbf{F}^{\wedge}(f_M \sqcup j)$ where f_M is the function represented by M, i.e., $f_M = \bigsqcap M$. This makes a lot of sense since, by the invariant, $f_M \sqsubseteq f$ and, from the test, $j \sqsubseteq f$: thus $f_M \sqcup j \sqsubseteq f$. The other updates can be explained similarly.

$$\mathbf{2} \in \mathcal{A}$$

$$\frac{A \in \mathcal{A}}{A_\bot \in \mathcal{A}} \qquad \frac{A \in \mathcal{A}}{A^\top \in \mathcal{A}}$$

$$\frac{A_1 \in \mathcal{A} \ \cdots \ A_n \in \mathcal{A}}{A_1 \times \cdots \times A_n \in \mathcal{A}}$$

$$\frac{A_1 \in \mathcal{A} \qquad A_2 \in \mathcal{A}}{[A_1 \to_m A_2] \in \mathcal{A}}$$

Figure 8.2: A Family of Finite Distributive Lattices

$[\![e]\!]^J \delta'$, where $\delta' = \delta[x \mapsto a]$. Since the abstract interpretations of constants are typically defined in terms of meets and joins, this in turn will often reduce either to the problem of deciding whether $j \sqsubseteq ([\![e_1]\!]^J \delta') \sqcap ([\![e_2]\!]^J \delta')$, or to deciding whether $j \sqsubseteq ([\![e_1]\!]^J \delta') \sqcup ([\![e_2]\!]^J \delta')$. By the definition of meet, the former case is equivalent to the conjunction of tests for $j \sqsubseteq [\![e_1]\!]^J \delta'$ and $j \sqsubseteq [\![e_2]\!]^J \delta'$: if the one we try first fails then we can avoid the other test altogether. For the latter case we can exploit the join irreducibility of j by using Lemma 8.1.5, which shows the test to be equivalent to the *disjunction* of tests for $j \sqsubseteq [\![e_1]\!]^J \delta'$ and $j \sqsubseteq [\![e_2]\!]^J \delta'$: if the one we try first *succeeds* then we can avoid the other test altogether. The consideration of tests $f \sqsubseteq m$ is dual.

8.4 The Operations of the Algorithm

To implement the Generalised Frontiers Algorithm, and to compute with frontiers in an abstract interpretation, there are a number of operations which we must describe in a form suitable for an implementation. We begin by defining a family of finite distributive lattices which is sufficiently rich to enable us to implement any of the abstract interpretations considered in this thesis. The family $\mathcal{A}$ is inductively defined as shown in Figure 8.2. Note that the rule which allows us to conclude that $A \in \mathcal{A}$, is always unique. We will often appeal to induction on the structure of $A \in \mathcal{A}$ when strictly speaking we mean induction on the height of the proof that $A \in \mathcal{A}$.

By Lemma 8.1.3 every member of $\mathcal{A}$ is a finite distributive lattice. We are therefore able to represent the elements of these lattices by their meet and join frontiers. First we must describe $\mathcal{J}(A)$ and $\mathcal{M}(A)$ for each $A \in \mathcal{A}$.

Figure 8.3 shows an inductive definition of the meet and join irreducible elements

$$\mathcal{J}(\mathbf{2}) = \{1\} \qquad \mathcal{M}(\mathbf{2}) = \{0\}$$

$$\begin{array}{rcl} \mathcal{J}(A_\perp) & = & \{\mathit{lift}(\perp_A)\} \cup \{\mathit{lift}(j) \mid j \in \mathcal{J}(A)\} \\ \mathcal{M}(A_\perp) & = & \{\perp\} \cup \{\mathit{lift}(m) \mid m \in \mathcal{M}(A)\} \end{array}$$

$$\begin{array}{rcl} \mathcal{J}(A^\top) & = & \{\top\} \cup \{\mathit{colift}(j) \mid j \in \mathcal{J}(A)\} \\ \mathcal{M}(A^\top) & = & \{\mathit{colift}(\top_A)\} \cup \{\mathit{colift}(m) \mid m \in \mathcal{M}(A)\} \end{array}$$

$$\begin{array}{rcl} \mathcal{J}(A_1 \times \cdots \times A_n) & = & \bigcup_{i=1}^{n} \{(\perp_1, \ldots, \perp_{i-1}, j, \perp_{i+1}, \ldots, \perp_n) \mid j \in \mathcal{J}(A_i)\} \\ \mathcal{M}(A_1 \times \cdots \times A_n) & = & \bigcup_{i=1}^{n} \{(\top_1, \ldots, \top_{i-1}, m, \top_{i+1}, \ldots, \top_n) \mid m \in \mathcal{M}(A_i)\} \end{array}$$

where $\perp_k \equiv \perp_{A_k}$ and $\top_k \equiv \top_{A_k}$

$$\begin{array}{rcl} \mathcal{J}([A \to_m B]) & = & \{\lceil a, j \rceil \mid a \in A, j \in \mathcal{J}(B)\} \\ \mathcal{M}([A \to_m B]) & = & \{\lfloor a, m \rfloor \mid a \in A, m \in \mathcal{M}(B)\} \end{array}$$

Figure 8.3: Join and Meet Irreducible Elements for $A \in \mathcal{A}$

of every member of $\mathcal{A}$. We have already explained the characterisation of the join and meet irreducible elements for function spaces (Theorem 8.2.1). Except for products, the remaining cases are straightforward. To understand the case for products, let $(a_1, \ldots, a_n)$ be join irreducible. Since $\perp$ is not join irreducible, at least one component, say a_i, must be non-$\perp$. Now suppose that a_k is also non-$\perp$, with $i \neq k$. But then $(a_1, \ldots, a_{k-1}, \perp, a_{k+1}, \ldots, a_n)$ and $(a_1, \ldots, a_{i-1}, \perp, a_{i+1}, \ldots, a_n)$ are distinct tuples, neither is equal to $(a_1, \ldots, a_n)$, and yet they have $(a_1, \ldots, a_n)$ as their join: this contradicts the join irreducibility of $(a_1, \ldots, a_n)$. Thus all join irreducible tuples must be of the form $(\perp, \ldots, \perp, j, \perp, \ldots, \perp)$, and it is easy to see that such a tuple is join irreducible if and only if j is.

The operations required for computing with frontiers for elements in A fall into two camps: the 'micro' operations on the elements of $\mathcal{J}(A)$ and $\mathcal{M}(A)$ themselves, and the 'macro' operations on the (representations of) elements of A. We consider these in turn.

8.4.1 Operations on $\mathcal{J}(A)$ and $\mathcal{M}(A)$

Comparisons

Given $j, j' \in \mathcal{J}(A)$, we need a method for deciding whether $j \sqsubseteq j'$. For the function case we use Lemma 8.2.2: assuming $\lceil x, j \rceil$ and $\lceil x', j' \rceil$ to be represented by the pairs (x, j) and (x', j'), comparison is simply given by

$$\lceil x, j \rceil \sqsubseteq \lceil x', j' \rceil \iff x' \sqsubseteq x \text{ and } j \sqsubseteq j'.$$

The other cases are straightforward.

Minimal Upper Bounds and Maximal Lower Bounds

Figure 8.4 gives an inductive definition of the minimal upper bounds and maximal lower bounds of pairs of elements in $\mathcal{J}(A)$ for each $A \in \mathcal{A}$. Symmetrical cases are left implicit. Observe that for each $A \in \mathcal{A}$, for any $j, j' \in \mathcal{J}(A)$, the sets $j \mathbin{\tilde{\vee}} j'$ and $j \mathbin{\tilde{\wedge}} j'$ are either empty or singleton (proof by routine induction on A). This is because for each $A \in \mathcal{A}$, the poset $\mathcal{J}(A)$ is isomorphic to a disjoint sum of lattices. The inductive definition for $\mathcal{M}(A)$ is dual. The function case is justified by appeal to the isomorphism $\mathcal{J}([A \to_m B]) \cong A^{\mathrm{OP}} \times \mathcal{J}(B)$ (Corollary 8.2.3). Note that the definition of $\lceil a, j \rceil \mathbin{\tilde{\vee}} \lceil a', j' \rceil$ depends on being able to calculate $a \sqcap a'$. Assuming a and a' to be represented by frontiers, this is covered by Lemma 8.2.4.

The Isomorphisms $\mathcal{J}(A) \cong \mathcal{M}(A)$

Figure 8.5 gives an inductive definition of *JtoM* and *MtoJ* for each $A \in \mathcal{A}$. As usual, the less obvious case is that for functions. It suffices to show that $\lfloor a, JtoM(j) \rfloor$ is the greatest meet irreducible element of $[A \to_m B]$ which is not above $\lceil a, j \rceil$. To do this we use the corollary to the following lemma.

Lemma 8.4.1 *Let P be a poset and let L be a lattice. Let $x, x' \in P$ and let $y, y' \in L$ with $y \neq \bot$. Then*

$$\lceil x, y \rceil \sqsubseteq \lfloor x', y' \rfloor \iff (x \sqsubseteq x' \Rightarrow y \sqsubseteq y').$$

Proof To show the implication from right to left, assume $x \sqsubseteq x' \Rightarrow y \sqsubseteq y'$. We must show that for any a, $\lceil x, y \rceil (a) \sqsubseteq \lfloor x', y' \rfloor (a)$. We need only consider the case when $\lceil x, y \rceil (a) \neq \bot$ and $\lfloor x', y' \rfloor (a) \neq \top$, i.e., when $x \sqsubseteq a \sqsubseteq x'$. But then by assumption, $y \sqsubseteq y'$.

2

$$1 \mathbin{\tilde{\vee}} 1 = \{1\} \qquad 1 \mathbin{\tilde{\wedge}} 1 = \{1\}$$

$A_\perp$

$$\begin{array}{rcll}
\mathit{lift}(\perp_A) & \tilde{\vee} & j & = j \\
\mathit{lift}(j) & \tilde{\vee} & \mathit{lift}(j') & = \{\mathit{lift}(j'') \mid j'' \in j \mathbin{\tilde{\vee}} j'\}, \qquad j, j' \neq \perp_A \\
\\
\mathit{lift}(\perp_A) & \tilde{\wedge} & j & = \mathit{lift}(\perp_A) \\
\mathit{lift}(j) & \tilde{\wedge} & \mathit{lift}(j') & = \{\mathit{lift}(j'') \mid j'' \in j \mathbin{\tilde{\wedge}} j'\}, \qquad j, j' \neq \perp_A
\end{array}$$

$A^\top$

$$\begin{array}{rcll}
\top & \tilde{\vee} & j & = \top \\
\mathit{colift}(j) & \tilde{\vee} & \mathit{colift}(j') & = \{\mathit{colift}(j'') \mid j'' \in j \mathbin{\tilde{\vee}} j'\} \\
\\
\top & \tilde{\wedge} & j & = j \\
\mathit{colift}(j) & \tilde{\wedge} & \mathit{colift}(j') & = \{\mathit{colift}(j'') \mid j'' \in j \mathbin{\tilde{\wedge}} j'\}
\end{array}$$

$A_1 \times \cdots \times A_n$

$$(\perp_1, \ldots, \perp_{i-1}, j, \perp_{i+1}, \ldots, \perp_n) \mathbin{\tilde{\vee}} (\perp_1, \ldots, \perp_{i'-1}, j', \perp_{i'+1}, \ldots, \perp_n)$$
$$= \begin{cases} \emptyset & \text{if } i \neq i' \\ \{(\perp_1, \ldots, \perp_{i-1}, j'', \perp_{i+1}, \ldots, \perp_n) \mid j'' \in j \mathbin{\tilde{\vee}} j'\} & \text{if } i = i' \end{cases}$$

$$(\perp_1, \ldots, \perp_{i-1}, j, \perp_{i+1}, \ldots, \perp_n) \mathbin{\tilde{\wedge}} (\perp_1, \ldots, \perp_{i'-1}, j', \perp_{i'+1}, \ldots, \perp_n)$$
$$= \begin{cases} \emptyset & \text{if } i \neq i' \\ \{(\perp_1, \ldots, \perp_{i-1}, j'', \perp_{i+1}, \ldots, \perp_n) \mid j'' \in j \mathbin{\tilde{\wedge}} j'\} & \text{if } i = i' \end{cases}$$

where $\perp_k \equiv \perp_{A_k}$

$[A \to_m B]$

$$\lceil a, j \rceil \;\tilde{\vee}\; \lceil a', j' \rceil = \{\lceil a \sqcap a', j'' \rceil \mid j'' \in j \mathbin{\tilde{\vee}} j'\}$$

$$\lceil a, j \rceil \;\tilde{\wedge}\; \lceil a', j' \rceil = \{\lceil a \sqcup a', j'' \rceil \mid j'' \in j \mathbin{\tilde{\wedge}} j'\}$$

Figure 8.4: Minimal Upper Bounds and Maximal Lower Bounds in $\mathcal{J}(A)$

2
$JtoM(1) = 0 \qquad MtoJ(0) = 1$
$A_\perp$
$\begin{array}{lcl} JtoM(\mathit{lift}(\perp_A)) & = & \perp \\ JtoM(\mathit{lift}(j)) & = & \mathit{lift}(JtoM(j)), \quad j \neq \perp_A \\ & & \\ MtoJ(\perp) & = & \mathit{lift}(\perp_A) \\ MtoJ(\mathit{lift}(m)) & = & \mathit{lift}(MtoJ(m)) \end{array}$
$A^\top$
$\begin{array}{lcl} JtoM(\top) & = & \mathit{colift}(\top_A) \\ JtoM(\mathit{colift}(j)) & = & \mathit{colift}(JtoM(j)) \\ & & \\ MtoJ(\mathit{colift}(\top_A)) & = & \top \\ MtoJ(\mathit{colift}(m)) & = & \mathit{colift}(MtoJ(m)), \quad m \neq \top_A \end{array}$
$A_1 \times \cdots \times A_n$
$JtoM(\perp_1, \ldots, \perp_{i-1}, j, \perp_{i+1}, \ldots, \perp_n) = (\top_1, \ldots, \top_{i-1}, JtoM(j), \top_{i+1}, \ldots, \top_n)$ $MtoJ(\top_1, \ldots, \top_{i-1}, m, \top_{i+1}, \ldots, \top_n) = (\perp_1, \ldots, \perp_{i-1}, MtoJ(m), \perp_{i+1}, \ldots, \perp_n)$ where $\perp_k \equiv \perp_{A_k}$ and $\top_k \equiv \top_{A_k}$
$[A \to_m B]$
$JtoM(\lceil a, j \rceil) = \lfloor a, JtoM(j) \rfloor$ $MtoJ(\lfloor a, m \rfloor) = \lceil a, MtoJ(m) \rceil$

Figure 8.5: Components of the Isomorphism $\mathcal{J}(A) \cong \mathcal{M}(A)$

To show the implication from left to right, assume $\lceil x, y\rceil \sqsubseteq \lfloor x', y'\rfloor$, and $x \sqsubseteq x'$. We must show show that $y \sqsubseteq y'$. This is simple since $\lceil x, y\rceil (x') = y \sqsubseteq \lfloor x', y'\rfloor (x') = y'$. □

Corollary 8.4.2 *Let P, L, x, x', y and y' be given as in the statement of the Lemma. Then*

$$\lceil x, y\rceil \not\sqsubseteq \lfloor x', y'\rfloor \iff x \sqsubseteq x' \text{ and } y \not\sqsubseteq y'.$$

Proof This is just the contraposition of the Lemma. □

This corollary, together with Lemma 8.2.2 shows that to maximise $\lfloor a', m\rfloor$ such that $\lceil a, j\rceil \not\sqsubseteq \lfloor a', m\rfloor$, we minimise $a' \in A$ such that $a \sqsubseteq a'$ and maximise $m \in \mathcal{M}(B)$ such that $j \not\sqsubseteq m$. But then a' is clearly just a and by the definition of *JtoM*, m must be $JtoM(j)$.

Upper and Lower Complements

Figure 8.6 gives an inductive definition of $Ucmp_{\mathcal{M}(A)}$ for each $A \in \mathcal{A}$. The definition of $Lcmp_{\mathcal{J}(A)}$ is dual. The definitions in Figure 8.6 assume a method for calculating the meet frontier of $\bot_A$ and the join frontier of $\top_A$, for each $A \in \mathcal{A}$. We do not give the details, but they are straightforward. The maps $M_i : \mathcal{I}_u(\mathcal{M}(A_i)) \to \mathcal{I}_u(\mathcal{M}(A_1 \times \cdots \times A_n))$, used in the product case, are defined by:

$$M_i(X) = \{(\top_1, \ldots, \top_{i-1}, m, \top_{i+1}, \ldots, \top_n) \mid m \in X\},$$

where $\top_k \equiv \top_{A_k}$.

Note that for function spaces the definition of $Ucmp_{\mathcal{M}([A \to_m B])}$ depends on that of $Lcmp_A$. Dually, $Lcmp_{\mathcal{J}([A \to_m B])}$ depends on $Ucmp_A$. These definitions are dealt with in Subsection 8.4.2. The case in Figure 8.6 which is most in need of explanation is that for functions.

Lemma 8.4.3 *Let A and B be finite distributive lattices. Let $a \in A$ and let $m \in \mathcal{M}(B)$. Then $Ucmp_{\mathcal{M}([A \to_m B])}(\lfloor a, m\rfloor)$ is $Min(X \cup Y)$, where the sets X and Y are given by:*

$$\begin{array}{rcl} X & = & \{\lfloor a', m'\rfloor \mid a' \in Lcmp_A(a), m' \in \mathbf{F}^\wedge(\bot_B)\} \\ Y & = & \{\lfloor \top_A, m'\rfloor \mid m' \in Ucmp_{\mathcal{M}(B)}(m)\}. \end{array}$$

Proof (sketch) We use the fact that $Min(X \cup Y) = Min(Min(X) \cup Min(Y))$ (see Lemma A.1.2 in the Appendix). By definition, $Ucmp_{\mathcal{M}([A \to_m B])}(\lfloor a, m\rfloor)$ is the set of

$\mathcal{M}(\mathbf{2})$
$$Ucmp(0) = \emptyset$$
$\mathcal{M}(A_\perp)$
$$\begin{array}{lcl} Ucmp(\perp) & = & \{lift(m) \mid m \in \mathbf{F}^\wedge(\perp_A)\} \\ Ucmp(lift(m)) & = & \{lift(m') \mid m' \in Ucmp_{\mathcal{M}(A)}(m)\} \end{array}$$
$\mathcal{M}(A^\top)$
$$\begin{array}{lcl} Ucmp(colift(\top_A)) & = & \emptyset \\ Ucmp(colift(m)) & = & \left\{ \begin{array}{ll} colift(\top_A) & \text{if } X = \emptyset \\ \{colift(m') \mid m' \in X\} & \text{if } X \neq \emptyset \end{array} \right. \end{array}$$ where $m \neq \top_A$ and $X \equiv Ucmp_{\mathcal{M}(A)}(m)$
$\mathcal{M}(A_1 \times \cdots \times A_n)$
$$\begin{array}{l} Ucmp((\top_1, \ldots, \top_{i-1}, m, \top_{i+1}, \ldots, \top_n)) \\ \quad = \quad M_i(Ucmp_{\mathcal{M}(A_i)}(m)) \cup \bigcup_{1 \leq k \leq n, k \neq i} M_k(\mathbf{F}^\wedge(\perp_k)) \end{array}$$ where $\perp_k \equiv \perp_{A_k}$ and $\top_k \equiv \top_{A_k}$
$\mathcal{M}([A \to_m B])$
$$\begin{array}{lcl} Ucmp(\lfloor a, m \rfloor) & = & \{\lfloor a', m' \rfloor \mid a' \in Lcmp_A(a), m' \in \mathbf{F}^\wedge(\perp_B), m' \notin Ucmp_{\mathcal{M}(B)}(m)\} \\ & & \qquad \cup \\ & & \{\lfloor \top_A, m' \rfloor \mid m' \in Ucmp_{\mathcal{M}(B)}(m)\} \end{array}$$

Figure 8.6: Upper Complements in $\mathcal{M}(A)$

minimal meet irreducible elements not below $\lfloor a, m \rfloor$. By Lemma 8.2.2, we can divide the $\lfloor a', m' \rfloor$ such that $\lfloor a', m' \rfloor \not\sqsubseteq \lfloor a, m \rfloor$, into those for which $m' \not\sqsubseteq m$ and those for which $a \not\sqsubseteq a'$. Taking the minimal elements of these two sets then gives the sets Y and X respectively. □

This result justifies the function case in Figure 8.6 by observing that $\top \notin Lcmp(a)$ for any a, so the only way an element $\lfloor a', m' \rfloor$ in X can be comparable with an element $\lfloor \top, m'' \rfloor$ in Y, is if $m'' \sqsubseteq m'$, in which case $m' = m''$ since $m' \in \mathbf{F}^{\wedge}(\bot_B)$ is minimal in B.

8.4.2 Operations on A

Comparisons

Given the meet frontiers for a and a' we must be able to decide whether $a \sqsubseteq a'$. This is straightforward: by Birkhoff's Representation Theorem, $a \sqsubseteq a' \iff \uparrow\mathbf{F}^{\wedge}(a') \subseteq \uparrow\mathbf{F}^{\wedge}(a)$, and $\uparrow\mathbf{F}^{\wedge}(a') \subseteq \uparrow\mathbf{F}^{\wedge}(a) \iff \mathbf{F}^{\wedge}(a') \leq_u \mathbf{F}^{\wedge}(a)$. The method for deciding $\mathbf{F}^{\wedge}(a') \leq_u \mathbf{F}^{\wedge}(a)$ is obvious from the definition of $\leq_u$ (Definition 7.2.4). Dually, we could use the join frontiers for a and a', since $a \sqsubseteq a' \iff \mathbf{F}^{\vee}(a) \leq_l \mathbf{F}^{\vee}(a')$.

Least Upper Bound and Greatest Lower Bound

Given the meet frontiers for a and a', the meet frontier for $a \sqcup a'$ is given by Lemma 8.2.4. The meet frontier for $a \sqcap a'$ is just $Min(\mathbf{F}^{\wedge}(a) \cup \mathbf{F}^{\wedge}(a'))$. Dually, the join frontier for $a \sqcup a'$ is $Max(\mathbf{F}^{\vee}(a) \cup \mathbf{F}^{\vee}(a'))$.

Upper and Lower Complements

For any $A \in \mathcal{A}$ and $a \in A$, given $\mathbf{F}^{\vee}(a)$ and $\mathbf{F}^{\wedge}(a)$ we need to be able to calculate $\mathbf{F}^{\vee}(a')$ and $\mathbf{F}^{\wedge}(a')$ for each $a' \in Ucmp_A(a)$ (and dually, for each $a' \in Lcmp_A(a)$). This turns out to be simple given the ability to calculate $Ucmp_{\mathcal{M}(A)}$ and $Lcmp_{\mathcal{J}(A)}$.

Lemma 8.4.4 *Let L be a finite distributive lattice and let $x \in L$. Then*

1. $Ucmp_L(x) = \{MtoJ(m) \mid m \in \mathbf{F}^{\wedge}(x)\}$;
2. $Lcmp_L(x) = \{JtoM(j) \mid j \in \mathbf{F}^{\vee}(x)\}$.

Proof We prove the first of these directly. The second is dual. By the definition of $Ucmp_L$ (7.3.4) and Proposition 8.1.9 we have that $Ucmp_L(m) = \{MtoJ(m)\}$ for any $m \in \mathcal{M}(L)$. Then

$$\begin{aligned}
Ucmp_L(x) &= Min\,\{x' \in L \mid x' \not\sqsubseteq x\} \\
&= Min\,\{x' \in L \mid x' \not\sqsubseteq \sqcap \mathbf{F}^{\wedge}(x)\} \\
&= Min\,\{x' \in L \mid \exists m \in \mathbf{F}^{\wedge}(x).\, x \not\sqsubseteq m\} && \text{by definition of meet} \\
&= Min\left(\textstyle\bigcup_{m \in \mathbf{F}^{\wedge}(x)} \{x' \in L \mid x \not\sqsubseteq m\}\right) \\
&= Min\left(\textstyle\bigcup_{m \in \mathbf{F}^{\wedge}(x)} Min\,\{x' \in L \mid x \not\sqsubseteq m\}\right) && \text{by Lemma A.1.2} \\
&= Min\left(\textstyle\bigcup_{m \in \mathbf{F}^{\wedge}(x)} Ucmp_L(m)\right) && \text{by definition of } Ucmp_L \\
&= Min\left(\textstyle\bigcup_{m \in \mathbf{F}^{\wedge}(x)} \{MtoJ(m)\}\right) \\
&= \{MtoJ(m) \mid m \in \mathbf{F}^{\wedge}(x)\},
\end{aligned}$$

where the last step follows by the fact the $MtoJ$ is an isomorphism from $\mathcal{M}(L)$ onto $\mathcal{J}(L)$ (hence an order embedding of $\mathcal{M}(L)$ into L), and thus preserves irredundancy. □

Lemma 8.4.5 *Let L be a finite distributive lattice, let $m \in \mathcal{M}(L)$ and let $j \in \mathcal{J}(L)$. Then*

1. $\mathbf{F}^{\wedge}(MtoJ(m)) = Ucmp_{\mathcal{M}(L)}(m)$;
2. $\mathbf{F}^{\vee}(JtoM(j)) = Lcmp_{\mathcal{J}(L)}(j)$.

Proof By definition, $\mathbf{F}^{\wedge}(MtoJ(m)) = Min\,\{m' \in \mathcal{M}(L) \mid MtoJ(m) \sqsubseteq m'\}$. But by Proposition 8.1.9, $MtoJ(m) \sqsubseteq m' \iff m' \not\sqsubseteq m$. Thus $\mathbf{F}^{\wedge}(MtoJ(m)) = Min\,\{m' \in \mathcal{M}(L) \mid m' \not\sqsubseteq m\}$, which is just the definition of $Ucmp_{\mathcal{M}(L)}(m)$. The second part is dual. □

Corollary 8.4.6 *Let L, m and j be given as in the statement of the Lemma. Then*

1. $\mathbf{F}^{\wedge}(j) = Ucmp_{\mathcal{M}(L)}(JtoM(j))$;
2. $\mathbf{F}^{\vee}(m) = Lcmp_{\mathcal{J}(L)}(MtoJ(m))$.

Let L be a finite distributive lattice and let $x \in L$. By Lemma 8.4.4, the elements of $Ucmp_L(x)$ are the join irreducible elements $MtoJ(m)$ for $m \in \mathbf{F}^{\wedge}(x)$. Clearly, for any join irreducible element j, $\mathbf{F}^{\vee}(j) = \{j\}$. Thus the join frontiers for the elements of the upper complement of x are:

$$\{\mathbf{F}^{\vee}(x') \mid x' \in Ucmp_L(x)\} = \{\{MtoJ(m)\} \mid m \in \mathbf{F}^{\wedge}(x)\}.$$

By Lemma 8.4.5, the meet frontier for a join irreducible element $MtoJ(m) \in \mathcal{J}(L)$ is just $Ucmp_{\mathcal{M}(L)}(m)$. Thus the meet frontiers for the elements of the upper complement of x are:

$$\{\mathbf{F}^{\wedge}(x') \mid x' \in Ucmp_L(x)\} = \{Ucmp_{\mathcal{M}(L)}(m) \mid m \in \mathbf{F}^{\wedge}(x)\}\,.$$

Dual arguments show that the meet frontiers for the elements of the lower complement of x are

$$\{\mathbf{F}^{\wedge}(x') \mid x' \in Lcmp_L(x)\} = \{\{JtoM(j)\} \mid j \in \mathbf{F}^{\vee}(x)\}\,,$$

and the join frontiers for the elements of the lower complement of x are

$$\{\mathbf{F}^{\vee}(x') \mid x' \in Lcmp_L(x)\} = \{Lcmp_{\mathcal{J}(L)}(j) \mid j \in \mathbf{F}^{\vee}(x)\}\,.$$

Remark It is the need to calculate upper and lower complements which forces us to use both the meet frontier and join frontier representations, rather than one or the other: the above discussion shows that to calculate the upper complement of x we need the meet frontier for x, while to calculate the lower complement of x we need the join frontier for x.

Chapter 9

Approximate Fixed Points

In this chapter we argue that despite the benefits gained from the use of frontiers it will often be necessary to reduce the size of the abstract domains before attempting to find the fixed point of a function. We provide a method of doing this without having to change the original abstract interpretation. In general this will entail settling for imprecise but safe approximations to the actual fixed point of a function.

9.1 A problem of complexity

For many of the functions which arise in abstract interpretation, establishing the graph of a function, using *any* method, will be intractable. To see why this is so, consider the function definition:

$$
\begin{array}{lcl}
\texttt{fold(f, nil, z)} & = & \texttt{z} \\
\texttt{fold(f, cons(x,y), z)} & = & \texttt{f x (fold(f, y, z))}
\end{array}
$$

We have used a pattern matching recursion equation style of function definition, but it is obvious that the same definition could be phrased in the language $\Lambda_{\mathcal{T}^{\overline{\mu}}}$ of Chapter 6. We will address the question of what type `fold` has in a moment. Suppose we wish to analyse programs using `fold` for strictness in the style of [BHA86], using Wadler's domains for lists ([Wad87]). For example, consider the definition of the catenate function

$$\texttt{cat} : \mathit{list}(\mathit{list}(\mathit{int})) \rightarrow \mathit{list}(\mathit{int})$$

$$\texttt{cat l} \;=\; \texttt{fold(append, l, nil)}$$

where `append` is assumed to have the usual definition. A little thought will show that `fold` must have the rather complex type:

$$\texttt{fold} : (\mathit{list}(\mathit{int}) \to \mathit{list}(\mathit{int}) \to \mathit{list}(\mathit{int})) \times \mathit{list}(\mathit{list}(\mathit{int})) \times \mathit{list}(\mathit{int}) \to \mathit{list}(\mathit{int}).$$

In the strictness analysis of [Bur91], the interpretation used for *int* is $\mathbf{2}$ and $\mathit{list}(\sigma)$ is interpreted as $(D^{\mathrm{B}}_{\sigma})$ lifted twice. This induces interpretations of $(\mathbf{2}_\perp)_\perp$ (written $\mathbf{4}$) for $\mathit{list}(\mathit{int})$ and $(\mathbf{4}_\perp)_\perp$ (written $\mathbf{6}$) for $\mathit{list}(\mathit{list}(\mathit{int}))$. The abstract function for `fold` would thus be an element of $[[\mathbf{4} \to [\mathbf{4} \to \mathbf{4}]] \times \mathbf{6} \times \mathbf{4} \to \mathbf{4}]$.

Even taking monotonicity into account, it is not hard to show that the argument domain alone for this function contains of the order of 10^6 elements. Clearly, if we have to evaluate even one of the approximations to `fold` at a significant proportion of these elements to establish its value, the operation of finding a fixed point will be too costly to be considered in any practical compiler.

Remark In most functional languages, `fold` would be given the polymorphic type `fold` : $\forall\alpha,\beta.\,(\alpha \to \beta \to \beta) \times \mathit{list}(\alpha) \times \beta \to \beta$, which raises the question of whether it is necessary to interpret `fold` at all the specific instances at which it is used. As we saw in Section 2.5, it is not always possible to avoid this if we want to get good results from our analysis but, in any case, it would obviously be possible to construct a monomorphic example of the same complexity.

The kinds of function which are 'well behaved' with respect to the frontiers algorithm are described in [JC87]: because the frontiers algorithm searches the argument lattice from the top and bottom, working towards the middle, well behaved functions are those which have frontiers whose elements are either very low down or very high up the argument lattice. For such functions the frontier sets are small and the frontiers algorithm will find them with little effort. On the other hand, badly behaved functions have frontier sets consisting of elements from the middle of the lattice and in the worst case the frontiers algorithm will evaluate the function at every point in the lattice before finding the frontier sets.

Experience with an implementation of a strictness analyser employing the frontiers algorithm suggests that higher-order functions are often badly behaved (as is certainly the case for `fold`). One reason for this is that higher-order functions tend to apply some of their arguments to others and thus behave as more or less exotic variants of the *apply* function. The problem with *apply* is that, for example, $\mathit{apply}(f, d)$ will be high up in the result lattice if *either* f *or* d are high in their respective lattices: this implies that the minimal and maximal elements of the inverse image of any given result value will themselves be neither high nor low in the argument lattice as a whole.

9.1.1 Reducing the Size of a Lattice

Our solution to the complexity problem described above, is to work in a smaller lattice to establish bounds on the required fixed point. We use maps, reminiscent of the α and γ maps used in abstract interpretation, to move between larger and smaller lattices. First we must formalise the notion of one lattice being smaller than another.

Definition 9.1.1 *The* abstraction ordering $\preceq$ *is defined on* $\mathcal{A}$ *as follows:*

$$\begin{array}{lcll} \mathbf{2} & \preceq & B & \textit{for all } B \in \mathcal{A} \\ A_1 \times \ldots \times A_n & \preceq & B_1 \times \ldots \times B_n & \textit{if } A_i \preceq B_i, 1 \leq i \leq n \\ A_\perp & \preceq & B_\perp & \textit{if } A \preceq B \\ A^\top & \preceq & B^\top & \textit{if } A \preceq B \\ [A_1 \to A_2] & \preceq & [B_1 \to B_2] & \textit{if } A_1 \preceq B_1 \textit{ and } A_2 \preceq B_2. \end{array}$$

If $A \preceq B$*, we say that* A is an abstraction of B.

Note that the definition of $\preceq$ in the function case is *not* contravariant. As we shall see, this is because our notion of when one lattice is smaller than another is formulated such that when $A \preceq B$ there are Galois connections embedding A into B. This avoids contravariance in the same way that the use of embedding-projection pairs avoids it in the solution of recursive domain equations ([SP82], see also Chapter 6).

We next define two families of Galois connections relating elements of members of $\mathcal{A}$: the 'safe' maps, which give overestimates of values, thus allowing us to derive upper bounds on fixed points, and their 'live' counterparts, which give underestimates and allow us to derive lower bounds.

Definition 9.1.2 *Let $A, B \in \mathcal{A}$ with $B \preceq A$. The* safe *maps $Abs^s_{A,B} : A \to B$ and $Conc^s_{B,A} : B \to A$, are defined by:*

$$
\begin{array}{lcl}
Abs^s_{A,\mathbf{2}}\ a & = & \begin{cases} 0 & \textit{if } a = \bot_A \\ 1 & \textit{otherwise} \end{cases} \\
Conc^s_{\mathbf{2},A}\ b & = & \begin{cases} \bot_A & \textit{if } b = 0 \\ \top_A & \textit{if } b = 1 \end{cases}
\end{array}
$$

$$
\begin{array}{lcl}
Abs^s_{A_\bot,B_\bot}\ x & = & \begin{cases} \bot & \textit{if } x = \bot \\ \mathit{lift}(Abs^s_{A,B}\ a) & \textit{if } x = \mathit{lift}(a) \end{cases} \\
Conc^s_{B_\bot,A_\bot}\ x & = & \begin{cases} \bot & \textit{if } x = \bot \\ \mathit{lift}(Conc^s_{B,A}\ b) & \textit{if } x = \mathit{lift}(b) \end{cases}
\end{array}
$$

$$
\begin{array}{lcl}
Abs^s_{A^\top,B^\top}\ x & = & \begin{cases} \top & \textit{if } x = \top \\ \mathit{colift}(Abs^s_{A,B}\ a) & \textit{if } x = \mathit{colift}(a) \end{cases} \\
Conc^s_{B^\top,A^\top}\ x & = & \begin{cases} \top & \textit{if } x = \top \\ \mathit{colift}(Conc^s_{B,A}\ b) & \textit{if } x = \mathit{colift}(b) \end{cases}
\end{array}
$$

$$
\begin{array}{lcl}
Abs^s_{[A_1 \to A_2],[B_1 \to B_2]}\ f & = & Abs^s_{A_2,B_2} \circ f \circ Conc^s_{B_1,A_1} \\
Conc^s_{[B_1 \to B_2],[A_1 \to A_2]}\ f & = & Conc^s_{B_2,A_2} \circ f \circ Abs^s_{A_1,B_1}
\end{array}
$$

For $A = A_1 \times \ldots \times A_n$ and $B = B_1 \times \ldots \times B_n$:

$$
\begin{array}{lcl}
Abs^s_{A,B}(a_1, \ldots, a_n) & = & (Abs^s_{A_1,B_1}\ a_1, \ldots, Abs^s_{A_n,B_n}\ a_n) \\
Conc^s_{B,A}(b_1, \ldots, b_n) & = & (Conc^s_{B_1,A_1}\ b_1, \ldots, Conc^s_{B_n,A_n}\ b_n)
\end{array}
$$

Definition 9.1.3 *The definitions of the* live *maps are given by substituting Abs^l for Abs^s and $Conc^l$ for $Conc^s$ everywhere in definition 9.1.2, except for the base case for Abs^l, which is:*

$$
Abs^l_{A,\mathbf{2}}\ a \ = \ \begin{cases} 1 & \textit{if } a = \top_A \\ 0 & \textit{otherwise.} \end{cases}
$$

The following lemma shows that the Abs and $Conc$ pairs form Galois connections, and indeed that they satisfy the stronger property termed *exact adjointness* in [Myc81] (perhaps a better known terminology is that which identifies the $Conc^l$–Abs^l pairs as *embedding-projection* pairs and the $Conc^s$–Abs^s pairs as *embedding-closure* pairs).

Lemma 9.1.4 : *For all $A, B \in \mathcal{A}$, such that $B \preceq A$*

1. $Abs^l_{A,B} \circ Conc^l_{B,A} = id_B = Abs^s_{A,B} \circ Conc^s_{B,A}$
2. $Conc^l_{B,A} \circ Abs^l_{A,B} \sqsubseteq id_A \sqsubseteq Conc^s_{B,A} \circ Abs^s_{A,B}$

Proof Straightforward induction on B. □

Corollary 9.1.5 *For all $A, B \in \mathcal{A}$ such that $B \preceq A$:*

1. *$Conc^s_{B,A}$ and $Conc^l_{B,A}$ are injective;*
2. *$Abs^s_{A,B}$ and $Abs^l_{A,B}$ are onto;*
3. *$Abs^s_{A,B}$ and $Abs^l_{A,B}$ are strict.*

For each $A \in \mathcal{A}$, let fix_A be the fixed point operator on $[A \to A]$. Somewhat surprisingly, both the safe and live Abs maps 'preserve' fix.

Lemma 9.1.6 *For all lattices $A, B \in \mathcal{A}$ such that $B \preceq A$:*

1. $\mathit{fix}_B = Abs^s_{[[A \to A] \to A],[[B \to B] \to B]}(\mathit{fix}_A)$;
2. $\mathit{fix}_B = Abs^l_{[[A \to A] \to A],[[B \to B] \to B]}(\mathit{fix}_A)$.

Proof

1. Let $A' = [A \to A]$ and $B' = [B \to B]$. A routine induction on i suffices to show that for all i, for all $f \in B'$:

$$Abs^s_{A,B}((Conc^s_{B',A'}\ f)^i\ \bot_A) = f^i\ \bot_B.$$

Then, for any $f \in [B \to B]$,

$$\begin{aligned}
& (Abs^s_{[[A \to A] \to A],[[B \to B] \to B]}\ \mathit{fix}_A)\ f \\
= \;& Abs^s_{A,B}(\mathit{fix}_A(Conc^s_{B',A'}\ f)) \\
= \;& Abs^s_{A,B}\left(\bigsqcup_{i=0}^{\infty}(Conc^s_{B',A'}\ f)^i \bot_A\right) \\
= \;& \bigsqcup_{i=0}^{\infty} Abs^s_{A,B}((Conc^s_{B',A'}\ f)^i\ \bot_A) \qquad \text{since } Abs^s_{A,B} \text{ monotone, } A \text{ finite} \\
= \;& \bigsqcup_{i=0}^{\infty} f^i\ \bot_B \\
= \;& \mathit{fix}_B\ f.
\end{aligned}$$

2. The proof for part 1 goes through identically substituting Abs^l for Abs^s and $Conc^l$ for $Conc^s$.

□

The next result uses this lemma to show that we can construct safe (upper) and live (lower) approximations to the fixed point of a function over some member of $\mathcal{A}$, by abstracting the function to a smaller member of $\mathcal{A}$ and finding the fixed point there.

Theorem 9.1.7 *For all lattices $A, B \in \mathcal{A}$ such that $B \preceq A$, for all $f \in [A \to A]$:*

1. $Conc^s_{B,A}(\mathit{fix}_B(Abs^s_{[A\to A],[B\to B]}f)) \sqsupseteq \mathit{fix}_A\ f$;

2. $Conc^l_{B,A}(\mathit{fix}_B(Abs^l_{[A\to A],[B\to B]}f)) \sqsubseteq \mathit{fix}_A\ f$.

Proof

$$
\begin{array}{rlr}
1. & Conc^s_{B,A} \circ \mathit{fix}_B \circ Abs^s_{[A\to A],[B\to B]} & \\
= & Conc^s_{B,A} \circ (Abs^s_{[[A\to A]\to A],[[B\to B]\to B]}\mathit{fix}_A) \circ Abs^s_{[A\to A],[B\to B]} & \text{Lemma 9.1.6} \\
= & Conc^s_{B,A} \circ Abs^s_{A,B} \circ \mathit{fix}_A \circ Conc^s_{[B\to B],[A\to A]} \circ Abs^s_{[A\to A],[B\to B]} & \text{Def. 9.1.2} \\
\sqsupseteq & \mathit{fix}_A & \text{Lemma 9.1.4} \\
2. & Conc^l_{B,A} \circ \mathit{fix}_B \circ Abs^l_{[A\to A],[B\to B]} & \\
= & Conc^l_{B,A} \circ (Abs^l_{[[A\to A]\to A],[[B\to B]\to B]}\mathit{fix}_A) \circ Abs^l_{[A\to A],[B\to B]} & \text{Lemma 9.1.6} \\
= & Conc^l_{B,A} \circ Abs^l_{A,B} \circ \mathit{fix}_A \circ Conc^l_{[B\to B],[A\to A]} \circ Abs^l_{[A\to A],[B\to B]} & \text{Def. 9.1.2} \\
\sqsubseteq & \mathit{fix}_A & \text{Lemma 9.1.4}
\end{array}
$$

□

9.1.2 Applying *Conc* to a Frontier

Given a function $G \in [A \to A]$, we obtain upper and lower bounds on the value of $\mathit{fix}_A\ G$ by evaluating

$$Conc^s_{B,A}(\mathit{fix}_B(Abs^s_{A,B} \circ G \circ Conc^s_{B,A}))$$

and

$$Conc^l_{B,A}(\mathit{fix}_B(Abs^l_{A,B} \circ G \circ Conc^l_{B,A}))$$

If we are using the frontier representations described in the previous two chapters we must be able to implement Abs and $Conc$ on frontiers. The non-function cases are straightforward. In the function case, we have so far only worked out the details (Lemma 9.1.8 below) for coping with the restricted case of Chapter 7, i.e., for functions in spaces of the form $[A \to \mathbf{2}]$. In the future we hope to extend the applicability of this method.

For a function $f \in [B \to \mathbf{2}]$, we can construct the maximum–0–frontier for

$$Conc^s_{[B\to\mathbf{2}],[A\to\mathbf{2}]}\ f$$

by using the following result.

Lemma 9.1.8 *For $A, B \in \mathcal{A}$ such that $B \preceq A$, for $f \in [B \to \mathbf{2}]$,*

$$(f \circ Abs^s_{A,B})^{-1}\{0\} = \downarrow\left\{Conc^s_{B,A}\ b \mid b \in f^{-1}\{0\}\right\}$$

Proof To show the inclusion from left to right assume $y \in (f \circ Abs^s_{A,B})^{-1}\{0\}$, i.e., $y \in A$ and $f(Abs^s_{A,B}\ y) = 0$. Then $(Conc^s_{B,A}(Abs^s_{A,B}\ y)) \in \left\{Conc^s_{B,A}\ b \mid b \in f^{-1}\{0\}\right\}$. By lemma 9.1.4, $y \sqsubseteq Conc^s_{B,A}(Abs^s_{A,B}\ y)$. Hence $y \in \downarrow\left\{Conc^s_{B,A}\ b \mid b \in f^{-1}\{0\}\right\}$. For the inclusion from right to left assume $y \in \downarrow\left\{Conc^s_{B,A}\ b \mid b \in f^{-1}\{0\}\right\}$. Then $y \sqsubseteq Conc^s_{B,A}\ b$, for some $b \in f^{-1}\{0\}$. Thus, $f(Abs^s_{A,B}\ y) \sqsubseteq f(Abs^s_{A,B}(Conc^s_{B,A}\ b))$ $= f\ b$ (by lemma 9.1.4). Hence $y \in (f \circ Abs^s_{A,B})^{-1}\{0\}$, since $f\ b = 0$. □

From this it is easy to show that

$$\mathbf{F\text{-}0}(Conc^s_{[B\to\mathbf{2}],[A\to\mathbf{2}]}\ f) = \left\{Conc^s_{B,A}\ b \mid b \in \mathbf{F\text{-}0}f\right\}$$

since the *Conc* maps are injective. We also need to calculate the minimum–1–frontier for $Conc^s_{[B\to\mathbf{2}],[A\to\mathbf{2}]}\ f$. This can be done using the following result, which allows us to convert a meet frontier into a join frontier.

Lemma 9.1.9 *Let L be a finite distributive lattice and let $x \in L$. Then*

$$\mathbf{F}^{\vee}(x) = \bigcap_{\mathcal{J}(L)}^{max} \{Lcmp_{\mathcal{J}(L)}(MtoJ(m)) \mid m \in \mathbf{F}^{\wedge}(x)\}.$$

Proof By Corollary 8.4.6, we know that $Lcmp_{\mathcal{J}(L)}(MtoJ(m)) = \mathbf{F}^{\vee}(m)$. Then by Proposition 8.2.4,

$$\bigcap_{\mathcal{J}(L)}^{max} \{Lcmp_{\mathcal{J}(L)}(MtoJ(m)) \mid m \in \mathbf{F}^{\wedge}(x)\}$$

is the join frontier for $\bigsqcap\{m \mid m \in \mathbf{F}^{\wedge}(x)\}$, and this is just x. □

Dual results hold for the live *Conc* maps.

9.1.3 Using the Upper and Lower Bounds

How can we make use of the ability to place upper and lower bounds on the value of a function? Here we outline a possible approach to using *Abs* and *Conc* in strictness analysis.

Suppose that $f^{\mathrm{s}} \in D^{\mathrm{s}}_\sigma$ is the standard interpretation of the term $[\![\mathbf{Y}_\sigma\ e]\!]$, where $e : \sigma \to \sigma$ is a closed term with abstract interpretation $F \in [A \to A]$, with $A \in \mathcal{A}$. We wish to evaluate $f^{\mathbf{B}} = \bigsqcup_{i=0}^{\infty} F^i(\perp_A)$, but the lattice A may be too large for this to be practical. In this case we choose a smaller lattice, $B \in \mathcal{A}$.

First we calculate the fixed point of $Abs^l_{A,B} \circ F \circ Conc^l_{B,A}$, to which we apply $Conc^l_{B,A}$, giving us a lower bound on $f^{\mathbf{B}}$. Call this f_{lb}. Using f_{lb} we can place an upper limit on the degree of strictness which our abstract interpretation would determine if time and space allowed. If we find that even f_{lb} does not imply that f^{s} is strict in ways in which we are interested, there is no need to proceed any further. On the other hand, if the lower bound shows that f^{s} *may* be strict in such ways, we can go on to calculate an upper bound, say f_{ub}, using $Abs^s_{A,B}$ and $Conc^s_{B,A}$. If f_{ub} confirms that f^{s} is strict our job is done.

In the remaining case, f_{lb} and f_{ub} 'disagree' concerning the strictness of f^{s}. We must then decide whether to cut our losses and accept that we are unable to confirm that f^{s} is strict, or try to calculate improved upper and lower bounds by repeating the process using a new lattice B', with $B \prec B' \preceq A$. In choosing B' we would have to be sure that we did not run into the complexity problems we were trying to avoid in the first place. This would not be entirely dependent on the absolute size of B', since $Abs^l_{A,B'}(f_{lb})$ and $Abs^s_{A,B'}(f_{ub})$ place lower and upper bounds on the values of $\mathit{fix}_{B'}(Abs^l_{[A\to A],[B'\to B']}F)$ and $\mathit{fix}_{B'}(Abs^s_{[A\to A],[B'\to B']}F)$. The lower bound allows us to start the fixed point iterations at a point above $\perp_{B'}$ and the frontiers algorithm can use both upper and lower bounds as a means of reducing the search space when establishing the frontier of each approximation. One possibility this raises is that the work done in the smaller domains might achieve in a few 'big steps' what would take many 'little steps' in the original domain.

Further experimentation is needed to determine whether the use of *Abs* and *Conc* to render an abstract interpretation tractable should be an iterative process of refinement, as is suggested above, or whether we should choose a reasonably sized domain and stick with it.

Chapter 10

Conclusions

We conclude by summarising the achievements of this thesis and considering directions for further work.

10.1 Summary

Our contributions to the area of abstract interpretation for higher-order functional languages can be roughly divided into those which concern the theory of abstract interpretation and those which concern its practice. We consider these in turn.

Theory

We have presented a new semantic framework for the abstract interpretation of higher-order functional languages. Its advantage over existing frameworks is that it captures a wider range of program properties and thus allows new analyses to be developed and proved correct for higher-order languages.

We saw that the existing frameworks of [BHA86] and [Abr90] were restricted by identifying properties with sets. In particular we saw that this did not allow these frameworks to capture properties which could be described in terms of projections, such as those used in the first-order analyses of [WH87] and [Lau89].

We introduced the idea of using certain equivalence relations, the kernels of projections, as an alternative to using the projections themselves and showed the two approaches to be equivalent. We then introduced partial equivalence relations (pers), and in particular the complete pers, as the natural generalisation of equivalence relations in the setting of domains and higher-order functions.

By moving from the binary logical relations used in [Abr90], to ternary logical relations, we were able to develop a framework for abstract interpretation in which properties are identified with complete pers rather than sets. Of key importance in the development of this framework was the Logical Concretisation Map Theorem: this allowed us to move freely between the relational and concretisation map formulations of correctness.

It was shown that preservation of meets by logical concretisation maps is an inherited property, and that preservation of meets implies the existence of best interpretations for constants. We gave an inductive definition of a family of abstraction maps which map the standard interpretation of a constant to its best abstract interpretation, but we showed that non-injective concretisation maps could prevent the natural derivation of best interpretations.

We showed how our basic language could be enriched, and the abstract interpretation framework extended accordingly, both by adding list types and more generally by adding recursive types. We presented an analysis able to detect head-strictness in higher-order functional programs. We adapted [Lau89] to construct recursive descriptions of pers as properties for recursive types, and we used the framework of [SP82] to give meaning to such recursive descriptions in an appropriate category. We were able to induce abstract interpretations for the enriched language, but only by restricting the use of $\rightarrow$ in recursive types. We showed that the concretisation maps of the induced interpretations need not preserve meets and hence that best interpretations for constants need not exist.

Practice

We have described a method, based on Birkhoff's Representation Theorem for finite distributive lattices, for representing the graphs of functions, and a method for finding approximate fixed points. Both methods are suitable for use in the implementation of abstract interpretations.

We began by explaining the need to explicitly calculate the graphs of recursively defined functions in abstract interpretation. We showed how the frontier representations for functions in $[P \rightarrow \mathbf{2}]$, due to [JC87], can be understood in terms of upper and lower subsets of P. Based on this understanding we were able to develop a simple algorithm for constructing frontier representations.

We then introduced a strong generalisation of frontiers, our key insight being that the frontier representations of [JC87] are special cases of Birkhoff's Representation Theorem for finite distributive lattices. We developed an algorithm for exploiting this generalisation, explaining in some detail how the operations of the algorithm can be implemented for a particular family of finite distributive lattices.

Finally we argued that the lattices in abstract interpretations for higher-order types, though finite, can be so large that approximation techniques must be employed when finding fixed points. We described such a technique based on the use of Galois connections.

10.2 Suggestions for Further Work

In this section we consider some of the shortcomings of the material presented in this thesis and suggest ways in which it could be developed.

Best Interpretations and Non-Injective Concretisation Maps

The discussion in Chapter 4 about best interpretations and expected forms suggested that it may be possible to construct a maximally accurate analysis without the interpretations of the constants necessarily being best. The main problem with this discussion is the vagueness of some of the central ideas, particularly that of 'expected form'. It would be interesting to pursue this topic and to try and give the ideas precise form, particularly the analogy with full abstraction.

A related but rather more clearly defined issue is that of non-injective concretisation maps. Apart from causing problems with the derivation of best interpretations, there is a more fundamental objection to concretisation not being injective: it implies that the abstract lattices contain too many points. As long as we are forced to compute the full graphs of functions, this is more than a theoretical irritation, since it entails an obvious inefficiency.

In the case of constancy analysis, we suggested that it might be possible to obtain injective concretisation maps by using Berry's theory of bi-domains to construct a cartesian closed category (CCC) of finite lattices, whose exponents would give the 'right' interpretation of function types. This suggestion highlights the fact that in some respects our definition of interpretation may be too concrete. We could have further parameterised interpretations by the CCC in which terms were to be given meaning, with types, environments and lambda abstraction being interpreted in the CCC structure in the standard way ([LS86]). To adapt the abstract interpretation framework to this more general notion of interpretation we would need an appropriately abstract definition of logical relation (although it is probably not wise to get *too* abstract, since for the purposes of implementation we obviously need concrete representations of the objects and morphisms).

Recursive Types

There are two main problems with our treatment of recursive types. The first is that we were unable to construct finite lattice interpretations for recursive types in the general case (unrestricted use of $\to$). We saw that for the type $\mu\alpha\,.\,\mathit{int} + (\alpha \to \alpha)$, in order to obtain a monotone concretisation map, the three induced descriptions of properties would have to be ordered:

$$\begin{tikzcd} & c \arrow[dl, dash] \arrow[dr, dash] & \\ a & & b \end{tikzcd}$$

We might try to deal with this by formally completing the poset, i.e., by adding a new point $a \wedge b$ and setting $\gamma^J(a \wedge b) = (\gamma^J\ a) \wedge (\gamma^J\ b)$. This raises the question of whether it is in general *decidable* what the correct ordering on $\mathcal{P}_{\sigma^\alpha}$ should be. As yet we have no answer to this.

The second problem with our treatment of recursive types is that it does not generalise the head-strictness analysis of Chapter 5. The most promising approach to solving this problem may be to return to the way projections are used to describe strictness and use the idea of lifting as in [WH87], although [HL90] indicates that a forwards analysis using this approach may be rather weak.

Polymorphism

In Chapter 2 we discussed the fact that our chosen language is only simply typed and that this is a serious drawback, since polymorphic type systems are such an important feature of realistic functional languages. This is clearly a prime candidate for further work. There are two approaches we would like to take here. The first is to attempt to derive polymorphic invariance results for the properties which can be described using pers, in particular it seems intuitively clear that constancy should be a polymorphic invariant. The second approach is to try and adapt the work of [HL91]. This work exploits the fact that since projections are just special kinds of functions, it makes sense to define polymorphic projections: the challenge is to make it make sense for pers. In both approaches we hope to exploit the fact that pers can also be used to provide models of polymorphic lambda calculi (e.g., [AP90]).

Minimal Function Graphs

The generalised frontiers representation and algorithm we have described allow us to construct graphs of functions in an efficient way. But the fact remains that there is a huge inherent inefficiency involved in constructing complete graphs when all we need is a small part of the graph. What we would like to do is calculate the minimum part of a function's graph which is sufficient to answer the questions we want to ask. Because functions may be recursive, this will involve calculating parts of the graph which are not of direct interest but are needed to evaluate recursive calls. The problem in higher-order abstract interpretation is to decide when we have constructed a 'self-contained' subset of the graph, known as a minimal function graph ([JM86]), without having to compute the *full* graphs of functional arguments. In the first-order case this is not a problem.

The potential benefits of developing a minimal function graph method which works properly in the higher-order case are great, making this a priority area for research.

Appendix

Proofs from Chapters 7 and 8

A.1 Proofs from Chapter 7

We first need to establish some basic results about minimal and maximal elements for finite sets. We begin by restating and proving Lemma 7.2.2:

Lemma 7.2.2 *Let P be a poset and let $X \subseteq P$ be finite. Then:*

1. $\downarrow Max(X) = \downarrow X$;

2. $\uparrow Min(X) = \uparrow X$.

Proof We prove 2 directly, 1 is dual.

Assume $x \in \uparrow Min(X)$. Then $x \in \uparrow X$ since $Min(X) \subseteq X$. Now assume $x \in \uparrow X$ and let $X' \subseteq X$ be the set $\{x' \in X \mid x' \sqsubseteq x\}$. Suppose towards a contradiction that $x \notin \uparrow Min(X)$. Then for all $x' \in X'$, we must have $x' \notin Min(X)$. But this clearly requires X' to contain an infinite strictly decreasing chain, which contradicts X being finite. $\square$

Lemma A.1.1 *Let $X \subseteq P$ with P a poset. Then:*

1. $Min(\uparrow X) = Min(X)$;

2. $Max(\downarrow X) = Max(X)$.

Proof Obvious. $\square$

Lemma A.1.2 *Let P be a finite poset. Let $\{X_i\}_{i \in I}$ be a family of subsets of P. Then:*

1. $Min\left(\bigcup_{i \in I} X_i\right) = Min\left(\bigcup_{i \in I} Min(X_i)\right)$;

2. $Max\left(\bigcup_{i \in I} X_i\right) = Max\left(\bigcup_{i \in I} Max(X_i)\right)$.

Proof We prove 1 directly, 2 is dual.

$$
\begin{array}{lll}
\multicolumn{2}{l}{Min\left(\bigcup_{i \in I} X_i\right)} & \\
= & Min\left(\uparrow \bigcup_{i \in I} X_i\right) & \text{Lemma A.1.1} \\
= & Min\left(\bigcup_{i \in I} \uparrow X_i\right) & \\
= & Min\left(\bigcup_{i \in I} \uparrow Min(X_i)\right) & \text{Lemma 7.2.2} \\
= & Min\left(\uparrow \bigcup_{i \in I} Min(X_i)\right) & \\
= & Min\left(\bigcup_{i \in I} Min(X_i)\right) & \text{Lemma A.1.1.}
\end{array}
$$

□

Given these facts we can restate and prove Proposition 7.3.2.

Proposition 7.3.2 *Let P be a poset. Let $S_1, S_2 \in \mathcal{U}(P)$ and let $T_1, T_2 \in \mathcal{L}(P)$. Then*

1. $Min(S_1 \cap S_2) = Min\left(\bigcup \{x_1 \tilde{\vee}\, x_2 \mid x_1 \in Min(S_1), x_2 \in Min(S_2)\}\right)$;

2. $Max(T_1 \cap T_2) = Max\left(\bigcup \{y_1 \tilde{\wedge}\, y_2 \mid y_1 \in Max(T_1), y_2 \in Max(T_2)\}\right)$.

Proof We prove 1 directly, 2 is dual.

Let $M_1 = Min(S_1)$ and let $M_2 = Min(S_2)$. Since S_1 and S_2 are upper, we have $S_1 = \uparrow S_1$ and $S_2 = \uparrow S_2$. Thus

$$
\begin{array}{lll}
\multicolumn{2}{l}{Min(S_1 \cap S_2)} & \\
= & Min(\uparrow S_1 \cap \uparrow S_2) & \\
= & Min(\uparrow Min(S_1) \cap \uparrow Min(S_2)) & \text{Lemma 7.2.2} \\
= & Min\left(\bigcup_{x_1 \in M_1} \bigcup_{x_2 \in M_2} \{y \in P \mid y \sqsupseteq x_1 \;\; and \;\; y \sqsupseteq x_2\}\right) & \\
= & Min\left(\bigcup_{x_1 \in M_1} \bigcup_{x_2 \in M_2} Min\, \{y \in P \mid y \sqsupseteq x_1 \;\; and \;\; y \sqsupseteq x_2\}\right) & \text{Lemma A.1.2} \\
= & Min\left(\bigcup_{x_1 \in M_1} \bigcup_{x_2 \in M_2} x_1 \tilde{\vee}\, x_2\right) & \text{definition of } \tilde{\vee}.
\end{array}
$$

□

A.2 Proofs from Chapter 8

Recall from Chapter 8 that for a finite distributive lattice L, the maps $JtoM : \mathcal{J}(L) \to \mathcal{M}(L)$ and $MtoJ : \mathcal{M}(L) \to \mathcal{J}(L)$ are defined by

- $JtoM(j) = \bigsqcup(L \setminus \uparrow j)$
- $MtoJ(m) = \bigsqcap(L \setminus \downarrow m)$

with $\uparrow j$ and $\downarrow m$ being calculated in L. As promised, we will show (Theorem A.2.2) that these maps establish an isomorphism between $\mathcal{J}(L)$ and $\mathcal{M}(L)$. First we must show that they are well defined. This is a straightforward corollary to Proposition 8.1.9, which we now restate and prove.

Proposition 8.1.9 *Let L be a finite distributive lattice, let $j \in \mathcal{J}(L)$ and let $m \in \mathcal{M}(L)$. Then:*

1. $L \setminus \uparrow j = \downarrow \bigsqcup(L \setminus \uparrow j)$;
2. $L \setminus \downarrow m = \uparrow \bigsqcap(L \setminus \downarrow m)$,

where each $\uparrow$ and $\downarrow$ is calculated in L.

Proof We prove the first directly. The second is dual. Let $x \in L$. We must show that $x \sqsubseteq \bigsqcup(L \setminus \uparrow j) \iff j \not\sqsubseteq x$. Assume $x \sqsubseteq \bigsqcup(L \setminus \uparrow j)$ and suppose towards a contradiction that $j \sqsubseteq x$. We have $j \sqsubseteq x \sqsubseteq \bigsqcup(L \setminus \uparrow j)$, but by Lemma 8.1.5 this implies that $j \sqsubseteq x'$ for some $x' \in L \setminus \uparrow j$, a contradiction. Now assume $j \not\sqsubseteq x$. Then $x \in L \setminus \uparrow j$, hence $x \sqsubseteq \bigsqcup(L \setminus \uparrow j)$. $\square$

Corollary A.2.1 *Let L be a finite distributive lattice. Let $j \in \mathcal{J}(L)$ and let $m \in \mathcal{M}(L)$. Then $\bigsqcup(L \setminus \uparrow j) \in \mathcal{M}(L)$. Dually, $\bigsqcap(L \setminus \downarrow m) \in \mathcal{J}(L)$.*

Proof By Lemma 8.1.5 we must show that $\bigsqcap X \sqsubseteq \bigsqcup(L \setminus \uparrow j) \iff \exists x \in X.\, x \sqsubseteq \bigsqcup(L \setminus \uparrow j)$, for all non-empty $X \subseteq L$. Let X be such a subset. Then

$$
\begin{array}{lll}
\multicolumn{2}{l}{\bigsqcap X \sqsubseteq \bigsqcup(L \setminus \uparrow j)} & \\
\iff & j \not\sqsubseteq \bigsqcap X & \text{by the Proposition} \\
\iff & \exists x \in X.\, j \not\sqsubseteq x & \text{definition of meet} \\
\iff & \exists x \in X.\, x \sqsubseteq \bigsqcup(L \setminus \uparrow j) & \text{by the Proposition.}
\end{array}
$$

$\square$

Thus the maps $JtoM$ and $MtoJ$ are well defined.

Theorem A.2.2 *Let L be a finite distributive lattice. Then $JtoM$ and $MtoJ$ establish an isomorphism between the posets $\mathcal{J}(L)$ and $\mathcal{M}(L)$.*

Proof It is easy to see that both maps are monotone (e.g., $j \sqsubseteq j' \Rightarrow L \setminus \uparrow j \subseteq L \setminus \uparrow j' \Rightarrow \bigsqcup L \setminus \uparrow j \sqsubseteq \bigsqcup L \setminus \uparrow j'$). Then it suffices to show, for all $x \in L$ and $m \in \mathcal{M}(L)$, that $x \sqsubseteq JtoM(MtoJ(m)) \iff x \sqsubseteq m$ (the case for the converse composition follows by duality). Let $x \in L$ and let $m \in \mathcal{M}(L)$. Then:

$$\begin{array}{lll} x \sqsubseteq JtoM(MtoJ(m)) & & \\ \iff & MtoJ(m) \not\sqsubseteq x & \text{by Prop. 8.1.9 (1)} \\ \iff & x \sqsubseteq m & \text{by contraposition of Prop. 8.1.9 (2)} \end{array}$$

□

We now return to the proof of Theorem 8.2.1. First we need some subsidiary lemmas.

Lemma A.2.3 *Let P be a poset and let L be a finite distributive lattice. Let $f \in [P \to_m L]$. Then:*

1. $f = \bigsqcup \{ \lceil x, j \rceil \mid x \in P, j \in \mathcal{J}(L), \lceil x, j \rceil \sqsubseteq f \}$;
2. $f = \bigsqcap \{ \lfloor x, j \rfloor \mid x \in P, j \in \mathcal{M}(L), \lfloor x, j \rfloor \sqsupseteq f \}$.

Proof We prove 1 directly, 2 is dual.

Let G be the set $\{ \lceil x, j \rceil \mid x \in P, j \in \mathcal{J}(L), \lceil x, j \rceil \sqsubseteq f \}$. Note that $\lceil x, j \rceil \sqsubseteq f \iff j \sqsubseteq f(x)$, hence

$$G = \{ \lceil x, j \rceil \mid x \in P, j \in \mathcal{J}(L), j \sqsubseteq f(x) \}.$$

We must show that $f(x') = (\bigsqcup G)(x')$ for all $x' \in P$. Let $x' \in P$. Then $(\bigsqcup G)(x') = \bigsqcup \{ g(x') \mid g \in G \}$. Let S be the set $\{ g(x') \mid g \in G \}$, i.e.,

$$S = \{ \lceil x, j \rceil (x') \mid x \in P, j \in \mathcal{J}(L), j \sqsubseteq f(x) \}.$$

Now let J be given by

$$J = \{ j \in \mathcal{J}(L) \mid j \sqsubseteq f(x') \}.$$

Certainly $J \subseteq S$, since $j = \lceil x', j \rceil (x')$ for any j, and $j \sqsubseteq f(x') \Rightarrow \lceil x', j \rceil (x') \in S$. Now suppose for some $x \in P$ and $j \in \mathcal{J}(L)$ that $j \sqsubseteq f(x)$ but $\lceil x, j \rceil (x') \notin J$. Then either $\lceil x, j \rceil (x') \notin \mathcal{J}(L)$ or $\lceil x, j \rceil (x') \not\sqsubseteq f(x')$. In the first case we must have

$\lceil x,j \rceil (x') = \bot$, since $j \in \mathcal{J}(L)$, and the second case is impossible, since it requires that $x \sqsubseteq x'$ and $j \sqsubseteq f(x)$ and $j \not\sqsubseteq f(x')$. Thus either $S = J$ or $S = J \cup \{\bot\}$. In both cases $\bigsqcup S = \bigsqcup J$, so $(\bigsqcup G)(x') = \bigsqcup\{j \in \mathcal{J}(L) \mid j \sqsubseteq f(x')\}$, hence by Birkhoff's Representation Theorem, $(\bigsqcup G)(x') = f(x')$. □

Lemma A.2.4 *Let P be a poset and let L be a lattice. Let $x \in P$, let $j \in \mathcal{J}(L)$ and let $m \in \mathcal{M}(L)$. Then*

1. $\lceil x,j \rceil \in \mathcal{J}([P \rightarrow_m L])$;
2. $\lfloor x,m \rfloor \in \mathcal{M}([P \rightarrow_m L])$.

Proof We prove 1 directly, 2 is dual.

Firstly $\lceil x,j \rceil \neq \bot$, since $\lceil x,j \rceil (x) = j$ and $j \neq \bot$ since j is join irreducible. Assume $\lceil x,j \rceil = f \sqcup g$. We must show that $\lceil x,j \rceil = f$ or $\lceil x,j \rceil = g$. For any $x' \not\sqsupseteq x$ we have $f(x') \sqcup g(x') = \lceil x,j \rceil (x') = \bot$, so $f(x') = g(x') = \bot$. It remains to show that either $f(x') = j$ for all $x' \sqsupseteq x$, or $g(x') = j$ for all $x' \sqsupseteq x$. Consider $x' = x$: we have $\lceil x,j \rceil (x) = j = f(x) \sqcup g(x)$. But j is join irreducible, so either $f(x) = j$ or $g(x) = j$. Suppose without loss of generality that $f(x) = j$. Then for all $x' \sqsupseteq x$, we have $f(x') \sqsupseteq j$ and $f(x') \sqcup g(x') = \lceil x,j \rceil (x') = j$, hence $f(x') = j$. □

Theorem 8.2.1 *Let P be a finite poset and let L be a finite distributive lattice. Then*

1. $\mathcal{J}([P \rightarrow_m L]) = \{\lceil x,j \rceil \mid x \in P, j \in \mathcal{J}(L)\}$;
2. $\mathcal{M}([P \rightarrow_m L]) = \{\lfloor x,m \rfloor \mid x \in P, m \in \mathcal{M}(L)\}$.

Proof We prove 1 directly, 2 is dual.

By Lemma A.2.4 it remains to show that for any $f \in \mathcal{J}([P \rightarrow_m L])$, there is some $x \in P$ and $j \in \mathcal{J}(L)$ such that $f = \lceil x,j \rceil$. Let $f \in \mathcal{J}([P \rightarrow_m L])$. By Lemma A.2.3, we have that

$$f = \bigsqcup\{\lceil x,j \rceil \mid x \in P, j \in \mathcal{J}(L), \lceil x,j \rceil \sqsubseteq f\}.$$

But by Lemma 8.1.5 this implies that there is some $x \in P$ and $j \in \mathcal{J}(L)$ such that $f \sqsubseteq \lceil x,j \rceil \sqsubseteq f$. □

Bibliography

[Abr86] Samson Abramsky. Strictness analysis and polymorphic invariance. In *Programs as Data Objects*. Springer Verlag, 1986. LNCS 217.

[Abr90] Samson Abramsky. Abstract interpretation, logical relations and Kan extensions. *Journal of Logic and Computation*, 1(1):5–39, 1990.

[AJ91] Samson Abramsky and Thomas P. Jensen. A relational approach to strictness analysis of higher order polymorphic functions. In *Proc. ACM Symposium on Principles of Programming Languages*, 1991.

[Ama91] Roberto M. Amadio. Recursion over realizabilty structures. *Information and Computation*, 91:55–85, 1991.

[AMSW90] Samson Abramsky, John Mitchell, Andre Scedrov, and Phil Wadler. Relators. Draft paper, 1990.

[AP90] M. Abadi and G. Plotkin. A per model of polymorphism and recursive types. In *Logic in Computer Science*. IEEE, 1990.

[Asp90] A. G. Asperti. *Categorical Topics in Computer Science*. PhD thesis, Università di Pisa, 1990.

[Bar91a] Gebreselassie Baraki. A note on abstract interpretation of polymorphic functions. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference*. Springer-Verlag, August 1991. LNCS 523.

[Bar91b] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[Ber78] G. Berry. Stable modeles of typed lambda-calculi. In *5th International Colloquium on Automata, Languages and Programming*, 1978. LNCS 62.

[BFSS90] E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. In G. Huet, editor, *Logical Foundations of Functional Programming*. Addison Wesley, 1990.

[BHA86] Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. Strictness analysis of higher-order functions. *Science of Computer Programming*, 7:249–278, November 1986.

[Bur87] Geoffrey L. Burn. *Abstract Interpretation and the Parallel Evaluation of Functional Languages*. PhD thesis, Department of Computing, Imperial College of Science and Technology, University of London, 1987.

[Bur90] Geoffrey L. Burn. A relationship between abstract interpretation and projection analysis (extended abstract). In *POPL 90, Conference on Principles of Programming Languages, 17–19 January, 1990, San Francisco, USA*. ACM, January 1990.

[Bur91] Geoffrey L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1991.

[BW90] Michael Barr and Charles Wells. *Category Theory for Computing Science*. International Series in Computer Science. Prentice Hall, 1990.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles Of Programming Languages*, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *6th ACM Symposium on Principles Of Programming Languages*, 1979.

[CJ85] Chris Clack and Simon Peyton Jones. Strictness analysis - a practical approach. In *Functional Programming Languages and Computer Architecture*, pages 35–49. Springer Verlag, September 1985. LNCS 201.

[CL90] Luca Cardelli and Giuseppe Longo. A semantic basis for quest (extended abstract). In *Lisp And Functional Programming*. ACM Press, 1990.

[DP90] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[Fre89] P. Freyd. Structural polymorphism. Unpublished, 1989.

[GHK+80] G.Gierz, K.H. Hofmann, K.Keimel, J.D. Lawson, M.Mislove, and D.S. Scott. *A Compendium of Continuous Lattices*. Springer-Verlag, 1980.

[GS90] Carl A. Gunter and D.S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. North-Holland, 1990.

[Gun91] Carl A. Gunter. Structures and techniques for the semantics of programming languages. Third draft of unpublished notes for a course taught at the University of Pennsylvania., January 1991.

[HH91] Sebastian Hunt and Chris Hankin. Fixed points and frontiers: A new perspective. *Journal of Functional Programming*, 1(1), 1991.

[HL90] John Hughes and John Launchbury. Towards relating forwards and backwards analysis. In *Proceedings of the Third Annual Glasgow Workshop on Functional Programming*. Springer Verlag, 1990. to appear in workshop series.

[HL91] John Hughes and John Launchbury. Projections for polymorphic first-order strictness analysis. To Appear in Mathematical Structures in Computer Science, 1991.

[HS91] Sebastian Hunt and David Sands. Binding time analysis: A new PERspective. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'91)*. ACM Press, 1991.

[Hug88] John Hughes. Abstract interpretation of first-order polymorphic functions. In *Glasgow Workshop on Functional Programming*, University of Glasgow, Department of Computing Science, August 1988. Research Report 89/R4.

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

[Hun90a] Sebastian Hunt. PERs generalise projections for strictness analysis. Technical Report DOC 90/14, Imperial College, 1990.

[Hun90b] Sebastian Hunt. PERs generalise projections for strictness analysis (extended abstract). In *Proceedings of the Third Annual Glasgow Workshop on Functional Programming*. Springer Verlag, 1990.

[HY86] Paul Hudak and Jonathan Young. Higher-order strictness analysis in untyped lambda calculus. In *13th Conference on Principles of Programming Languages*, January 1986.

[JC87] Simon Peyton Jones and Chris Clack. Finding fixpoints in abstract interpretation. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 11. Ellis Horwood, 1987.

[Jen91] Thomas P. Jensen. Strictness analysis in logical form. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference*. Springer-Verlag, August 1991. LNCS 523.

[JM86] Neil D. Jones and Alan Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *13th Conference on Principles of Programming Languages*, pages 296–306, 1986.

[Jon88] Neil D. Jones. Automatic program specialization: A re-examination from basic principles. In D. Bjørner, A.P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 225–282. North-Holland, 1988.

[Kam91] Samuel Kamin. Head-strictness is not a monotonic abstract property. To appear in Information Processing Letters, 1991.

[Lau88] John Launchbury. Projections for specialisation. In D. Bjørner, A.P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 299–315. North-Holland, 1988.

[Lau89] John Launchbury. *Projection Factorisations in Partial Evaluation*. PhD thesis, Department of Computing, University of Glasgow, 1989.

[LS86] J. Lambek and P.J. Scott. *Introduction to higher order categorical logic*. Cambridge studies in advanced mathematics. Cambridge University Press, 1986.

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

[Mey85] Albert Meyer. Complexity of program flow analysis for strictness: Application of a fundamental theorem of denotational semantics. unpublished note, 1985.

[MH87] Chris Martin and Chris Hankin. Finding fixed points in finite lattices. In *Functional Programming Languages and Computer Architecture*, pages 426–445. Springer Verlag, September 1987. LNCS 274.

[MJ85] Alan Mycroft and Neil D. Jones. A relational framework for abstract interpretation. In Neil D. Jones, editor, *Programs as Data Objects*. Springer-Verlag, 1985. LNCS 215.

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

[Nie84] Flemming Nielson. *Abstract Interpretation using Domain Theory*. PhD thesis, University of Edinburgh, 1984.

[Nie85] Flemming Nielson. Expected forms of data flow analyses. Technical Report R 85–9, Institute of Electronic Systems, Aalborg, Denmark, May 1985.

[Nie89] Flemming Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science*, 69:117–242, 1989.

[Plo73] G. Plotkin. Lambda definability and logical relations. Memorandum SAI-RM-4, Department of AI, University of Edinburgh, 1973.

[Plo77] G. Plotkin. LCF considered as a programming language. In *Theoretical Computer Science, vol 5.*, pages 223–256. North-Holland Publishing Company, 1977.

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

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing*, 83:513–523, 1983.

[SP82] M. Smyth and G. Plotkin. The category theoretic solution of recursive domain equations. *SIAM J. Comput.*, 11(4):761–783, November 1982.

[Sta85] R. Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65:85–97, 1985.

[Wad87] Philip Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 12. Ellis Horwood, 1987.

[WH87] Philip Wadler and John Hughes. Projections for strictness analysis. In *LNCS 274. Functional Programming Languages and Computer Architectures, Oregon, USA*, 1987.

[YH86] Jonathan Young and Paul Hudak. Finding fixpoints on function spaces. Technical Report YALEEU/DCS/RR-505, Yale University, Department of Computer Science, 1986.