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

Abstract Interpretation In Logical Form

Thomas Philip Jensen

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

November 1992

Abstract

The topic of this thesis is the relationship between program analyses formulated as type systems and analyses formulated as abstract interpretations. As one of the main results we show the correspondence between abstract interpretation for higher order functional languages and a type system with conjunctive types. We use this correspondence as a starting point for developing new analyses such that each analysis has both a type system and an abstract interpretation formulation. On one side this gives a characterisation of the abstract interpretation in terms of a logical system; on the other side the abstract interpretation gives an effective technique for implementing the logic.

Non-standard type systems are program logics defined over a preorder of properties where the ordering is logical entailment. By grouping the properties into sets of properties closed under entailment, we obtain a partial order over which we can then define a denotational semantics such that the denotation of a program is the set of properties provable of the program in the logic. Similarly we show how a partial order gives rise to a set of properties by taking certain subsets of the partial order to represent the properties. In this way a program satisfies a property if its denotation belongs to the set of elements representing the property. This provides a means of passing between the partial orders used in abstract interpretation and the formal systems of properties underlying the type systems.

As a first application of this correspondence we give an abstract interpretation strictness analysis for higher order functional languages and prove it equivalent to a program logic with conjunctive properties. It has been observed that abstract interpretation can prove more properties than type systems that lack conjunctions; this result provides a precise characterisation of the relationship. We then consider the issue of increasing the power of the analysis by adding disjunctions to the program logic and find an equivalent abstract interpretation based on lower-closed subsets of partial orders. This extends earlier work on capturing disjunctive properties by using tensor products of lattices.

Finally we construct a program logic for reasoning about uniform properties of recursive data structures. We axiomatise uniform properties in a way similar to the axiomatic description of the Plotkin powerdomain. This means that the partial order modelling the properties can be obtained as a partial order of convex subsets as in the Plotkin powerdomain. The corresponding abstract interpretation defined over these partial orders embodies the standard strictness analysis for lists and offers a rational reconstruction of the elements in Wadler's four point domain.

Acknowledgments

Chris Hankin deserves my warmest thanks for having supervised this PhD work. Chris has always been a source of support and helpful comments. Thanks are also due to my friends and office mates Sebastian Hunt and Dave Sands for providing a stimulating research environment where all matters, technical as well as not so technical, could be discussed. Samson Abramsky and Geoff Burn have always been willing to share their insight and listen to my ideas. They, together with many other people in the Computing Department, have made my PhD studies at Imperial College a rewarding experience. Thanks to Paul Taylor for his excellent TEX-macros.

Outside Imperial, I have benefitted from the interaction with the research groups at DIKU (Copenhagen), University of Glasgow and Ecole Polytechnique (Paris) made possible by the ESPRIT Basic Research Action SEMANTIQUE. Their critical remarks on presentations of the ideas lying behind this thesis have been very valuable to me. Thanks to my examiners Peter Burton and Phil Wadler and to Nick Benton, Chris Colby, Bob Muller and Mitchell Wand for useful comments.

This research was funded first through the ESPRIT BRA SEMANTIQUE and later by a scholarship from Department of Computer Science, University of Copenhagen (DIKU). Additional funding for travel has been provided by The Danish Research Council. I thank them all for their generous support.

I would like to express my gratitude towards my parents who have always taken great interest in my academic career. Without their encouragement over the years this thesis would never have come about. Finally, thanks to Sandra whose love and support made the writing of this thesis a worthwhile effort.

Contents

Abstract **3**

Acknowledgments **4**

1 Program analysis **11**

1.1 Type systems for program analysis 13

1.2 Abstract interpretation 14

1.2.1 Abstract interpretation of functional languages 15

1.3 Overview of the thesis 17

2 Logics and lattices **20**

2.1 Conjunctive systems 21

2.1.1 Approximable mappings 22

2.1.2 The Lindenbaum algebra 23

2.2 From properties to denotations 24

2.3 From denotations to properties 26

2.4 An equivalence 27

2.5 Filter models and domain logic 33

2.6 Summary 34

3 Conjunctive program logics **35**

3.1 The typed lambda calculus 35

3.2 Axiomatisation of conjunctive properties 37

3.2.1 Semantics of formulae 40

3.3 Strictness analysis 45

3.3.1 Abstract interpretation 45

3.3.2 Strictness logic 45

3.3.3 Soundness and Completeness Theorems 47

3.4 Binding time analysis 51

3.5 Intersection types 54

3.6 Summary 56

4 Disjunctive properties **57**

4.1 Introduction 57

4.2 Axiomatisation 58

4.3 Partial order semantics of formulae 61

4.4 Disjunctive strictness analysis 70

4.5 Disjunctive abstract interpretation 72

4.6 Soundness and completeness 76

4.7 Relational program analysis 80

4.8 Summary 82

5 Uniform properties **84**

5.1 Introduction 84

5.2 A formal system of uniform properties 85

5.3 Convex sets 88

5.4 Examples . . . 93
5.5 Operations on sets . . . 95
5.6 The cone powerdomain . . . 98
5.7 Summary . . . 100

6 Recursive Data Structures **101**

6.1 Introduction . . . 101
6.2 Algebraic data types . . . 102
6.2.1 Semantics of recursive data structures . . . 103
6.2.2 Data structures as multi-sets . . . 104
6.3 Sum types . . . 105
6.3.1 Operations on sums . . . 108
6.4 Recursive types . . . 109
6.4.1 The rules for fold and unfold . . . 110
6.5 Analysing length and map . . . 113
6.5.1 Computing the length of an infinite list . . . 113
6.5.2 Mapping a strict function over a list containing an undefined element . . . 114
6.6 Abstract interpretation of sums . . . 115
6.6.1 Lattices for sum types . . . 115
6.6.2 Abstract interpretation of in and case . . . 119
6.7 Lattices for recursive types . . . 124
6.8 Abstract interpretation of fold and unfold . . . 125
6.9 Operations on lists . . . 131
6.10 Uniform properties and binding time analysis . . . 133

6.11 Summary 134

7 Conclusions **136**

7.1 Summary and appraisal 136

7.2 Related topics and further work 139

7.2.1 Polymorphism 139

7.2.2 Implementations 140

7.2.3 Backwards analysis in logical form 141

7.2.4 Other logical frameworks 143

7.3 Final remarks 144

Bibliography **145**

List of Figures

3.1 Simply typed lambda calculus 36

3.2 Axiomatisation of conjunctive properties 39

3.3 Abstract interpretation for strictness analysis 46

3.4 Conjunctive strictness logic for simply typed lambda calculus 47

3.5 An example derivation in the conjunctive logic 48

4.1 Axiomatisation of disjunctive properties 59

4.2 Types modelled as lower sets 63

4.3 Semantics of formulae 65

4.4 Disjunctive strictness logic 70

4.5 Disjunctive abstract interpretation 73

5.1 Semantics of uniform formulae 90

5.2 The lattice $[\![P(\mathsf{Int})]\!] = \mathsf{D}(P_C(J(\mathsf{Int})))$ 94

6.1 Axiomatisation of sum properties 106

6.2 Rules for in and case. 108

6.3 Rules for fold 111

6.4 Rules for unfold. 112

6.5 Semantics of sum formulae. 116

6.6 The partial order $J(\mathsf{Int\ List})$ 124

6.7 Abstract interpretation of fold and unfold. 126

Chapter 1

Program analysis

Analysing a program is an experience familiar to anyone who has ever been involved in the development of any substantial computer application. Any sort of behaviour that does not conform with our expectations calls for closer scrutiny of the program to identify what caused this discrepancy. Any kind of changes we might want to make to the program requires that we convince ourselves that the changes will not cause any unexpected results.

The methods employed in the analysis process vary in the degree of automation. At one end of the spectrum we find informal reasoning based on common sense and perhaps supported by some calculations by hand. Next to that is the use of automated tools like debuggers and proof-checkers to verify automatically hypotheses we might form about the program's behaviour. At the other end of the spectrum are the fully automated program analysers which can analyse programs without having to rely on human intervention to direct the analysis process. One attractive aspect of the latter kind of analyser is that it does not require the person using it to have a deep understanding of how the analysis method works. The drawback is that for computability reasons a fully automated analyser can only give approximate information about the programs it analyses. Thus in some cases the analyser might answer *"don't know"* to a question even though there is a precise answer to the question. Fully automated program analysis, or static program analysis as it is also called, is the topic of this thesis.

One reason for the interest in static program analysis is its use in programming language processors, *i.e.*, programs that manipulate other programs. Here we think in particular of optimising compilers and program transformation systems. Common to these programs is that they aim at improving their input program with respect to some parameter. To make these improvements some information about the be-

haviour of the input program is usually required, necessitating an analysis of the program. If we want our compilers and program transformers to be fully automatic we therefore need fully automated analyses. We shall here list some examples of information that is of interest:

- The input-output behaviour of a program. Given a specific piece of input, what will the output of the program be? In a more detailed analysis we might also ask what values the variables of a program obtain either at a particular point in the execution or during the whole execution. In particular we might want to know whether a variable always has the same value throughout a computation. A number of optimisations can be done by a compiler based on this information [ASU86].

- The termination properties of a program. Will it always terminate? Is there input for which the program does not terminate? This is considered by Mycroft and Abramsky [Myc81, Abr90].

- The use of storage and the run-time of a program. When is a piece of data not used any more? The use of storage in a program can be improved significantly if we are able to tell when it is safe to reuse a particular part of storage. A good strategy for reusing storage can lead to less *garbage collection* thus improving the overall run-time of a program. These problems have been investigated in a large number of publications [Sch78, Myc81, Hud87, JM89, JM90, San90, Wad91].

- Does a program use its input during a computation? Can we perform some of the computation without knowledge of the input? Information of this kind is of importance in *partial evaluation*, especially in its application to compiler generation [JSS89, Lau89].

- In which order will a program evaluate its expressions. How are the variables of a program accessed? Information of this kind can for example be used to infer whether a local variable can be allocated globally or has to be allocated on the stack for each new invocation of the procedure to which it is local. For recent treatments of these issues see the theses by Sestoft and Gomard [Ses91, Gom91].

Analyses can be classified according to what information they are meant to obtain and according to the technique with which this information is obtained. We shall now look at two techniques that can be used to find this information about programs: Non-standard type inference and abstract interpretation.

1.1 Type systems for program analysis

We now briefly review some uses of type systems for strictness analysis and binding time analysis. The question that strictness analysis tries to answer is *"if an argument to a function is undefined, is the result of applying the function to that argument also undefined"*. Such a function is called *strict*. An example of a strict function is the identity function and an example of a non-strict function is the constant function that always returns 17. Strictness information is very useful in optimising lazy languages because if a function is strict in some argument we know that we can evaluate that argument in advance knowing that if evaluation of the argument does not terminate evaluation of the function call would not have terminated. Thus we can replace the expensive call-by-name parameter mechanism with the less costly call-by-value parameter passing without changing the termination properties of the program. Strictness analysis has other application as well and has generated a proliferation of papers on analysis of and optimisation based on strictness properties. These ideas appeared in Mycroft's thesis [Myc81], a recent survey of the area can be found in Burn's monograph [Bur91].

Kuo and Mishra [KM87, KM89] advocated the use of type systems for strictness analysis of typed and untyped functional languages. The conjunctive strictness logic presented in Chapter 3 is essentially an extension of their "typed" analysis to include product types as well. The untyped analysis operates with a set of strictness types defined by the grammar

$$\tau = \phi \mid \Box \mid \tau_1 \to \tau_2.$$

The intention is that ϕ is the type of all non-terminating expressions, $\Box$ represents all expressions terminating or not, *i.e.*, this is the *don't know* information, and $\tau_1 \to \tau_2$ denotes those expressions that, when given an argument satisfying τ_1, return an answer satisfying τ_2.

Kuo and Mishra presents a type system for deducing strictness types of programs. The system is similar to type systems with coercions [Mit91]. With this type system we can do simple strictness proofs such as showing that the function *Twice*, defined by

$$Twice \equiv \lambda f.\lambda x.f(f(x)),$$

returns a strict function when applied to a strict function. In the strictness type system this is expressed by the type judgement

$$Twice : (\phi \to \phi) \to \phi \to \phi$$

and the following type inference is a proof of this typing judgement.

$$\dfrac{\dfrac{\dfrac{\dfrac{f:\phi\to\phi, x:\phi \vdash f:\phi\to\phi \qquad f:\phi\to\phi, x:\phi\vdash f(x):\phi}{f:\phi\to\phi, x:\phi\vdash f(f(x)):\phi}}{f:\phi\to\phi\vdash \lambda x.f(f(x)):(\phi\to\phi)}}{\lambda f.\lambda x.f(f(x)):(\phi\to\phi)\to\phi\to\phi}$$

Binding time analysis is another area where type systems have been applied successfully. Binding time analyses are used in partial evaluators to determine which part of a program can be evaluated given only partial knowledge about the input. A recent account of binding time analysis by type inference can be found in Gomard's thesis [Gom91]. In *loc. cit.* a series of analyses of untyped functional languages is presented, all based on type systems and implemented by what the author calls *partial type inference*. The input to the binding time analysis is a program and a description of which part of the input to the program will be known. The output is an annotated program where the annotations indicate whether an expression can be reduced based on the partial input available or has to be left as *residual code* in the program. The type systems for binding time analysis, like Gomard's type system and others, can thus be viewed as a program logic for deducing when it is possible to evaluate a given part of a program given a specific amount of input.

1.2 Abstract interpretation

It is of paramount importance that an analysis gives safe information about the behaviour of programs. As already argued we cannot expect to get exact information but we must demand that the approximations found by the analysis never lead us to draw conclusions about a program's behaviour that are incorrect. For a program that always returns a positive number as result, an input-output analysis can give the (useless) information *"don't know"* or the (more useful) information *"not negative"* about the answer but it must never claim that the answer is negative. The term *semantics-based program analysis* is frequently used for analyses that have been proved correct with respect to a semantics of the programming language. The desire to develop a generic framework for proving correctness of program analyses lead Cousot and Cousot to the definition of abstract interpretation [CC77, CC79, CC92]. An abstract interpretation interprets a program as a function over a partially ordered set of abstract properties. The partial order $\sqsubseteq$ on abstract properties is the approximation ordering on properties such that if property P entails property Q then $P \sqsubseteq Q$. We shall here present how this has been applied to functional languages.

1.2.1 Abstract interpretation of functional languages

The technique of abstract interpretation was carried over to first order functional languages based on recursion equations by Mycroft [Myc81]. One of the results obtained by this was a strictness analysis formulated as an abstract interpretation. For each type of value we consider two abstract properties:

1	representing all values
|	
0	meaning the undefined value

To handle the recursive definitions the standard semantics is now defined over domains rather than sets and Mycroft uses powerdomains as sets of properties over the standard semantics. The abstract interpretation of an n-ary function is a monotone function $f^{\#} : 2^n \to 2$. Similarly to Cousot and Cousot, Mycroft shows how the abstract interpretation of a program can be defined compositionally, using the operations on the lattice of abstract properties to combine the abstract interpretations of the sub-expressions in a program. For example, an accumulative version of the factorial function might include a function like

$$afac\ m\ n\ =\ \mathsf{if}\ m = 0\ \mathsf{then}\ n\ \mathsf{else}\ afac\ (n-1)\ (n * m)$$

The abstraction of the *afac* function is calculated as follows

$$\begin{aligned} afac^{\#}\ m\ n &= (\mathsf{if}\ m = 0\ \mathsf{then}\ n\ \mathsf{else}\ afac\ (n-1)\ (n*m))^{\#} \\ &= (m=0)^{\#} \sqcap (n^{\#} \sqcup (afac\ (n-1)\ (n*m)))^{\#} \\ &= m \sqcap (n \sqcup afac^{\#}\ (n-1)^{\#}\ (n*m)^{\#}) \\ &= m \sqcap (n \sqcup \mathrm{afac}^{\#}\ n\ (n \sqcap m)) \end{aligned}$$

Another way of reading this recursive definition of $afac^{\#}$ is that $afac^{\#}$ must be a fixed point of the functional

$$\mathcal{F} \equiv \lambda F\ .\ \lambda m\ .\ \lambda n\ .\ m \sqcap (n \sqcup \mathrm{afac}^{\#}\ n\ (n \sqcap m))$$

Again the smaller an abstract value is the more information it carries so the least fixed point is to be preferred. It can be computed by a fixed point iteration starting with approximating $afac^{\#}$ by the function mapping everything to the smallest value 0, *i.e.*,

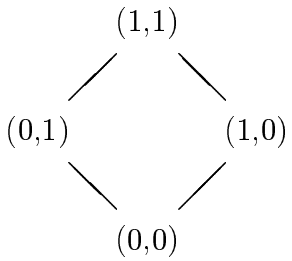
$$afac^{\#} = \bigsqcup_{n=0}^{\infty} \mathcal{F}^n(\lambda x.\lambda y.0)$$

That this is possible relies on the fact that we interpret functions as monotone functions over the two-point lattice. This gives rise to a lattice of functions where the ordering is the pointwise ordering on functions:

$$f \sqsubseteq g \;\equiv\; \forall d \,.\, f(d) \sqsubseteq g(d).$$

This, on the other hand, made it possible to extend abstract interpretation to higher order (typed) languages [BHA86]. A functional object is abstracted to an element in the corresponding lattice of functions between the lattices of abstract properties. This abstract interpretation is the basis of the abstract interpretation presented in Chapter 3.

Abstract interpretation has also been extended to programs using more complex data structures. The interpretation of product types is usually defined by the cartesian product of the interpretation of each type. Assuming that properties of integers are just the two–point domain $\mathbf{2} = \{0 \sqsubseteq 1\}$ where 0 represents the property *"undefined"* and 1 represents *"no information"* the interpretation of $\mathsf{Int}\times\mathsf{Int}$ is then the lattice $\mathbf{2} \times \mathbf{2}$:



Here the point $(0, 1)$ represents the property that the first component of a pair is undefined and $(1, 0)$ represents the property that the first component of a pair is undefined. But $(1, 1)$ represents the *"no information"* of pairs which means that there is a disjunctive property (*"either the first or the second component is undefined"*) that is not represented in the abstraction 2×2 of $\mathsf{Int}\times\mathsf{Int}$. This lead Nielson to suggest that the *tensor* product, rather than the cartesian product, be used to build lattices for product types [Nie85]. We shall later show how an analysis defined using tensor products correspond to a program logic with disjunctions.

The extension of strictness analysis to list structures was considered by Wadler [Wad87]. The difficulty here is that we have to represent to pieces of information: the strictness properties of the elements in list and the degree of definedness of the lsit itself. Since lists can be infinite we cannot list a property for each element in the list. Instead we can use a collection of properties that describes the set of elements as a whole. Wadler suggested the following four point lattice of properties:

$1\in$ all lists

$\mid$

$0\in$ lists containing an undefined element and partial or infinite lists

$\mid$

∞ all partial or infinite lists

$\mid$

$\bot$ the undefined list

The abstract interpretation of the list operations are defined over this lattice such that *e.g.*, we have $\mathsf{cons}^{\#}\ 0\ \ 1 \in = \ \ 0 \in$ and $\mathsf{length}^{\#}\ \infty\ \ = \ \ 0$ representing the facts that adding an undefined element to an arbitrary list results in a list with an undefined element and taking the length of a list whose ending is undefined gives an undefined result. In Chapter 6 we shall use Wadler's ideas to develop a framework for analysing a class of recursive data structures based on properties describing the content of data structures.

1.3 Overview of the thesis

The topic of this thesis is the relationship between analyses formulated as type systems and analyses formulated as abstract interpretations. As one of the main results we show the correspondence between abstract interpretation for higher order functional languages and a type system with conjunctive types. We use this correspondence as a starting point for developing new analyses such that each analysis has both a type system and an abstract interpretation formulation. On one side this gives a characterisation of the abstract interpretation in terms of a logical system; on the other side the abstract interpretation gives an effective technique for implementing the logic.

In **Chapter 2** we set up a framework for relating the partial orders used in abstract interpretation to the formal system of types used in type systems. We argue that the relationship can be viewed as that between a logic and a model: a type is modelled by a set of elements *viz.*, the set of elements satisfying the property and an element corresponds to a set of types *viz.*, the the types (properties) the element satisfies. This intuition is formalised by the notions of ideals and filters in semi-lattices and

we show that for finite lattices this correspondence is a duality of categories

$$\mathbf{JSL}_f \underset{\mathsf{Fil}}{\overset{\mathsf{Idl}}{\rightleftarrows}} \mathbf{MSL}_f^{op}.$$

This duality is a special case of *Stone Duality*. The implication of this duality is that the partial orders and logical systems are in one-to-one correspondence. Given an object in $\mathbf{JSL}_f$ we can derive a corresponding logical system in $\mathbf{MSL}_f^{op}$ and *vice versa*.

In **Chapter 3** we use this duality to relate the partial orders in an abstract interpretation of the typed lambda calculus to a set of conjunctive types. We then show that the abstract interpretation is a *sound and complete* model of a type system with conjunctive types in the sense that the set of properties provable of a program using the type system corresponds via duality to the abstract interpretation of that term. In **Chapter 4** we extend the logic to include disjunctions of properties. This added power is particularly useful for analysing data structures and is a prerequisite for the developments in the subsequent chapters. Disjunctive properties are modelled by lower sets of partial orders. We construct an abstract interpretation equivalent to the disjunctive program logic and show that for properties of product type our interpretation corresponds to earlier analyses based on tensor products. This provides an explicit characterisation of the extra logical power the use of tensor products adds to an abstract interpretation and extends earlier work by Nielson from first order to higher order functions.

In **Chapter 5** we develop a formal system intended for analysing the content of data structures. We operate with two kind of properties: *"all elements satisfy a property P"* and *"there exists an element satisfying property P"*. The rules used in the axiomatisation are taken from an axiomatisation of the Plotkin powerdomain. We construct a model of these uniform properties based on convex sets. This model is then used in **Chapter 6** where we extend Wadler's strictness analysis for lists to uniform strictness properties of a simple class of recursively defined algebraic data structures. The extension is based on a refined system of sum type properties and is given in terms of logical rules for the fold and unfold operations associated with recursive data structures. Apart from the extension of the analysis to a greater variety of types this also provides a clear logical characterisation of the elements in the abstract lattices used in Wadler's original analysis. The work presented hence reconstructs Wadler's list properties in terms of uniform properties but the resulting lattice of properties contains distinct points representing the same properties of lists, a problem avoided in Wadler's original work. Finally **Chapter 7** summarises the

thesis, gives an assessment of the results obtained and discusses related and future work.

Notes to the reader

The thesis is about semantics-based program analysis and to fully appreciate the work reported some prior knowledge about semantics of programming languages would be beneficial. We recommend the books by Schmidt and Tennent [Sch86, Ten91] as introductions. To get an overview of the field of program analysis we recommend the collection by Abramsky and Hankin [AH87]. Although much has happened since, this is still a useful source. In the thesis we employ a certain amount of theory of partially ordered sets, which can all be found in the excellent introduction by Davey and Priestley [DP90]. The amount of category theory present in this thesis is limited and serves only organisational purposes. It can all be found in standard textbooks [Mac71, BW90].

Chapter 2

Logics and lattices

The purpose of any program analysis activity is to establish that a program satisfies certain properties. A clear understanding of the properties we want to detect is therefore a prerequisite for obtaining reliable program analyses. If these analyses were to be automated it would furthermore require that we could give a presentation of these properties together with rules for how they relate to each other that could be represented in a machine. This chapter introduces *conjunctive systems* as a syntactic description of the properties underlying the program analyses considered in the chapters to follow. The logic we consider is a particularly simple kind of propositional logic; the only logical connective being conjunction. The presentation of the logic consists of a collection of (syntactic) formulae together with a preorder (the *entailment* relation) on formulae. In this syntactic presentation we have formulae that entail each other *i.e.*, are logically equivalent but still are syntactically different. Equivalent formulae represent the same property, so by grouping logically equivalent formulae together in equivalence classes we obtain a structure where each element represents one of the properties formalised by the logic. This structure is known as the *Lindenbaum algebra* [Men79] of the logic.

To create the link with the domains used in abstract interpretation we show how the Lindenbaum algebra of a conjunctive system determines a partial order where each element corresponds to a set of properties *viz.*, the properties which the element satisfies. These partial orders will serve as domains in an abstract interpretation. Furthermore, there is a natural definition of what the properties of such a domain are. The central result of the chapter is that if a domain is obtained from a Lindenbaum algebra then the system of properties over this domain is the same logical system as that described by the Lindenbaum algebra. The construction used to show this correspondence is a special case of a general concept known as *Stone Duality* [Joh82]. We shall give the construction and show how the main result follows.

2.1 Conjunctive systems

A formal logical system involves a set of formulae and a set of logical connectives that enables us to build complex formulae from simpler ones. The set of formulae must be closed under formation of new formulae. It furthermore involves a preorder (the entailment relation $\leq$) that conveys the logical relationship between formulae. The entailment relation can be specified either by listing the entire relation or by providing a set of rules for deducing when a formula entails another. We shall do the latter here. We define the notion of a *conjunctive system* that models the logical structure on the properties from our program analyses, together with *approximable mappings* that are used to relate two conjunctive systems. This is inspired by information systems [Sco82, LW84] and extensions thereof [Abr91b]. By quotienting the preorder of formulae by provable equality we obtain a partial order (in fact, a meet-semilattice) called the Lindenbaum algebra of the system. The Lindenbaum algebra can be given the structure of a conjunctive system and is as such isomorphic to the conjunctive system it is derived from. The stronger algebraic properties of the Lindenbaum algebra makes it in some respects easier to reason with.

Definition 2.1.1 *A* conjunctive system *is a structure*

$$\mathbf{A} = (A, 1_A, \wedge_A, \leq_A, =_A)$$

where A is a set (the carrier of the structure), 1_A is an element of A, $\wedge_A$ is a binary operation on A and $\leq_A, =_A$ are relations on A, satisfying the axioms:

- $\varphi \leq \varphi$
- $\dfrac{\varphi \leq \psi, \psi \leq \chi}{\varphi \leq \chi}$
- $\varphi \wedge \psi \leq \varphi$
- $\dfrac{\varphi \leq \psi_1, \varphi \leq \psi_2}{\varphi \leq \psi_1 \wedge \psi_2}$
- $\varphi \wedge \psi \leq \psi$
- $\dfrac{\varphi \leq \psi, \psi \leq \varphi}{\varphi = \psi}$
- $\dfrac{\varphi = \psi}{\varphi \leq \psi, \psi \leq \varphi}$
- $\varphi \leq 1_A$

Examples The one-point conjunctive system **1** freely generated over the one-point set $\{\bullet\}$ where $\bullet = 1_{\mathbf{1}}$. The carrier of **1** is the set $\{\bullet, \bullet \wedge \bullet, \bullet \wedge \bullet \wedge \bullet, \ldots\}$. All formulae are equal.

The conjunctive system **2** freely generated over the set $\{0, 1\}$. The carrier consists of formulae $0, 1, 0 \wedge 1, 1 \wedge 1, \ldots$. The relation $\leq$ is defined to be the smallest relation on formulae satisfying the axioms.

2.1.1 Approximable mappings

We used a relation, $\leq$, to describe how formulae within one conjunctive system entailed each other. Similarly, we can use a relation to describe how properties in one conjunctive system entail properties in another. We use relations since we want to relate a formula to all the formulae it entails and this relation need not be representable by a function from formulae to formulae. Such a relation is called an *approximable mapping*. Approximable mappings must respect the logical structure in the conjunctive systems in the sense that if a property a in one system entails both property b_1 and b_2 in another system then it must also entail the property $b_1 \wedge b_2$. Furthermore, assume $a_2 \leq a_1$ in one system $\mathbf{A}$ and $b_1 \leq b_2$ in another system $\mathbf{B}$; if a_1 entails property b_1 then we would also expect the stronger property a_2 to entail the weaker property b_2. We define:

Definition 2.1.2 *Let* $\mathbf{A} = (A, 1_A, \wedge_A, \leq_A, =_A)$ *and* $\mathbf{B} = (B, 1_B, \wedge_B, \leq_B, =_B)$ *be conjunctive systems. An approximable mapping* f *from* $\mathbf{A}$ *to* $\mathbf{B}$ *is a relation* $f \subseteq A \times B$ *satisfying*

i) $a\ f\ b_1$ *and* $a\ f\ b_2$ *implies* $a\ f\ b_1 \wedge b_2$

ii) $a_2 \leq a_1\ f\ b_1 \leq b_2$ *implies* $a_2\ f\ b_2$

Approximable mappings are composed by the usual relational composition. It is easy to verify that relational composition of two approximable mappings yields an approximable mapping. The identity approximable mapping on $\mathbf{A}$, id_A, is defined by:

$$a\ id_A\ b \equiv a \leq b.$$

We thus get a category **CONJ** of conjunctive systems and approximable mappings. In particular we can talk about when two conjunctive systems are isomorphic. Informally, we would say that two conjunctive systems, $\mathbf{A}$ and $\mathbf{B}$, represent the same logical structure if there is a map f from the carrier of one system onto the carrier of the other system such that

$$a_1 \leq_A a_2 \ \Leftrightarrow\ f(a_1) \leq_B f(a_2).$$

That this indeed implies isomorphism is proved in the following lemma.

Lemma 2.1.3 *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *be conjunctive system and let* f *be a map from the carrier* A *of* $\mathbf{A}$ *to the carrier* B *of* $\mathbf{B}$ *such that* f *is surjective and for all* a_1, a_2 *in*

A:

$$a_1 \leq_A a_2 \;\Leftrightarrow\; f(a_1) \leq_B f(a_2).$$

Then f induces an isomorphism R_f between $\mathbf{A}$ *and* $\mathbf{B}$ *defined by*

$$aR_f b \equiv f(a) \leq_B b.$$

Proof. It is straightforward to see that this defines an approximable mapping. We now define an approximable mapping R_f^{-1} from $\mathbf{B}$ to $\mathbf{A}$ and show that this is indeed the inverse in the category **CONJ**. Define

$$bR_f^{-1}a \equiv \exists a' \in f^{-1}(b) \,.\, a' \leq a.$$

We show $R_f^{-1} \circ R_f = id$ in **CONJ**. The other equation required for isomorphism, $R_f \circ R_f^{-1} = id$ follows by a similar argument.

$\subseteq$. If $a_1 \; (R_f^{-1} \circ R_f) \; a_2$ then there exists a $b \in B$ such that

$$f(a_1) \leq b \qquad \text{and} \qquad \exists a' \in f^{-1}(b) \,.\, a' \leq a_2.$$

Then $f(a_1) \leq b = f(a') \leq f(a_2)$ which implies that $a_1 \leq a_2$.

$\supseteq$. If $a_1 \leq a_2$ then with $b = f(a_1)$ we have $a_1 \; R_f \; b$ and $b \; R_f^{-1} \; a_2$ which means that $a_1 \; (R_f^{-1} \circ R_f) \; a_2$. ■

We shall later use this lemma when showing two conjunctive systems isomorphic.

2.1.2 The Lindenbaum algebra

The entailment relation on a conjunctive system is only required to be a preorder on the carrier set. This is done in order to accommodate those systems where we have syntactically different, but logically equivalent, formulae. Logically equivalent formulae denote the same property so by grouping the formulae into equivalence classes we obtain a structure that represents the properties axiomatised by the conjunctive system. This structure is called the *Lindenbaum algebra* of the conjunctive system and is a well known construction from logic.

Definition 2.1.4 *Let* $\mathbf{A} = (A, 1_A, \wedge_A, \leq_A, =_A)$ *be a conjunctive system. The* Lindenbaum algebra *of* $\mathbf{A}$ *is the partial order*

$$\mathcal{LA}(\mathbf{A}) = (A \;/{=_A}, \leq_A/{=_A}).$$

For an element $a \in A$ we write $[a]$ for the equivalence class in $A /=_A$ that contains a. The partial order has a top element $[1_A]$. Similarly, the binary operation $\wedge_A /=_A$ on $\mathcal{LA}(\mathbf{A})$ defined by $[a_1] \wedge_A /=_A [a_2] = [a_1 \wedge a_2]$ is well defined.

Theorem 2.1.5 *Let* $\mathbf{A} = (A, 1, \wedge, \leq, =)$ *be a conjunctive system. Then* $\mathcal{LA}(\mathbf{A})$ *is a meet-semilattice with top element* $[1]$ *and meet operation* $\wedge /=$.

Proof. That $[a_1 \wedge a_2]$ is a lower bound for $[a_1]$ and $[a_2]$ is clear. Assume now that $[a] \leq /= [a_1]$ and $[a] \leq /= [a_2]$. Then $a \leq a_1$ and $a \leq a_2$ hence $a \leq a_1 \wedge a_2$ so $[a] \leq /= [a_1 \wedge a_2]$ *i.e.*, $\wedge /=$ is a greatest lower bound operator. ■

Since any meet-semilattice satisfies the axioms of a conjunctive system, the Lindenbaum algebra of a conjunctive system is itself a conjunctive system. What is more, a system and its Lindenbaum algebra are isomorphic in **CONJ**. To see this, observe that the quotient map from a conjunctive system to its Lindenbaum algebra is surjective and that

$$a \leq b \;\Leftrightarrow\; [a] \leq /= [b]$$

i.e., it satisfies the conditions in Lemma 2.1.3 for the existence of an isomorphism between $\mathbf{A}$ and $\mathcal{LA}(\mathbf{A})$.

Since meet-semilattices can be viewed as conjunctive systems we have that all the objects of a category of meet-semilattices will be objects in **CONJ**. Denote by **MSL** the category of meet-semilattices with morphisms the $\wedge$-homomorphisms. Since every $\wedge$-homomorphism between meet-semilattices M, N induces an approximable between M and N, viewed as objects in **CONJ**, we get that the category **MSL** can be embedded in the category **CONJ**.

2.2 From properties to denotations

So far we have been working on the logical side. We have introduced conjunctive systems and their Lindenbaum algebras and we argued that the Lindenbaum algebra could be identified with the properties axiomatised by a conjunctive system. We shall now see how we can construct a partial order which has exactly as many elements as can be distinguished with the properties in the conjunctive system.

The key idea is to take as elements *those sets of properties that can be viewed as describing an element* and then see what structure this collection of sets has. Only sets satisfying certain consistency requirements can be considered as describing an element. The sets must be closed under implication and conjunction because, if

property P_1 holds of an element p and P_1 implies P_2, then P_2 will hold of p as well. Similarly, if properties P_1 and P_2 hold of an element p then so does $P_1 \wedge P_2$. These closure conditions are formalised by the notion of a point of a conjunctive system.

Definition 2.2.1 *A* point *of a conjunctive system* $\mathbf{A} = (A, 1_A, \wedge_A, \leq_A, =_A)$ *is a non-empty subset* $P \subseteq A$ *closed under* $\wedge_A$ *and closed upwards with respect to* $\leq$. *The set of points of a conjunctive system* $\mathbf{A}$ *is denoted by* $|\ \mathbf{A}\ |$.

For example, in the conjunctive system **2** there are two points: the whole carrier of **2** forms a point and so does the set of formulae equivalent to 1_2. In fact, these two points, the whole carrier and the equivalence class [1] occur in every conjunctive system. The former is obviously a superset of any other point; the latter a subset of any point.

We shall be using points as denotations of programs. The denotational semantics (the abstract interpretation) will be designed in such a way that the denotation of a program is the set of properties that are guaranteed to hold of this program. The partial orders used in analyses like strictness analysis and binding time analysis are such that the smaller an element is, the more information it carries. Especially, it is common to have a top element with no information content at all. For points we have that the more formulae a point contains the more information it contains. This suggests that points should be ordered by reverse inclusion where points with more properties are smaller in the ordering.

A common problem in abstract interpretation is to deal with conditional expressions where the execution path is determined at run-time depending on the current state. In an expression like

$$\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$$

it is the value of the boolean expression b that determines whether e_1 or e_2 will be evaluated. Since abstract interpretation in general only provides approximate information about an expression we are in general unable to predict which of the branches will be taken. For the abstract interpretation this means that we are given the denotations of two branches but we do not know which one of them will be chosen. The denotation of the whole choice construct should therefore be those properties that are common to both branches. With the denotation of a branch being the set of properties that hold of that branch, the choice operation can therefore be modelled by *intersection* of points.

This kind of choice operation is usually modelled by the least upper bound operator. With the ordering being reverse inclusion, the points of a conjunctive system do indeed form a join-semilattice with intersection as least upper bound. We recall

that a *join-semilattice* is a partial order $(M, \sqsubseteq)$ with a least element, usually written $\bot$, and a binary operator, $\sqcup$, so that $m \sqcup n$ is the least upper bound of m and n with respect to $\sqsubseteq$.

Theorem 2.2.2 *The points of a conjunctive system $(A, 1_A, \wedge_A, \leq_A, =_A)$ ordered by reverse inclusion, $\supseteq$, form a join-semilattice with A as least element and intersection as the join operation.*

Proof. The set A is upwards closed and closed under $\wedge$ and it is clearly least with respect to the ordering $\supseteq$. Let P, Q be two points and assume that $a_1, a_2 \in P \cap Q$ Then a_1, a_2 belong to both P and Q and as both sets are closed under $\wedge$ we have that $a_1 \wedge a_2 \in P$ as well as $a_1 \wedge a_2 \in Q$ *i.e.*, $a_1 \wedge a_2 \in P \cap Q$. Similarly, if $a \in P \cap Q$ and $a \leq b$ then $b \in P$ and $b \in Q$ as both P and Q are upwards closed, hence $b \in P \cap Q$. So $P \cap Q$ *is* a point and is therefore clearly the least upper bound with respect to $\supseteq$. ■

The join-semilattice arising from the set of points of a conjunctive system satisfies the additional property of having a greatest element *viz.*, the equivalence class [1] corresponding to the trivial property that is always true. Thus not all join-semilattices can be constructed, or *represented*, as the set of points of a conjunctive system. However, finite join-semilattices always have a greatest element so we can hope that such a representation is possible in the finite case. We shall prove in a later section that this is indeed the case.

Since meet-semilattices are conjunctive systems the notion of point makes sense for them as well. A point of a meet-semilattice is called a *filter*.

Definition 2.2.3 *A* filter *in a meet-semilattice M is a non-empty subset $F \subseteq A$ such that A is upwards closed and closed under $\wedge$. The set of filters of M is denoted by $\mathsf{Fil}(M)$. A filter F is* principal *if it is the upwards closure of a single element,* i.e., *if $F = \uparrow(m)$ for some $m \in M$.*

Theorem 2.2.2 gives that the set $\mathsf{Fil}(M)$, ordered by reverse inclusion, is a join-semilattice.

2.3 From denotations to properties

We saw how, given a conjunctive system, we can construct a join-semilattice where the elements consist of properties of the conjunctive system. This allows us to define

that an element satisfies a property precisely when that property is contained in the element. The lattice so obtained can be regarded as the optimal choice of domain of denotations for a semantics where we only use the properties in the underlying conjunctive system to distinguish between objects. The question we set out to answer in this section is: given a domain of denotations, can we find a conjunctive system such that the given domain is the optimal domain corresponding to the conjunctive system in the above sense.

Our assumption on the domains is that they are join-semilattices where the ordering is so that smaller elements satisfy more properties and that the join of two elements a, b is an element that satisfies those properties satisfied by both a and b. A property over a domain is identified with the subset of elements for which the property is true. We can characterise the subsets representing properties as follows. If an element a satisfies a property P then so will any element smaller than a, so the set of elements satisfying a particular property is downwards closed. Similarly, if two elements satisfy a property P then so does $a \sqcup b$ so the set should be closed under the join operation. Subsets satisfying these conditions are called *ideals*.

Definition 2.3.1 *An* ideal *of a join-semilattice $(S, \sqcup, \bot, \sqsubseteq)$ is a non-empty subset $I \subseteq S$ that is downwards closed with respect to $\sqsubseteq$ and closed under $\sqcup$. The set of ideals of a join-semilattice S is written $\mathsf{Idl}(S)$. An ideal I is* principal *if it is the down-closure of a single element a,* i.e., *$I = {\downarrow}\{a\}$ for some $a \in S$.*

Theorem 2.3.2 *Let $(S, \sqsubseteq, \sqcup)$ be a join-semilattice. Then the structure*

$$(\mathsf{Idl}(S), S, \cap, \subseteq)$$

forms a meet-semilattice and hence a conjunctive system.

Proof. S is clearly greatest in the ordering $\subseteq$. We verify that intersection of two ideals yields another ideal. Given ideals I_1, I_2 and $a, b \in I_1 \cap I_2$, we get that $a, b \in I_1$ and $a, b \in I_2$ hence $a \sqcup b \in I_1$ and $a \sqcup b \in I_2$ so $a \sqcup b \in I_1 \cap I_2$ *i.e.*, $I_1 \cap I_2$ is closed under $\sqcup$. By a similar argument $I_1 \cap I_2$ is downwards closed, so intersection does map from ideals to ideals. The set $I_1 \cap I_2$ is obviously the greatest lower bound of I_1 and I_2 with respect to the ordering $\subseteq$. ■

2.4 An equivalence

We have seen how a join-semilattice of denotations gives rise to a meet-semilattice of properties and *vice versa* via the maps Fil and Idl. We now extend the maps Fil

and Idl such that they become functors between a category of join-semilattices and a category of meet-semilattices. Using the categorical framework we prove that, under the assumption of finiteness, two objects in one category are isomorphic if and only if their images in the other category are isomorphic. It should be noted that the result presented in this section is well-known [Joh82, p. 251] where it comes out as a corollary of a general theory. The following is a direct proof of this corollary.

Let $\mathbf{JSL}_f$ denote the category of *finite* join-semilattices with morphisms the $\vee$-homomorphisms and let $\mathbf{MSL}_f$ denote the category of *finite* meet-semilattices with morphisms the $\wedge$-homomorphisms. The opposite category $\mathbf{MSL}_f^{op}$ has the same objects as $\mathbf{MSL}_f$ but a $\mathbf{MSL}_f^{op}$-morphism from M to N is a $\wedge$-homomorphism from N to M.

We extend Fil and Idl so that they become functors

$$\mathsf{Fil} : \mathbf{MSL}_f^{op} \to \mathbf{JSL}_f$$
$$\mathsf{Idl} : \mathbf{JSL}_f \to \mathbf{MSL}_f^{op}.$$

Let $f : M \to N$ be a morphism in $\mathbf{MSL}_f^{op}$, *i.e.*, f is a $\wedge$-homomorphism from N to M. We define $\mathsf{Fil}(f) : \mathsf{Fil}(M) \to \mathsf{Fil}(N)$ by

$$\mathsf{Fil}(f)(F) = f^{-1}(F)$$

for arbitrary F in $\mathsf{Fil}(M)$. To ensure that this does indeed define a filter observe that if $n_1, n_2 \in N$ such that $f(n_1) \in F$ and $f(n_2) \in F$ then $f(n_1 \wedge n_2) = f(n_1) \wedge f(n_2) \in F$ since F itself is a filter. Upwards closure follows by a similar argument. Toe see that $\mathsf{Fil}(f)$ is a $\vee$-homomorphism observe that

$$\mathsf{Fil}(f)(F \cap G) = f^{-1}(F \cap G) = f^{-1}(F) \cap f^{-1}(G) = \mathsf{Fil}(f)(F) \cap \mathsf{Fil}(f)(G).$$

Let then $m : J \to K$ be a $\vee$-homomorphism between join-semilattices J and K. To define a morphism $\mathsf{Idl}(m) : \mathsf{Idl}(J) \to \mathsf{Idl}(K)$ in $\mathbf{MSL}_f^{op}$ amounts to defining a $\wedge$-homomorphism from $\mathsf{Idl}(K)$ to $\mathsf{Idl}(J)$. So for given ideal $I \subseteq K$ we define

$$\mathsf{Idl}(m)(I) = m^{-1}(I).$$

Since m preserves joins this does define a mapping from ideals to ideals.

Lemma 2.4.1 Fil *and* Idl *are functors.*

Proof. That Fil and Idl preserve identities is straightforward. If $f : M \to N$

and $g : N \to O$ are $\mathbf{MSL}_f^{op}$ morphisms, *i.e.*, $\wedge$-homomorphisms $\hat{f} : N \to M$ and $\hat{g} : O \to N$, their composite is defined to be the $\wedge$-homomorphism $\hat{f} \circ \hat{g} : O \to M$, so $\mathsf{Fil}(g \circ f) = (\hat{f} \circ \hat{g})^{-1} = \hat{g}^{-1} \circ \hat{f}^{-1} = \mathsf{Fil}(g) \circ \mathsf{Fil}(f)$ which shows that Fil preserves composition. A similar argument shows that Idl preserves composition. ■

Given a join-semilattice J we can construct the meet-semilattice of ideals of J, which we consider to be the properties that are definable over J. From this meet-semilattice of properties we can go on to build the join-semilattice of filters of $\mathsf{Idl}(J)$. Informally, we can say that $\mathsf{Fil}(\mathsf{Idl}(J))$ is what J "ought to be" if J is to model the properties $\mathsf{Idl}(J)$. An important question is therefore how J and $\mathsf{Fil}(\mathsf{Idl}(J))$ are related. We prove that J and $\mathsf{Fil}(\mathsf{Idl}(J))$ are isomorphic in $\mathbf{JSL}_f$ hence the construction of taking the filter space of the set of ideals of J yields a lattice with the same order-structure as J. In a similar way we get that a meet-semilattice and the set of ideals of its filter space are isomorphic meet-semilattices.

In order to prove these isomorphisms we set up two families of maps η_M and ϵ_J, indexed by the objects of $\mathbf{MSL}_f^{op}$ and $\mathbf{JSL}_f$ respectively, as follows:

$$\epsilon_J : J \to \mathsf{Fil}(\mathsf{Idl}(J))$$
$$\eta_M : M \to \mathsf{Idl}(\mathsf{Fil}(M))$$

defined by

$$\epsilon_J(j) = \{I \in \mathsf{Idl}(J) \mid j \in I\}$$
$$\eta_M(m) = \{F \in \mathsf{Fil}(M) \mid m \in F\}$$

In terms of properties and denotations, ϵ_J and η_M are formalisations of the idea that "an element should be identified with the set of properties it satisfies" and "a property should be identified with the set of elements that satisfies that property". In our construction an element j satisfies the property represented by I if $j \in I$ and a property m is satisfied by an element represented by the filter F if $m \in F$.

The following technical result is of use in the proofs of the lemmas that follows.

Lemma 2.4.2 *Let M, N be meet-semilattices (join-semilattices) and let f be a function from M to N. Then f is an order-isomorphism if and only if f is a $\wedge$-homomorphism ($\vee$-homomorphism).*

Proof. The only difficult part is to see that order-isomorphism implies preservation

of $\wedge$. By monotonicity $f(m_1 \wedge m_2) \leq f(m_1) \wedge f(m_2)$, so, by applying f^{-1} to both sides of the inequality we get

$$m_1 \wedge m_2 \leq f^{-1}(f(m_1) \wedge f(m_2)) \leq f^{-1}(f(m_1)) \wedge f^{-1}(f(m_2)) = m_1 \wedge m_2$$

i.e.,

$$m_1 \wedge m_2 = f^{-1}(f(m_1) \wedge f(m_2))$$

and by applying f to both sides we get $f(m_1 \wedge m_2) = f(m_1) \wedge f(m_2)$. A similar proof shows the result about join-semilattices. ■

Lemma 2.4.3 *For every J an object in $\mathbf{JSL}_f$, ϵ_J is an $\vee$-isomorphism.*

Proof. We prove that ϵ_J is an order-isomorphism. First we observe that

$$\epsilon_J(j) = \{I \in \mathsf{Id|}(J) \mid j \in I\} = \{I \in \mathsf{Id|}(J) \mid \downarrow\{j\} \subseteq I\} = \uparrow(\downarrow\{j\}).$$

To show ϵ_J surjective, we notice that since J is finite so is the set of ideals $\mathsf{Id|}(J)$, hence any filter in $\mathsf{Id|}(J)$ is also finite. A finite filter contains the (finite!) meet of all its elements which is obviously the smallest element, so for any filter F of ideals there exists an ideal I such that $F = \uparrow(I)$. Since $I \subseteq J$ is also finite we get by a similar argument that I has a greatest element j, *i.e.*, $I = \downarrow\{j\}$. We have thus shown that

$$F = \uparrow(\downarrow\{j\}) = \epsilon_J(j) \qquad \text{for some } j \in J.$$

To show that ϵ_J is an order-embedding assume $j_1 \sqsubseteq j_2$. If $j_2 \in I$ for some ideal I then $j_1 \in I$ since I is downwards closed so

$$\epsilon_J(j_1) = \{I \in \mathsf{Id|}(J) \mid j_1 \in I\} \supseteq \{I \in \mathsf{Id|}(J) \mid j_2 \in I\} = \epsilon_J(j_2)$$

i.e., $\epsilon_J(j_1) \sqsubseteq \epsilon_J(j_1)$ when $\sqsubseteq$ is $\supseteq$. If, on the other hand, $\epsilon_J(j_1) \sqsubseteq \epsilon_J(j_1)$ then in particular $\downarrow\{j_2\} \in \{I \in \mathsf{Id|}(J) \mid j_1 \in I\}$ *i.e.*, $j_1 \in \downarrow\{j_2\}$ and $j_1 \sqsubseteq j_2$. By Lemma 2.4.2 we then conclude that ϵ_J is a $\vee$-homomorphism. ■

Lemma 2.4.4 *For every meet-semilattice M, η_M is a $\wedge$-isomorphism (and hence defines an $\mathbf{MSL}_f^{op}$ isomorphism from $\mathsf{Id|}(\mathsf{Fi|}(M))$ to M).*

Proof. By an argument similar to the one above we show that

$$\eta_M(m) = \downarrow\{\uparrow(m)\}$$

and that η_M is an order-isomorphism. Therefore, by Lemma 2.4.2, η_M is also a $\wedge$-homomorphism. ■

The two lemmas assure us that we do not change anything in the structure of the objects in $\mathbf{JSL}_f$ and $\mathbf{MSL}_f^{op}$ by applying Fil and Idl to them. It also means that any isomorphism class of objects in $\mathbf{JSL}$ will contain an object of the form $\mathsf{Fil}(M)$ where M is a meet-semilattice and any isomorphism class in $\mathbf{MSL}_f^{op}$ will have an object of form $\mathsf{Idl}(J)$.

Furthermore, since Fil and Idl are functors they map isomorphic objects to isomorphic objects. To prove that Fil and Idl will not map non-isomorphic objects to isomorphic objects we prove the following lemma:

Lemma 2.4.5 *η_M and ϵ_J are natural isomorphisms.*

Proof. There only remains to prove naturality of ϵ_J and η_M, *i.e.*, to prove that the following two diagrams commute:

$$\begin{array}{ccc} J & & J & \xrightarrow{\epsilon_J} & \mathsf{Fil}(\mathsf{Idl}(J)) \\ {\scriptstyle f}\downarrow & & {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathsf{Fil}(\mathsf{Idl}(f))} \\ K & & K & \xrightarrow{\epsilon_K} & \mathsf{Fil}(\mathsf{Idl}(K)) \end{array}$$

and

$$\begin{array}{ccc} M & & M & \xrightarrow{\eta_M} & \mathsf{Idl}(\mathsf{Fil}(M)) \\ {\scriptstyle f}\downarrow & & {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathsf{Idl}(\mathsf{Fil}(f))} \\ N & & N & \xrightarrow{\eta_N} & \mathsf{Idl}(\mathsf{Fil}(N)) \end{array}$$

We verify that the first diagram commutes. For $F \subseteq \mathsf{Idl}(J)$ a filter we have:

$$\mathsf{Fil}(\mathsf{Idl}(f))(F) = \{I \in \mathsf{Idl}(K) \mid \mathsf{Idl}(f)(I) \in F\} = \{I \in \mathsf{Idl}(K) \mid f^{-1}(I) \in F\}.$$

One way round in the diagram is

$$\begin{aligned} & \mathsf{Fil}(\mathsf{Idl}(f)) \circ \epsilon_J(j) \\ = \;& \mathsf{Fil}(\mathsf{Idl}(f))(\uparrow(\downarrow\{j\})) \\ = \;& \{I \in \mathsf{Idl}(K) \mid f^{-1}(I) \in \uparrow(\downarrow\{j\})\} \\ = \;& \{I \in \mathsf{Idl}(K) \mid \downarrow\{j\} \subseteq f^{-1}(I)\} \end{aligned}$$

so we have to prove

$$\{I \in \mathsf{Idl}(K) \mid {\downarrow}\{j\} \subseteq f^{-1}(I)\} = \epsilon_K \circ f(j) = {\uparrow}({\downarrow}\{f(j)\}).$$

If ${\downarrow}\{j\} \subseteq f^{-1}(I)$ then $j \in f^{-1}(I)$ so $f(j) \in I$ *i.e.*, ${\downarrow}\{f(j)\} \subseteq I$ implying $I \in {\uparrow}({\downarrow}\{f(j)\})$. Conversely, if ${\downarrow}\{f(j)\} \subseteq I$ then $f({\downarrow}\{j\}) \subseteq I$, *qua* monotonicity of f, which means that ${\downarrow}\{j\} \subseteq f^{-1}(f({\downarrow}\{j\})) \subseteq f^{-1}(I)$ so the two sets are equal.

Naturality of η is proved in a similar manner. ■

The three lemmas put together constitute a proof of the following theorem:

Theorem 2.4.6 $(\mathsf{Fil}, \mathsf{Idl}, \eta, \epsilon)$ *defines an equivalence between the categories* $\mathbf{MSL}_f^{op}$ *and* $\mathbf{JSL}_f$ *(and hence a duality between* $\mathbf{MSL}_f$ *and* $\mathbf{JSL}_f$*).*

The import of having an equivalence is that the categories $\mathbf{MSL}_f^{op}$ and $\mathbf{JSL}_f$ have the same isomorphism classes and that the functors Fil and Idl respect these isomorphism classes in the sense that two objects are isomorphic in $\mathbf{JSL}_f$ ($\mathbf{MSL}_f^{op}$) if and only if their images under Idl (Fil) are isomorphic [BW90][p. 69–70]. So two domains of denotations are isomorphic as ordered sets if and only if their corresponding set of properties are isomorphic as conjunctive systems. We shall apply the equivalence in the following way: to every type σ in a language we associate a conjunctive system, $\mathcal{L}(\sigma)$, of formulae and a join-semilattice $\sigma^{\#}$ and we interpret the formulae as properties over $\sigma^{\#}$ by the function

$$[\![\ ]\!]_\sigma : \mathcal{L}(\sigma) \to \mathsf{Idl}(\sigma^{\#}).$$

We prove that $[\![\ ]\!]_\sigma$ is surjective and an order-embedding. This implies that the quotient map

$$[\![\ ]\!]_\sigma /{=}: \mathcal{LA}(\sigma) \to \mathsf{Idl}(\sigma^{\#})$$

is an order-isomorphism. Applying the functor Fil to the isomorphism and using naturality of ϵ we obtain

$$\mathsf{Fil}(\mathcal{LA}(\sigma)) \cong \mathsf{Fil}(\mathsf{Idl}(\sigma^{\#})) \cong \sigma^{\#}$$

thus the set of filters of the logic associated with σ is isomorphic to the domain of abstract values used in the abstract interpretation.

2.5 Filter models and domain logic

The notion of conjunctive systems defined in this chapter is a variation of the notion of *information system* as introduced by Scott [Sco82] and later refined by Winskel and Larsen [LW84]. An information system (in the sense of the latter paper) is a triple $(T, Con, \vdash)$ where T is a set of tokens (basic pieces of information), Con is a predicate on finite sets of tokens that defines which pieces of information are consistent with each other and $\vdash$ is the entailment relation on tokens. The purpose of information systems is to provide a presentation of Domain Theory in which domains consist of elements that are sets of tokens of information: the pieces of information that the element represent. Such an element is a set where all finite subsets of tokens are consistent and the set itself is closed under entailment (*cf.* the notion of a point in Definition 2.2.1). The papers by Scott,Larsen and Winskel [Sco82] and [LW84] show how the usual domain constructors such as product, sum and function space together with the solution of recursive domain equations all can be given a presentation in terms of information systems.

The technique of constructing domains of denotations from a formal system of properties by taking filters as denotations has been used to construct a model of the untyped lambda calculus [BCD83]. The formal system of properties used in that paper is a system of *intersection types*, which in addition to type variables has a type constant ω (the universal type) and two type constructors, $\rightarrow$ and $\cap$. The intersection types are preorderd by an ordering similar to the $\leq$ from our conjunctive system and the filter space is shown to form a lambda model [Bar84] such that the interpretation of a lambda term (a filter) contains a type σ if and only if σ is a derivable type for that term in the intersection type discipline. The filter model was applied to prove the completeness of the Curry type system [Bar92] for untyped lambda calculus (*i.e.*, a type system without ω and $\cap$) with respect to the standard lambda model. We shall return to the intersection types in the next chapter.

An extensive study of the logical view of Domain Theory is the Domain Logic presented by Abramsky [Abr91b] in which the information system idea is extended to give a presentation of SFP domains [GS90] as logical theories. This work is based on a theory from topology known as Stone Duality which (for a certain kind of spaces) provides a means of reconstructing a topological space from its set of compact-open sets. Abramsky argues that the compact-open sets in the Scott topology on a domain can be identified with *observable properties*. The axiomatisation of the observable properties of a given type is then shown to give a presentation of the domain modelling that type. As here, the technique of passing from properties to denotations is to take denotations to be a certain kind of filters of properties. The main difference between the Domain Logic and our work is that the Domain

Logic works with open sets whereas we are axiomatising ideals which are closed in the Scott topology. We shall nevertheless base our axiomatisation on the one from Abramsky's domain logic [Abr91b] making changes where necessary.

2.6 Summary

In this chapter we have introduced the notion of a conjunctive system, a syntactic description of a given collection of properties plus their logical relationships. We showed how we can derive a domain of denotations from a logical system by taking filters of properties as elements of the domains thus identifying an element with the set of properties it satisfies. It was furthermore shown that the ideals of these domains could be considered as properties and that the set of ideals of a domain could be given the structure of a conjunctive system. This provides a way of passing back and forth between logic and denotation and it was shown that equivalent logics are taken to equivalent domains and *vice versa*. With this strong correspondence we now go on to relate a program logic built over these conjunctive systems with an abstract interpretation defined over the corresponding domains.

Chapter 3

Conjunctive program logics

In this chapter we study the relationship between an abstract interpretation and a program logic for the typed lambda calculus enriched with a fixed point operator and a conditional construct. The relationship is based on the duality between join- and meet-semilattices set up in the previous chapter. The abstract interpretation interprets expressions as elements in a join-semilattice. These elements can, via duality, be regarded as filters of properties from the dual meet-semilattice. We define a logic for reasoning about programs using these properties and prove that the denotation of an expression is exactly the filter of properties provable of the expression in the program logic. In other words, the abstract interpretation of an expression and the set of properties provable of this expression in the logic determine each other. Thus, with respect to finding properties of programs, the two analyses are equally powerful.

We shall establish the analogy for a higher order strictness analysis. The framework is, however, sufficiently general to carry over to other analyses. As an example we show how a minor modification to the analysis allows us to obtain a similar result for binding time analysis. We conclude the chapter by drawing some parallels to work on the use of intersection types in typing programs.

3.1 The typed lambda calculus

As programming language we shall be using the typed lambda calculus with various operators added. The typed lambda calculus is chosen because it is simple and yet allows us to study most of the problems associated with procedures and functions in "real" programming languages. We shall be using the simply typed lambda calculus

$$\textbf{Var}\quad x^\sigma : \sigma \qquad \textbf{Const}\quad c : \sigma_c$$

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

$$\textbf{Fst}\quad \frac{e : \sigma \times \tau}{\mathsf{fst}(e) : \sigma} \qquad \textbf{Snd}\quad \frac{e : \sigma \times \tau}{\mathsf{snd}(e) : \tau}$$

$$\textbf{Abs}\quad \frac{e : \tau}{\lambda x^\sigma.e : (\sigma \to \tau)} \qquad \textbf{App}\quad \frac{e_1 : (\sigma \to \tau) \quad e_2 : \sigma}{e_1\ e_2 : \tau}$$

$$\textbf{Fix}\quad \frac{e : \sigma \to \sigma}{\mathsf{fix}(e) : \sigma} \qquad \textbf{If}\quad \frac{b : \mathsf{Bool} \quad e_1 : \sigma \quad e_2 : \sigma}{\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \sigma}$$

Figure 3.1: Simply typed lambda calculus

with the two base types Bool for Boolean values and Int the type of integer values. There will be constants corresponding to the Boolean values true and false and a constant for each integer value. Furthermore there will be a collection of operators including a conditional and a fixpoint operator.

The set of types, T, is generated by the grammar

$$T = \mathsf{Bool} \mid \mathsf{Int} \mid T \times T \mid T \to T.$$

We shall use the letters σ and τ to range over types in T. In addition to the base type of Booleans and integers the types in T include the product types $\sigma \times \tau$ and the function type $\sigma \to \tau$.

We shall follow Barendregt [Bar92] in the presentation of typed lambda calculus and first define a set of *raw λ-terms*. They are given by the grammar below. It is assumed that for every type σ there is an infinite supply of variables $x^\sigma, y^\sigma, \ldots$ of type σ. We usually omit the type superscript on variables when we think it is clear from context.

$$\begin{aligned} e \;=\; & x^\sigma \mid c \mid (e_1, e_2) \mid \mathsf{fst}(e) \mid \mathsf{snd}(e) \mid \\ & \lambda x^\sigma.e \mid e_1 e_2 \mid \mathsf{fix}\ (e) \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3. \end{aligned}$$

In Figure 3.1 we present a type system in natural deduction style for proving typing judgements of the form $e : \sigma$ where e is a raw term and σ is a type. It is assumed that every constant c comes with a predefined type σ_c. The rule **Const** is used to express this fact in the type system.

From the set of raw terms we define the set of well typed terms, Λ_T, as the least set of closed terms that can be assigned a type. This set can be indexed by the types in T such that $\Lambda_T(\sigma)$ denotes the set of well typed terms of type σ.

3.2 Axiomatisation of conjunctive properties

In this section we use the theory developed in the previous chapter to describe an equivalence between an axiomatisation defining an entailment relation between strictness properties and the ideals of the domains used in higher order strictness analysis [BHA86]. We prove that these two structures, viewed as conjunctive systems, are isomorphic. This can then be translated to an isomorphism between the strictness domains and filters of strictness properties.

The strictness analysis by Burn, Hankin and Abramsky [BHA86] is formulated for the typed lambda calculus without product types. Types are interpreted as finite join-semilattices. Each base type is interpreted as the two-point lattice **2** and the function type as the lattice of monotone functions. We extend the interpretation of types as follows:

Definition 3.2.1 *For every σ belonging to T, the set of types, we define a join-semilattice $\sigma^\#$ inductively by:*

$$\begin{aligned} \mathsf{Bool}^\# &= \mathbf{2} \\ \mathsf{Int}^\# &= \mathbf{2} \\ (\sigma_1 \to \sigma_2)^\# &= \sigma_1^\# \to_m \sigma_2^\# \\ (\sigma_1 \times \sigma_2)^\# &= \sigma_1^\# \times \sigma_2^\#. \end{aligned}$$

where $\sigma_1^\# \times \sigma_2^\#$ is the Cartesian product of $\sigma_1^\#$ and $\sigma_2^\#$, ordered componentwise.

We next provide an axiomatic system of strictness formulae and prove that this axiomatic system, viewed as a conjunctive system, is isomorphic to the conjunctive system defined by the ideals of the $\sigma^\#$. For every type σ we define a logical theory

$$\mathcal{L}(\sigma) = (L(\sigma), \mathtt{t}_\sigma, \wedge_\sigma, \leq_\sigma, =_\sigma)$$

where $L(\sigma)$ is a set of formulae (denoting properties), the binary operator $\wedge_\sigma$ is conjunction, $\leq_\sigma$ corresponds to entailment and $=_\sigma$ is logical equivalence between formulae of that type.

The basic formulae for describing a property of an expression of type $\sigma \times \tau$ are of the form $\varphi \times \tau$ with $\varphi \in L(\sigma)$ and $\psi \in L(\tau)$. Here φ describes the first component and ψ the second component. The basic formulae for describing a function have form $\varphi \to \psi$ with the intention that a function satisfies this property if it maps arguments satisfying φ to results satisfying ψ. In the axiomatisation we introduce for each type a formula $\mathtt{f}_\sigma$ for denoting the strongest property, *i.e.*, the property that is the conjunction of all properties of type σ. The price we have to pay for introducing this redundant formula is that we have to add as extra axioms all facts provable about this formulae in the original system.

The sets of formulae are defined inductively as follows:

- $\mathtt{t}, \mathtt{f} \in L(\sigma)$
- $\dfrac{\varphi, \psi \in L(\sigma)}{\varphi \wedge \psi \in L(\sigma)}$
- $\dfrac{\varphi \in L(\sigma), \psi \in L(\tau)}{\varphi \times \psi \in L(\sigma \times \tau)}$
- $\dfrac{\varphi \in L(\sigma), \psi \in L(\tau)}{\varphi \to \psi \in L(\sigma \to \tau)}$

Among the formulae of the $L(\sigma)$'s there are formulae that are constructed with only restricted use of conjunction. We call these formulae *simple* formulae. The only place where we allow conjunctions to appear in simple formulae is in formulae for function spaces and then only in the sub-formula describing the argument. Simple formulae are useful because every strictness property can be expressed as a conjunction of simple formulae as proved in the Normal Form Theorem below. The set of simple formulae is characterised by the predicate Smpl defined by:

- $\mathsf{Smpl}(\mathtt{t}_\sigma)$
- $\mathsf{Smpl}(\mathtt{f}_\sigma)$
- $\dfrac{\mathsf{Smpl}(\varphi), \mathsf{Smpl}(\psi)}{\mathsf{Smpl}(\varphi \times \psi)}$
- $\dfrac{\mathsf{Smpl}(\psi)}{\mathsf{Smpl}(\varphi \to \psi)}$

The axioms and rules defining the logical structure of each $L(\sigma)$ are given in Figure 3.2. With this axiomatisation each $L(\sigma)$ is a conjunctive system.

In the previous chapter we argued that the Lindenbaum algebra could be regarded as the set of properties axiomatised by a conjunctive system. This algebra consists of equivalence classes where each class contains all formulae that are provably equal to each other. By introducing a normal form for the formulae of the system and showing that every formula is provably equal to a formula in normal form, we can always choose representatives in normal form.

- $\varphi \leq \varphi$
- $\dfrac{\varphi \leq \psi, \psi \leq \chi}{\varphi \leq \chi}$
- $\dfrac{\varphi \leq \psi_1, \varphi \leq \psi_2}{\varphi \leq \psi_1 \wedge \psi_2}$
- $\varphi \wedge \psi \leq \varphi$
- $\varphi \wedge \psi \leq \psi$
- $\dfrac{\varphi \leq \psi, \psi \leq \varphi}{\varphi = \psi}$
- $\dfrac{\varphi = \psi}{\varphi \leq \psi, \psi \leq \varphi}$
- $\varphi \leq \mathtt{t}$
- $\mathtt{f} \leq \varphi$
- $\mathtt{t}_{\sigma\times\tau} = \mathtt{t}_\sigma \times \mathtt{t}_\tau$
- $\mathtt{f}_{\sigma\times\tau} = \mathtt{f}_\sigma \times \mathtt{f}_\tau$
- $\mathtt{t}_{\sigma\to\tau} = \mathtt{t}_\sigma \to \mathtt{t}_\tau$
- $\mathtt{t}_\sigma \to \mathtt{f}_\tau = \mathtt{f}_{\sigma\to\tau}$
- $(\varphi_1 \times \psi_1) \wedge (\varphi_2 \times \psi_2) = (\varphi_1 \wedge \varphi_2) \times (\psi_1 \wedge \psi_2)$
- $\varphi \to \psi_1 \wedge \varphi \to \psi_2 = \varphi \to (\psi_1 \wedge \psi_2)$
- $\dfrac{\varphi_1 \leq \varphi_2, \psi_1 \leq \psi_2}{\varphi_1 \times \psi_1 \leq \varphi_2 \times \psi_2}$
- $\dfrac{\varphi' \leq \varphi, \psi \leq \psi'}{\varphi \to \psi \leq \varphi' \to \psi'}$

Figure 3.2: Axiomatisation of conjunctive properties

Theorem 3.2.2 (Normal Form) *For all $\varphi \in L(\sigma)$ there exists a finite set $\{\varphi_i\}_{i\in I} \subseteq L(\sigma)$, where $\mathsf{Smpl}(\varphi_i)$, such that*

$$\varphi = \bigwedge_{i\in I} \varphi_i.$$

Proof. By induction over the formation of φ. The formulae $\mathtt{t}$ and $\mathtt{f}$ are in normal form and if φ and ψ are in normal form so is $\varphi \wedge \psi$. We only have to verify that the formulae $\varphi \times \psi$ and $\varphi \to \psi$ can be converted into normal form. We can assume $\varphi = \bigwedge_{i\in I} \varphi_i$ and $\psi = \bigwedge_{j\in J} \psi_j$ with $\mathsf{Smpl}(\varphi_i)$ and $\mathsf{Smpl}(\psi_j)$. Then

$$\varphi \times \psi = \bigwedge_{i\in I} \varphi_i \times \bigwedge_{j\in J} \psi_j = \bigwedge_{i\in I} (\varphi_i \times \bigwedge_{j\in J} \psi_j) = \bigwedge_{i\in I} \bigwedge_{j\in J} \varphi_i \times \psi_j$$

and

$$\varphi \to \psi = \varphi \to \bigwedge_{j\in J} \psi_j = \bigwedge_{j\in J} (\varphi \to \psi_j)$$

demonstrate the relevant equivalences. ∎

3.2.1 Semantics of formulae

The connection between the logical system defining $\mathcal{L}(\sigma)$ and the collection of ideals of $\sigma^{\#}$ is established by means of semantic functions $[\![_]\!]_\sigma : \mathcal{L}(\sigma) \to \mathsf{Idl}(\sigma^{\#})$:

$$\begin{aligned}
[\![\mathtt{t}]\!]_\sigma &= \sigma^{\#} \\
[\![\mathtt{f}]\!]_\sigma &= \{0_{\sigma^{\#}}\} \\
[\![\textstyle\bigwedge_{i\in I}\psi_i]\!]_\sigma &= \textstyle\bigcap_{i\in I}[\![\psi_i]\!]_\sigma \\
[\![\varphi\to\psi]\!]_{\sigma\to\tau} &= \{f \in (\sigma\to\tau)^{\#} \mid f([\![\varphi]\!]_\sigma) \subseteq [\![\psi]\!]_\tau\} \\
[\![\varphi \times \psi]\!]_{\sigma\times\tau} &= \{(a,b) \mid a \in [\![\varphi]\!]_\sigma \text{ and } b \in [\![\psi]\!]_\tau\}
\end{aligned}$$

where $0_{\sigma^{\#}}$ denotes the least element in $\sigma^{\#}$.

We now set out to give a characterisation of the connection between the ideals in the domains and the formal system defined above. The Definability Theorem states that all ideals of a domain are indeed definable in the logical system in the sense that for all ideals there is a formula whose semantics is that ideal. In other words, $[\![_]\!]$ is surjective. The Soundness and Completeness Theorems assert that the axiomatisation is sound and complete with respect to the interpretation, *i.e.*, that all inclusions that hold between ideals in the domains can be proved in the logic (completeness) and no inclusions but those that hold can be proved (soundness).

In order to prove the theorems we need some results about how functions on lattices can be approximated by simpler objects. For these results it is important that for finite join-semilattices it is possible to define a greatest lower bound operator, *i.e.*, they are in effect complete lattices[1]. It is well known from Domain Theory [Plo81, Abr91b] that all continuous functions between two Scott-domains can be expressed using some simple functions called *step-functions*. These step functions are used to approximate functions from below. We shall be concerned with approximating functions from above, hence we study the dual to a step function:

Definition 3.2.3 *Let A, B be lattices with 1_B the top element in B and assume that $(a,b) \in A \times B$. The* co-step function $\lfloor (a,b) \rfloor : A \to B$ *is defined by*

$$\lfloor (a,b) \rfloor(x) = \begin{cases} b & \text{if } x \sqsubseteq a \\ 1_B & \text{otherwise} \end{cases}$$

[1]The greatest lower bound operator $\sqcap$ is defined by $a \sqcap b = \sqcup\{c \mid c \sqsubseteq a \text{ and } c \sqsubseteq b\}$.

We shall also use this notation for the greatest lower bound of a set of co-step functions. Let $u = \{(a_i, b_i) \mid i \in I\}$ be a set of pairs drawn from $A \times B$. The function $\lfloor u \rfloor$: A $\rightarrow$B induced by u is defined by

$$\lfloor u \rfloor(a) = \sqcap\{b_i \mid a \sqsubseteq a_i\}.$$

Any function is the greatest lower bound of the co-step functions greater than the function itself, so this gives a way of using co-step functions to build other functions. In fact we can be a bit more economical with the set of approximations due to the following lemma:

Lemma 3.2.4 *Let $f : A \rightarrow B$. Then*

$$f = \lfloor\{(a, f(a)) \mid a \in A\}\rfloor.$$

The usefulness of co-step functions is due to the fact that the principal ideal generated by the co-step function $\lfloor(a, b)\rfloor$ is the set of functions that map ideal $\downarrow\{a\}$ into the ideal $\downarrow\{b\}$. We generalise this fact in the following lemma:

Lemma 3.2.5 *Let $u = \{(a_i, b_i) \mid i \in I\}$. Then*

$$\downarrow\{\lfloor u \rfloor\} = \bigcap_{i \in I}\{g : A \rightarrow B \mid g(\downarrow\{a_i\}) \subseteq \downarrow\{b_i\}\}.$$

Proof.

$$\begin{aligned}
& f \sqsubseteq \lfloor u \rfloor \\
\Leftrightarrow\ & \forall a \in A\ .\ f(a) \sqsubseteq \sqcap\{b_i \mid a \sqsubseteq a_i\} \\
\Leftrightarrow\ & \forall i \in I\ \forall a \in A\ .\ a \sqsubseteq a_i \Rightarrow f(a) \sqsubseteq b_i \\
\Leftrightarrow\ & \forall i \in I\ .\ f \sqsubseteq \lfloor(a_i, b_i)\rfloor \\
\Leftrightarrow\ & f \in \cap_{i \in I}\downarrow\{\lfloor(a_i, b_i)\rfloor\} \\
\Leftrightarrow\ & f \in \cap_{i \in I}\{g : A \rightarrow B \mid g(\downarrow\{a_i\}) \subseteq \downarrow\{b_i\}\}
\end{aligned}$$

■

Thus, with φ_i, ψ_i and a_i, b_i such that $[\![\varphi_i]\!] = \downarrow\{a_i\}$ and $[\![\psi_i]\!] = \downarrow\{b_i\}$ we have that

$$[\![\bigwedge_i \varphi_i \rightarrow \psi_i]\!] = \bigcap_i\{g : A \rightarrow B \mid g(\downarrow\{a_i\}) \subseteq \downarrow\{b_i\}\} = \downarrow\{\lfloor\{(a_i, b_i)\}\rfloor\}.$$

The following theorem states that all ideals can be denoted by a formula from our formal system. In the formulation of the theorem we use the fact that all ideals are principal *i.e.*, downwards closures of a single element.

Theorem 3.2.6 (Definability) *For all elements $a \in \sigma^{\#}$ there exist a formula $\varphi_a \in L(\sigma)$ such that*

$$\downarrow\{a\} = [\![\varphi_a]\!]_\sigma.$$

Proof. The proof works by structural induction over the type σ. For the base type **2** it is straightforward to see that the two elements 0 and 1 correspond to the formulae $\mathtt{f}$ and $\mathtt{t}$, respectively. Now, let $f \in (A \to B)$. Since functions are ordered pointwise we have:

$$\downarrow\{f\} = \bigcap_{a \in A} \{g : A \to B \mid g(\downarrow\{a\}) \subseteq \downarrow\{f(a)\}\} = [\![\bigwedge_{a \in A} \varphi_a \to \varphi_{f(a)}]\!]$$

where φ_a and $\varphi_{f(a)}$ are the formula corresponding to $\downarrow\{a\}$ and $\downarrow\{f(a)\}$, given by the induction hypothesis. For the product type we have that an ideal in $(A \times B)$ is of the form $\downarrow\{(a_0, b_0)\}$. With φ, ψ such that $[\![\varphi]\!] = \downarrow\{a_0\}$ and $[\![\psi]\!] = \downarrow\{b_0\}$ we have

$$\downarrow\{(a,b)\} = \{(a,b) \mid a \in \downarrow\{a_0\} \text{ and } b \in \downarrow\{b_0\}\} = [\![\varphi \times \psi]\!].$$

■

Theorem 3.2.7 (Soundness) *For all elements $\varphi, \psi \in \mathcal{L}(\sigma)$:*

$$\varphi \leq \psi \quad \Rightarrow \quad [\![\varphi]\!]_\sigma \subseteq [\![\psi]\!]_\sigma.$$

Proof. The proof amounts to checking that each axiom and rule is sound. The theorem then follows by induction over the derivation of $\varphi \leq \psi$. Most of the rules are proved sound by simple set-theoretic reasoning. For example, $\varphi \wedge \psi \leq \varphi$ is sound since for all sets A, B we have $A \cap B \subseteq A$. We verify a few here:

- $[\![\mathtt{f}_{\sigma \to \tau}]\!] = \{0_{\sigma \to \tau}\} = \{\lambda x.0_\tau\} = \{f \mid f(\sigma^{\#}) \subseteq \{0_\tau\}\} = [\![\mathtt{t}_\sigma \to \mathtt{f}_\tau]\!]$.
- $[\![\varphi \to (\psi_1 \wedge \psi_2)]\!] = \{f \mid f([\![\varphi]\!]) \subseteq [\![\psi_1]\!] \cap [\![\psi_2]\!]\} = \{f \mid f([\![\varphi]\!]) \subseteq [\![\psi_1]\!]\} \cap \{f \mid f([\![\varphi]\!]) \subseteq [\![\psi_2]\!]\} = [\![\varphi \to \psi_1]\!] \wedge [\![\varphi \to \psi_2]\!]$.
- $[\![(\varphi_1 \wedge \varphi_2) \times (\psi_1 \wedge \psi_2)]\!] = \{(a,b) \mid a \in [\![\varphi_1]\!] \cap [\![\varphi_2]\!] \text{ and } b \in [\![\psi_1 \wedge \psi_2]\!]\} = \{(a,b) \mid a \in [\![\varphi_1]\!] \text{ and } b \in [\![\psi_1]\!]\} \cap \{(a,b) \mid a \in [\![\varphi_2]\!] \text{ and } b \in [\![\psi_2]\!]\} = [\![(\varphi_1 \times \psi) \wedge (\varphi_2 \times \psi)]\!]$.

■

Theorem 3.2.8 (Completeness) *For all formulae* $\varphi, \psi \in L(\sigma)$*:*

$$[\![\varphi]\!]_\sigma \subseteq [\![\psi]\!]_\sigma \quad \Rightarrow \quad \varphi \leq \psi.$$

Proof. Induction over the structure of σ. For the base case **2** we can easily prove the only non-trivial inclusion $[\![\mathtt{f}]\!] \subseteq [\![\mathtt{t}]\!]$ using either the rule $\varphi \leq \mathtt{t}$ or $\mathtt{f} \leq \varphi$. For the inductive step we can assume that all formulae are in normal form. Let $\bigwedge_{i\in I}(\varphi_i{\to}\psi_i)$ and $\bigwedge_{j\in J}(\varphi_j{\to}\psi_j)$ be two formulae in $L(\sigma{\to}\tau)$ and let a_n, b_n for $n \in I{\cup}J$ be such that $\downarrow\{a_n\} = [\![\varphi_n]\!]$ and $\downarrow\{b_n\} = [\![\psi_n]\!]$. If all $\psi_j = \mathtt{t}_\tau$ then $\bigwedge_{j\in J}(\varphi_j{\to}\psi_j) = \bigwedge_{j\in J} \mathtt{t}_{\sigma\to\tau} = \mathtt{t}_{\sigma\to\tau}$ and so is trivially entailed by $\bigwedge_{i\in I}(\varphi_i{\to}\psi_i)$. In fact, we can assume without loss of generality that all $\psi_j \neq \mathtt{t}_\tau$ since $\varphi_j{\to}\mathtt{t}_\tau = \mathtt{t}_{\sigma\to\tau}$ and so can always be added to a conjunction by the rule $\varphi \wedge \mathtt{t} = \varphi$. For the b_j this means that $b_j \neq 1_{\tau\#}$ for all $j \in J$. Then

$$\begin{aligned}
&[\![\bigwedge_{i\in I}(\varphi_i{\to}\psi_i)]\!] \subseteq [\![\bigwedge_{j\in J}(\varphi_j{\to}\psi_j)]\!] \\
\Rightarrow\ & \forall j \,.\, \textstyle\bigcap_{i\in I}[\![\varphi_i{\to}\psi_i]\!] \subseteq [\![\varphi_j{\to}\psi_j]\!] \\
\Rightarrow\ & \forall j \,.\, \textstyle\bigcap_{i\in I}\downarrow\{\lfloor(a_i, b_i)\rfloor\} \subseteq \downarrow\{\lfloor(a_j, b_j)\rfloor\} \\
\Rightarrow\ & \forall j \,.\, \downarrow\{\lfloor\{(a_i, b_i) \mid i \in I\}\rfloor\} \subseteq \downarrow\{\lfloor(a_j, b_j)\rfloor\} \\
\Rightarrow\ & \forall j \,.\, \lfloor\{(a_i, b_i) \mid i \in I\}\rfloor \sqsubseteq \lfloor(a_j, b_j)\rfloor
\end{aligned}$$

We want to show that $\bigwedge_{i\in I}(\varphi_i{\to}\psi_i)$ entails $\bigwedge_{j\in J}(\varphi_j{\to}\psi_j)$. We do this by showing that $\bigwedge_{i\in I}(\varphi_i{\to}\psi_i)$ entails $\varphi_j{\to}\psi_j$ for all $j \in J$. Fix a j. From the definition of co-step function it follows that

$$I_j \equiv \{a_i \mid a_j \sqsubseteq a_i\} \neq \emptyset \qquad \text{and} \qquad \bigsqcap_{i\in I_j} b_i \sqsubseteq b_j.$$

which means that

$$\forall i \in I_j \,.\, [\![\varphi_j]\!] \subseteq [\![\varphi_i]\!] \qquad \text{and} \qquad [\![\bigwedge_{i\in I_j} \psi_i]\!] = \bigcap_{i\in I_j} \downarrow\{b_i\} = \downarrow\{\bigsqcap_{i\in I_j} b_i\} \subseteq \downarrow\{b_j\} = [\![\psi_j]\!].$$

Applying the induction hypothesis we get that

$$\forall i \in I_j \,.\, \varphi_j \leq \varphi_i \qquad \text{and} \qquad \bigwedge_{i\in I_j} \psi_i \leq \psi_j$$

This verifies the following entailments

$$\bigwedge_{i\in I}(\varphi_i{\to}\psi_i) \leq \bigwedge_{i\in I_j}(\varphi_i{\to}\psi_i) \leq \bigwedge_{i\in I_j}(\varphi_j{\to}\psi_i) = \varphi_j{\to}\bigwedge_{i\in I_j}\psi_i \leq \varphi_j{\to}\psi_j.$$

Since j is arbitrary this holds for all $j \in J$ and we can deduce that

$$\bigwedge_{i\in I}(\varphi_i \to \psi_i) \leq \bigwedge_{j\in J}(\varphi_j \to \psi_j)$$

which proves completeness for the function space. For product types we have

$$\begin{aligned}
&[\![\textstyle\bigwedge(\varphi_i \times \psi_i)]\!] \subseteq [\![\bigwedge(\varphi_j \times \psi_j)]\!] \\
\Rightarrow\ & \textstyle\bigcap [\![\varphi_i \times \psi_i]\!] \subseteq \bigcap [\![\varphi_j \times \psi_j]\!] \\
\Rightarrow\ & \forall j\ \exists i_1, i_2\ .\ [\![\varphi_{i_1}]\!] \subseteq [\![\varphi_j]\!] \text{ and } [\![\psi_{i_2}]\!] \subseteq [\![\psi_j]\!] \\
\Rightarrow\ & \forall j\ \exists i_1, i_2\ .\ \varphi_{i_1} \leq \varphi_j \text{ and } \psi_{i_2} \leq \psi_j \qquad \text{by induction hypothesis} \\
\Rightarrow\ & \forall j\ \exists i_1, i_2\ .\ (\varphi_{i_1} \times \psi_{i_1}) \wedge (\varphi_{i_2} \times \psi_{i_2}) \leq \varphi_j \times \psi_j \\
\Rightarrow\ & \textstyle\bigwedge(\varphi_i \times \psi_i) \leq \bigwedge(\varphi_j \times \psi_j)
\end{aligned}$$

■

As already noticed, the Definability Theorem shows that $[\![\,]\!]$ is surjective on $\mathsf{Idl}(\sigma^\#)$. From the Soundness and Completeness Theorems for the axiomatisation we get that $[\![\,]\!]$ is an order-embedding of the pre-order $\mathcal{L}(\sigma)$ into $\mathsf{Idl}(\sigma^\#)$. By Lemma 2.1.3 we then conclude that

$$\mathsf{Idl}(\sigma^\#) \cong \mathcal{L}(\sigma)$$

as conjunctive systems. By the isomorphism proved in Theorem 2.4.6 we can conclude that for every type σ the map ϵ_σ defined by

$$\epsilon_\sigma(d) = \{[\varphi] \mid d \in [\![\varphi]\!]\}$$

defines an isomorphism of join-semilattices

$$\sigma^\# \cong \mathsf{Fil}(\mathsf{Idl}(\sigma^\#)) \cong \mathsf{Fil}(\mathcal{LA}(\sigma)).$$

This gives a precise connection between the elements of the domains in strictness analysis and the formulae underlying the strictness logic. The domains used in the abstract interpretation and the filter space of the conjunctive system defining the strictness properties are isomorphic join-semilattices. An abstract interpretation that assigns elements of the domains as denotations of programs can via this isomorphism be seen as assigning filters of properties to programs. We shall now prove that the properties in the filter assigned to a program are exactly the properties provable of the program in the logic.

3.3 Strictness analysis

In the previous section we laid the foundations for proving the equivalence of two strictness analyses (an abstract interpretation and a program logic), by providing a correspondence between elements of strictness domains and sets of strictness properties. This section completes the proof of the equivalence by showing that every deduction has its counterpart in the denotational semantics defining the abstract interpretation and that any property exhibited by the abstract interpretation of a program can be deduced in the logic. We first review strictness analysis for higher order languages by abstract interpretation and define a program logic for strictness analysis for the same language. We then prove this logic sound and complete with respect to the abstract interpretation.

3.3.1 Abstract interpretation

Strictness analysis for the typed lambda calculus by abstract interpretation works by interpreting the terms of Λ_T over non-standard domains built up from the two-point domain $\mathbf{2}$ interpreting the type constructors $\times$ as cartesian product of lattices and $\rightarrow$ as constructing the lattice of continuous functions between two lattices. The abstract interpretation of a term e of type $\mathsf{Int} \rightarrow \mathsf{Int}$ would then be a function $[\![e]\!] : \mathbf{2} \rightarrow \mathbf{2}$. The function $[\![]\!]$ takes as arguments a term in Λ_T and an environment mapping variables to values and returns as result the abstract value of the term. It is defined in Figure 3.3. Built-in operations like + must be defined specifically. For example $[\![+]\!] : \mathsf{Int}^{\#} \times \mathsf{Int}^{\#} \rightarrow \mathsf{Int}^{\#}$ can be defined by $[\![+]\!](a, b) = a \sqcap b$.

3.3.2 Strictness logic

We now define a program logic for the simply typed λ-calculus based on the formal system of strictness properties. It is shown that the program logic is equivalent to the abstract interpretation presented above in the sense that all strictness properties that hold of an abstract interpretation of a function can be proved in the strictness logic.

The strictness logic operates on judgements of the form

$$\Gamma \vdash e : \varphi$$

where Γ is an environment that associates variables $x_i^{\sigma_i}$ of type σ_i with formulae from $L(\sigma_i)$, e is a well typed term (of type σ, say) and φ is a formula from $L(\sigma)$.

$$
\begin{array}{lcl}
[\![x]\!]\rho & = & \rho(x) \\
[\![c]\!]\rho & = & 1 \\
[\![(e_1, e_2)]\!]\rho & = & ([\![e_1]\!]\rho, [\![e_2]\!]\rho) \\
[\![\mathsf{fst}(e)]\!]\rho & = & \pi_1([\![e]\!]\rho) \\
[\![\mathsf{snd}(e)]\!]\rho & = & \pi_2([\![e]\!]\rho) \\
[\![\lambda x.e]\!]\rho & = & \lambda v.[\![e]\!]\rho[x \mapsto v] \\
[\![e_1 e_2]\!]\rho & = & [\![e_1]\!]\rho([\![e_2]\!]\rho) \\
[\![\mathsf{fix}(e)]\!]\rho & = & \bigsqcup_{n\in\omega} ([\![e]\!]\rho)^n(\bot) \\
[\![\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\rho & = & \begin{cases} 0 & \text{if } [\![b]\!]\rho = 0_{\mathsf{Bool}} \\ [\![e_1]\!]\rho \sqcup [\![e_2]\!]\rho & \text{otherwise} \end{cases}
\end{array}
$$

Figure 3.3: Abstract interpretation for strictness analysis

The logic is presented in a natural deduction style reminiscent of the type system in Figure 3.1. Note that while a variable has a unique type it can be associated with different formulae. Hence we need to introduce an environment to keep track of what assumptions we have made about the variables. The logic is defined by the rules in Figure 3.4.

Other constants (including built-in functions) can be treated by adding more rules. The curried version of the addition function, which is strict in both arguments, is defined by the rule

$$\vdash + : \mathtt{f}\to\mathtt{t}\to\mathtt{f} \wedge \mathtt{t}\to\mathtt{f}\to\mathtt{f}.$$

The following proof demonstrates the use of the proof system. The function we analyse is taken from Kuo and Mishra's paper [KM89], where it was used to demonstrate the limitations of a type system without conjunctive types. It is defined as:

$$f\ x\ y\ z = \mathit{if}\ (z = 0)\ \mathit{then}\ x + y\ \mathit{else}\ f\ y\ x\ (z - 1)$$

Written in our language it looks like $\mathsf{fix}\lambda f.E$ where E is defined by

$$E \equiv \lambda x.\lambda y.\lambda z.\mathsf{if}\ z = 0\ \mathsf{then}\ x + y\ \mathsf{else}\ f\ y\ x\ (z - 1).$$

$$\textbf{Conj}\quad \frac{\Gamma \vdash e : \psi_1 \quad \Gamma \vdash E : \psi_2}{\Gamma \vdash e : \psi_1 \wedge \psi_2} \qquad \textbf{Weak}\quad \frac{\Gamma \leq \Delta \quad \Delta \vdash e : \phi \quad \phi \leq \psi}{\Gamma \vdash e : \psi}$$

$$\textbf{Taut}\quad \Gamma \vdash e : \mathtt{t} \qquad \textbf{Var}\quad \Gamma[x \mapsto \phi] \vdash x : \phi$$

$$\textbf{Fst}\quad \vdash \mathsf{fst} : (\varphi \times \phi) \to \varphi \qquad \textbf{Snd}\quad \vdash \mathsf{snd} : (\varphi \times \phi) \to \phi$$

$$\textbf{Pair}\quad \frac{\Gamma \vdash e_1 : \phi_1 \quad \Gamma \vdash e_2 : \phi_2}{\Gamma \vdash (e_1, e_2) : \phi_1 \times \phi_2}$$

$$\textbf{Abs}\quad \frac{\Gamma[x \mapsto \phi] \vdash e : \psi}{\Gamma \vdash \lambda x.e : (\phi \to \psi)} \qquad \textbf{App}\quad \frac{\Gamma \vdash e_1 : (\phi \to \psi) \quad \Gamma \vdash e_2 : \phi}{\Gamma \vdash e_1 e_2 : \psi}$$

$$\textbf{Fix}\quad \frac{\Gamma \vdash e : \phi \to \phi}{\Gamma \vdash \mathsf{fix}(e) : \phi}$$

$$\textbf{If-1}\quad \frac{\Gamma \vdash b : \mathtt{f}}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \mathtt{f}} \qquad \textbf{If-2}\quad \frac{\Gamma \vdash e_1 : \phi \quad \Gamma \vdash e_2 : \phi}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \phi}$$

Figure 3.4: Conjunctive strictness logic for simply typed lambda calculus

We want to show that this function is strict in x and y separately, which in our logic is expressed by the formula

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

For notational convenience denote this formula by ψ and let similarly ψ_1 and ψ_2 denote $\mathtt{f}{\to}\mathtt{t}{\to}\mathtt{t}{\to}\mathtt{f}$ and $\mathtt{t}{\to}\mathtt{f}{\to}\mathtt{t}{\to}\mathtt{f}$, respectively. Furthermore, let Γ be the environment $[f : \psi, x : \mathtt{t}, y : \mathtt{f}, z : \mathtt{t}]$. We then get the proof tree in Figure 3.5.

3.3.3 Soundness and Completeness Theorems

Before stating the main theorem that relates the program logic to the abstract interpretation we introduce some notation for semantic validity. Let σ be a type and e a term of type σ. Assume that $d \in \sigma^{\#}$, $\varphi \in L(\sigma)$ and let ρ and Γ be environments mapping variables to values and formulae respectively.

We write:

$$\begin{aligned} d \vDash \varphi &\equiv d \in [\![\varphi]\!] \\ \rho \vDash \Gamma &\equiv \forall x : \rho(x) \vDash \Gamma(x) \\ \Gamma \vDash e : \varphi &\equiv \forall \rho \vDash \Gamma : [\![e]\!]\rho \vDash \varphi \end{aligned}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash f : \psi}{\Gamma \vdash x : \mathtt{t} \quad \Gamma \vdash y : \mathtt{f}} \quad \dfrac{\Gamma \vdash f : \psi}{\Gamma \vdash f : \psi_1 \quad \Gamma \vdash y : \mathtt{f} \quad \Gamma \vdash x : \mathtt{t} \quad \Gamma \vdash (z-1) : \mathtt{t}}}{\Gamma \vdash x + y : \mathtt{f}} \quad \Gamma \vdash f\ y\ x\ (z-1) : \mathtt{f}}{\Gamma \vdash \mathsf{if}\ z = 0\ \mathsf{then}\ x + y\ \mathsf{else}\ f\ y\ x\ (z-1)) : \mathtt{f}}\ \mathbf{If-2}}{[f : \psi] \vdash E : \psi_1 \quad [f : \psi] \vdash E : \psi_2}\ \mathbf{Abs}}{[f : \psi] \vdash \lambda x.\lambda y.\lambda z.\mathsf{if}\ z = 0\ \mathsf{then}\ x + y\ \mathsf{else}\ f\ y\ x\ (z-1) : \psi}\ \mathbf{Conj}}{\vdash \lambda f.\lambda x.\lambda y.\lambda z.\mathsf{if}\ z = 0\ \mathsf{then}\ x + y\ \mathsf{else}\ f\ y\ x\ (z-1) : \psi \to \psi}\ \mathbf{Abs}}{\vdash \mathsf{fix}(\lambda f.\lambda x.\lambda y.\lambda z.\mathsf{if}\ z = 0\ \mathsf{then}\ x + y\ \mathsf{else}\ f\ y\ x\ (z-1)) : \psi}\ \mathbf{Fix}$$

Figure 3.5: An example derivation in the conjunctive logic

i.e., we overload the symbol $\models$ to mean *i*) a value satisfies a formula, *ii*) an environment of values satisfies an environment of formulae and *iii*) the denotation of a term e satisfies formula φ when e is evaluated in an environment satisfying environment Γ.

This is the standard notion of semantic validity. Following Abramsky [Abr91b] we refine this notion in order to simplify the Completeness proof of the logic. Since $[\![\varphi]\!]$ always denotes a principal ideal of the form $\downarrow\{a\}$ we get a relation, $\leftrightsquigarrow$, between formulae and elements defined by

$$a \leftrightsquigarrow \varphi \equiv \downarrow\{a\} = [\![\varphi]\!]. \tag{3.1}$$

For every environment Γ of formulae there exists an environment of values, ρ_Γ, defined by $\rho_\Gamma(x) \leftrightsquigarrow \Gamma(x)$. So $d \models \varphi$ if and only if $d \sqsubseteq a$, where a is the unique a such that $a \leftrightsquigarrow \varphi$, and $\rho \models \Gamma$ if and only if $\rho \sqsubseteq \rho_\Gamma$. The import of this is that we only have to consider one environment when proving semantic validity of a judgement in the logic.

Lemma 3.3.1 $\Gamma \models e : \varphi \ \Leftrightarrow\ [\![e]\!]\rho_\Gamma \in [\![\varphi]\!]$.

Proof. $\Rightarrow$ is clear since $\rho_\Gamma \models \Gamma$. To prove $\Leftarrow$ we notice that monotonicity of $[\![e]\!]$ in the environment means that if $[\![e]\!]\rho_\Gamma \in [\![\varphi]\!]$ and $\rho \sqsubseteq \rho_\Gamma$ then $[\![e]\!]\rho \in [\![\varphi]\!]$. ■

The main theorem linking the abstract interpretation with the strictness logic can be stated as follows:

Theorem 3.3.2 (Soundness & Completeness of the program logic) *Let e be a term of type σ and assume that Γ is an environment mapping the free variables of e to propositions. For φ a formula belonging to $L(\sigma)$ we have:*

$$\Gamma \vdash e : \varphi \Leftrightarrow \Gamma \vDash e : \varphi$$

Proof. The soundness (the "$\Rightarrow$" in the bi-implication) is proved by induction on the length of the proof. The completeness (the "$\Leftarrow$") of the logic is proved by structural induction on the term e.

Soundness. For the soundness it suffices to prove that if the premises of a given rule are valid in the semantics then the conclusion is valid too. Induction then gives us that all provable facts are indeed semantically valid. We treat a few of the rules here:

Abstraction. $\Gamma[x : \varphi] \vDash e : \psi$

$\Rightarrow \quad \forall \rho \vDash \Gamma \,.\, \forall d \vDash \varphi \,.\, [\![e]\!]\rho[x \mapsto d] \in [\![\psi]\!]$

$\Rightarrow \quad \forall \rho \vDash \Gamma \,.\, \forall d \in [\![\varphi]\!] \,.\, [\![\lambda x.e]\!]\rho(d) = [\![e]\!]\rho[x \mapsto d] \in [\![\psi]\!]$

$\Rightarrow \quad \Gamma \vDash \lambda x.e : \varphi \rightarrow \psi$

Fixed points. $\Gamma \vDash e : \psi \rightarrow \psi$

$\Rightarrow \quad \forall \rho \vDash \Gamma . \forall i \in \omega . [\![e]\!]^i \rho(\bot) \in [\![\psi]\!]$ since $\bot \in [\![\psi]\!]$

$\Rightarrow \quad \forall \rho \vDash \Gamma \,.\, [\![\mathsf{fix}(e)]\!]\rho \in [\![\psi]\!]$ since $[\![\psi]\!]$ is closed under $\sqcup$

$\Rightarrow \quad \Gamma \vDash \mathsf{fix}(e) : \psi$

If-1. $\Gamma \vDash e_1 : \mathtt{f}$

$\Rightarrow \quad \forall \rho \vDash \Gamma \,.\, [\![e_1]\!]\rho = \bot$

$\Rightarrow \quad \forall \rho \vDash \Gamma \,.\, [\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!]\rho = \bot$

$\Rightarrow \quad \Gamma \vDash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \mathtt{f}$

Completeness Completeness amounts to showing (by structural induction over the terms) that all facts that hold in the semantics can be proved in the logic.

Application. $\Gamma \vDash e_1 e_2 : \psi$

$\Leftrightarrow$ $[\![e_1 e_2]\!]\rho_\Gamma = [\![e_1]\!]\rho_\Gamma([\![e_1]\!]\rho_\Gamma) \in [\![\psi]\!]$ by Lemma 3.3.1

$\Leftrightarrow$ $d \in \downarrow\{[\![e_2]\!]\rho\} \Rightarrow [\![e_1]\!]\rho_\Gamma(d) \in [\![\psi]\!]$

With φ such that $[\![e_2]\!]\rho_\Gamma \leftrightsquigarrow \varphi$ we then have

$[\![e_2]\!]\rho_\Gamma \in [\![\varphi]\!]$ and $[\![e_1]\!]\rho_\Gamma \in [\![\varphi{\to}\psi]\!]$

$\Rightarrow$ $\Gamma \vdash e_1 : \varphi{\to}\psi$ and $\Gamma \vdash e_2 : \varphi$ by induction hypothesis

$\Rightarrow$ $\Gamma \vdash e_1 e_2 : \psi$

Abstraction. $\Gamma \vDash \lambda x.e : \varphi{\to}\psi$

$\Leftrightarrow$ $[\![\lambda x.e]\!]\rho_\Gamma \in [\![\varphi{\to}\psi]\!]$

$\Leftrightarrow$ $[\![\lambda x.e]\!]\rho_\Gamma([\![\varphi]\!]) \subseteq [\![\psi]\!]$

With $d \leftrightsquigarrow \varphi$ this implies that

$[\![e]\!]\rho_\Gamma[x \mapsto d] \in [\![\psi]\!]$

$\Rightarrow$ $\Gamma[x : \varphi] \vDash e : \psi$

$\Rightarrow$ $\Gamma[x : \varphi] \vdash e : \psi$ by induction hypothesis

$\Rightarrow$ $\Gamma \vdash \lambda x.e : \varphi{\to}\psi$

Fixed points. $\Gamma \vDash \mathsf{fix}(e) : \psi \Rightarrow [\![\mathsf{fix}(e)]\!]\rho_\Gamma \in [\![\psi]\!]$

Let φ be such that $[\![\varphi]\!] = \downarrow\{[\![\mathsf{fix}(e)]\!]\rho\}$. As $[\![\varphi]\!] \subseteq [\![\psi]\!]$ we have from Theorem 3.2.8 that $\varphi \leq \psi$, so we can deduce as follows:

$[\![e]\!]\rho_\Gamma([\![\mathsf{fix}(e)]\!]\rho_\Gamma) = [\![\mathsf{fix}(e)]\!]\rho_\Gamma$

$\Rightarrow$ $\Gamma \vDash e : \varphi{\to}\varphi$

$\Rightarrow$ $\Gamma \vdash e : \varphi{\to}\varphi$ by induction hypothesis

$\Rightarrow$ $\Gamma \vdash \mathsf{fix}(e) : \varphi$

$\Rightarrow$ $\Gamma \vdash \mathsf{fix}(e) : \psi$ by the rule **Weak**

Conditional $\Gamma \vDash \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \psi$

$\Leftrightarrow$ $[\![\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\rho_\Gamma \in [\![\psi]\!]$

There are now two possibilities: Either

$[\![b]\!]\rho_\Gamma = 0_{\mathsf{Bool}}$

$\Rightarrow$ $\Gamma \vdash b : \mathtt{f}$ by induction hypothesis

$\Rightarrow$ $\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \mathtt{f} \leq \psi$

or

$$\Rightarrow \quad [\![e_1]\!]\rho_\Gamma \sqcup [\![e_2]\!]\rho_\Gamma \in [\![\psi]\!]$$

$$\Rightarrow \quad [\![e_1]\!]\rho_\Gamma \in [\![\psi]\!] \quad \text{and} \quad [\![e_2]\!]\rho_\Gamma \in [\![\psi]\!]$$

$$\Rightarrow \quad \Gamma \vdash e_1 : \psi \quad \text{and} \quad \Gamma \vdash e_2 : \psi \quad \text{by induction hypothesis}$$

$$\Rightarrow \quad \Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \psi$$

■

By combining this theorem with the isomorphism between semantic domains and filters of formulae $\epsilon_\sigma : \sigma^\# \to \mathsf{Fil}(\mathcal{LA}(\sigma))$ we can give a precise characterisation of the correspondence between the abstract interpretation and the strictness logic. We have that for an expression e of type σ and environment of formulae Γ.

$$\epsilon_\sigma([\![e]\!]\rho_\Gamma) = \{[\varphi] \mid [\![e]\!]\rho_\Gamma \in [\![\varphi]\!]\} = \{[\varphi] \mid \Gamma \vdash e : \varphi\}$$

i.e., that, via the isomorphism ϵ_σ, the denotation of an expression given by the abstract interpretation is determined by and determines the set of formulae provable of e in the strictness logic.

3.4 Binding time analysis

So far the comparison between abstract interpretation and program logic has been done in the setting of strictness analysis. It should however be clear that strictness did not play a prominent role in the developments. With this section we want to demonstrate that the technique for relating the two kinds of analyses is not tied to strictness analysis but can, with minor changes, be adapted to other analyses. We shall do so by showing how the program logic for strictness analysis can be modified to give a binding time analysis corresponding to the abstract interpretation for binding time analysis presented by Hunt and Sands [HS91].

Binding time analysis aims at determining which part of a program can be evaluated given only partial knowledge about the input to the program. This information is essential for a *partial evaluator* in order to determine which part of the program can be evaluated during partial evaluation. The two basic properties that can be assigned to an expression are *static*, meaning that the value of the expression can be determined, and *dynamic*, meaning that there might not be enough information to compute the value of the expression. The abstract interpretation by Hunt and

Sands [HS91] uses as abstract domain the two element domain

$$\begin{array}{ll} D & \text{Dynamic} \\ | & \\ S & \text{Static} \end{array}$$

to model base types Int and Bool, *i.e.*, the same domain as in strictness analysis but with different interpretation of the elements. The type constructors $\times$ and $\rightarrow$ are interpreted exactly as in strictness analysis (*i.e.*, cartesian product and monotone function space) so the abstract domains are all order-isomorphic. The analysis by Hunt and Sands also includes list types but we shall disregard this for the moment. Lists are treated in Chapter 6.

The binding time interpretation of terms is equal to the strictness interpretation except for the if-expression (called *cond* in *loc. cit.*) and for constants. The interpretation of if in binding time analysis is

$$[\![\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\rho = \begin{cases} D & \text{if } [\![b]\!]\rho = D_{\mathsf{Bool}} \\ [\![e_1]\!]\rho \sqcup [\![e_2]\!]\rho & \text{if } [\![b]\!]\rho = S_{\mathsf{Bool}} \end{cases} \tag{3.2}$$

which should be understood as follows. In order to determine the value of a conditional the branching condition b must be known (S). In that case we see how much we can determine of the values of the branches and combine these results by taking the least upper bound. The interpretation of constants is S, *i.e.*, the smallest element in the domain **2** as opposed to strictness analysis where constants are interpreted by the top element. Apart from this the abstract interpretations are identical.

The similarity in the abstract interpretations is reflected in the program logics. The only amendment that has to be made to the strictness logic in Figure 3.4 is to replace the rules **If-1** and **If-2** with the following rule

$$(\textbf{If-BTA}) \quad \frac{\Gamma \vdash b : \mathtt{f} \quad \Gamma \vdash e_1 : \varphi \quad \Gamma \vdash e_2 : \varphi}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \varphi}$$

and introduce a rule

$$(\textbf{Const-BTA}) \quad \vdash c : \mathtt{f}$$

The Soundness and Completeness Theorem for the strictness logic can then be modified to cover the binding time analysis. The relationship between formulae and domains is unchanged; only now the semantics of the formula $\mathtt{f}$ is the ideal $\downarrow\{S\}$, *i.e.*, $\mathtt{f}$ represents *"static"* in binding time analysis. In the Soundness part we reason

as follows:

$$\Gamma \vDash b : \mathtt{f} \quad \text{and} \quad \Gamma \vDash e_1 : \varphi \quad \text{and} \quad \Gamma \vDash e_2 : \varphi$$

implies that

$$[\![b]\!]\rho_\Gamma = S \quad \text{and} \quad [\![e_1]\!]\rho_\Gamma \in [\![\varphi]\!] \quad \text{and} \quad [\![e_2]\!]\rho_\Gamma \in [\![\varphi]\!]$$

and since $[\![\varphi]\!]$ is an ideal and hence closed under $\sqcup$ we get that

$$[\![\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\rho_\Gamma = [\![e_1]\!]\rho_\Gamma \sqcup [\![e_2]\!]\rho_\Gamma \in [\![\varphi]\!]$$

i.e.,

$$\Gamma \vDash \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \varphi.$$

For the Completeness part we assume that

$$[\![\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\rho_\Gamma \subseteq [\![\varphi]\!].$$

If $\varphi = \mathtt{t}$ we can prove $\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \mathtt{t}$ by the rule **Taut**. So assume $\varphi \neq \mathtt{t}$. Then $[\![\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\rho_\Gamma$ must have been calculated by the second clause in the definition in (3.2) above, implying that

$$[\![b]\!]\rho_\Gamma = S \quad \text{and} \quad [\![e_1]\!]\rho_\Gamma \sqcup [\![e_2]\!]\rho_\Gamma \in [\![\varphi]\!]$$

which entails that

$$[\![b]\!]\rho_\Gamma = S \quad \text{and} \quad [\![e_1]\!]\rho_\Gamma \in [\![\varphi]\!] \quad \text{and} \quad [\![e_2]\!]\rho_\Gamma \in [\![\varphi]\!]$$

and the induction hypothesis now applies to give

$$\Gamma \vdash b : \mathtt{f} \quad \text{and} \quad \Gamma \vdash e_1 : \varphi \quad \text{and} \quad \Gamma \vdash e_2 : \varphi$$

and rule **If-BTA** yields

$$\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \varphi.$$

A similar argument shows that the rule for constants is in correspondence with the abstract interpretation.

3.5 Intersection types

In this section we look at some other uses of type systems with conjunctions. The program logic presented here relies on all terms being typed in advance. Logics with conjunctions for untyped lambda calculus have been considered in a number of papers [CDV80, BCD83, vB92, Pie91] under the name of intersection types. We start by considering the system by Barendregt *et. al.* [BCD83] since this is closest to the one presented here. There, the set of intersection types is defined by

$$\tau := \varphi_i \mid \omega \mid \tau \to \tau \mid \tau \cap \tau$$

where $\varphi_1, \varphi_2, \ldots$ is a set of type variables and ω is the "empty" type valid for all lambda terms. In *loc. cit.* a preorder on types is defined. This preorder can be regarded as an untyped equivalent to our $\leq$ (although products are not considered there). It is defined by

- $\varphi \leq \varphi$
- $\dfrac{\varphi \leq \psi, \psi \leq \chi}{\varphi \leq \chi}$
- $\dfrac{\varphi_1 \leq \psi_1, \varphi_2 \leq \psi_2}{\varphi_1 \cap \varphi_2 \leq \psi_1 \cap \psi_2}$
- $\varphi \cap \psi \leq \varphi$
- $\varphi \cap \psi \leq \psi$
- $\tau \leq \omega$
- $\omega \leq \omega \to \omega$
- $\varphi \to \psi_1 \cap \varphi \to \psi_2 = \varphi \to (\psi_1 \cap \psi_2)$
- $\dfrac{\varphi' \leq \varphi, \psi \leq \psi'}{\varphi \to \psi \leq \varphi' \to \psi'}$

Barendregt *et. al.* write ω and $\cap$ where we write $\mathtt{t}$ and $\wedge$ but otherwise these rules correspond to rules in Figure 3.2. The type inference rules are

$$\begin{array}{c}[x:\sigma] \\ \vdots \\ M:\tau\end{array}$$

- $\dfrac{M : \tau}{\lambda x.M : \sigma \to \tau}$
- $\dfrac{M : \sigma \to \tau \quad N : \sigma}{MN : \tau}$
- $\dfrac{M : \sigma \quad M : \tau}{M : \sigma \cap \tau}$
- $(\leq)\ \dfrac{M : \sigma \quad \sigma \leq \tau}{M : \tau}$
- $\dfrac{}{M : \omega}$

If B is a set of assumptions on term variables we write $B \vdash M : \sigma$ if $M : \sigma$

is deducible using the assumptions in B. The main result about the intersection type discipline is that it can characterise the terms having a *normal form* and a *head normal form* under β-reduction [Bar84]. More precisely, it is shown that for a lambda term M

- $\exists B\ \exists \tau \neq \omega\ .\ B \vdash M : \tau \Leftrightarrow M$ has a head normal form.
- $\exists B\ \exists \tau\ .\ B \vdash M : \tau$ and ω not in $B, \tau \Leftrightarrow M$ has a normal form.

The type system is so powerful that the following is a valid rule

$$\frac{M : \sigma \qquad M =_\beta N}{N : \sigma}$$

i.e., β-equivalent terms have the same types. Since β-equality is not a decidable property for pure, un-typed lambda calculus, it follows that there can be no automatic procedure for inferring a principal type for a lambda term in the intersection type system. This suggests that it is not immediately possible to apply the framework presented here to existing abstract interpretation for the untyped lambda calculus [HY86].

The above system is an extension of a system considered by Coppo *et. al.* [CDV80] where intersections can occur on the left hand side of an arrow but not on the right hand side and the $\leq$-rule has been removed from the system. More recently, Bakel has suggested a system of *strict types* intermediate between the system by Coppo *et. al.* and the system above. The inference rules of the strict system are

$$\begin{array}{c} [x : \sigma] \\ \vdots \\ M : \tau \end{array}$$

- $\dfrac{}{\lambda x.M : \sigma \to \tau}$
- $\dfrac{x : \sigma_1 \cap \ldots \cap \sigma_n}{x : \sigma_i}$
- $\dfrac{M : (\sigma_1 \cap \ldots \cap \sigma_n) \to \tau \quad N : \sigma_1 \ldots N : \sigma_n}{MN : \tau}$

where only the σ in the assumption in the rule form lambda abstraction can be an intersection of more than one type.

Note that the $\leq$-rule is not part of the strict type system either. However, Bakel [vB92] shows that it is indeed valid. Bakel also shows that typing is invariant under β-conversion so the strict type system has the same computability problems as the original intersection type system.

3.6 Summary

We have presented an abstract interpretation and a conjunctive program logic and proved that these two program analyses can establish the same set of properties of a given program. We have thus solved the problem of finding a logic equivalent in power to abstract interpretation and, as a by-product, obtained a characterisation of the kind of properties the abstract interpretation can determine of a program. The equivalence was first obtained for higher order strictness analysis and then for binding time analysis.

We have not made any attempt to prove the strictness logic correct with respect to a standard semantics for the language. This is because, once we have established the equivalence to the abstract interpretation we can rely on previous correctness results for the abstract interpretation. A proof for the logic would interpret the strictness properties in the standard semantics and prove the logical rules correct with respect to the standard semantics. This has been done by Benton [Ben92a] where (independently of our work reported in this chapter and in an earlier paper [Jen91]) Benton arrives at the same strictness logic as ours and proves it sound with respect to the standard denotational semantics.

Chapter 4

Disjunctive properties

4.1 Introduction

So far we have considered logics where properties could be combined by a conjunction and we have seen that such a logic corresponds to the usual abstract interpretation of higher order functions [BHA86]. This chapter considers the relationship between abstract interpretations and program logics for reasoning about disjunctions of properties. To see that such properties are of interest in program analysis consider the following functions:

$$\begin{aligned} \mathsf{f\ x} &= \mathsf{if\ B\ then\ (x,17)\ else\ (17,x)} \\ \mathsf{g\ x} &= \mathsf{plus(f(x))} \end{aligned}$$

where plus is addition of integers. Given that the value of the Boolean expression B cannot be determined at analysis time the most precise description of the result of applying f to an undefined argument is that it is a pair where either the first or the second component is undefined. Applying plus to such a pair gives an undefined result so g is a strict function. However in the formal system of conjunctive properties defined in Figure 3.2 the disjunction describing the pair cannot be expressed. The only safe description is to say that f returns a pair where both components may be defined. Applying plus to such a pair may give a well defined result and so the logic has failed to detect that g is strict.

The programme of this chapter is to extend the logic developed in the previous chapter to take disjunctions into account and to develop an abstract interpretation that is a sound and complete model of the logic. In Section 4.2 we give a formal

system that for each type σ extends the set of formulae $L(\sigma)$ to include disjunctions of formulae and axiomatises the set of disjunctive strictness properties of type σ. We prove a disjunctive normal form theorem for this system which states that all formulae can be written as a disjunction of a certain type of *irreducible* formulae. Section 4.3 defines a semantics for the formal system where a disjunctive property is modelled by a union of ideals just as a conjunctive property was modelled by a single ideal. It is shown that this is a sound and complete model of the formal system. In Section 4.4 we define a program logic for strictness analysis of a typed lambda calculus using the disjunctive properties and in Section 4.5 we present an abstract interpretation where the abstract domains are the same lattices as those that model the disjunctive properties. The main result, proved in Section 4.6 is that this abstract interpretation is a sound and complete model of the disjunctive strictness logic, thus, it is a disjunctive abstract interpretation. Section 4.7 relates the work presented here to earlier work by Nielson [Nie84, Nie85] and Burn [Bur92] on abstract interpretations for analysing disjunctive properties using the tensor product of lattices. The close connection between disjunctive program logic and an abstract interpretation using tensor products substantiates the claim that tensor products capture disjunctive properties.

4.2 Axiomatisation

This section describes an extension of the formal system for reasoning about conjunctive properties defined in Figure 3.2. The extension amounts to introducing a new logical connective: the disjunction. In the same way as with the conjunctive formulae we define for each type

$$\sigma = \mathsf{Int} \mid \mathsf{Bool} \mid \sigma \times \tau \mid \sigma \to \tau$$

in our language a set of formulae $L(\sigma)$, denoting the properties of that type, and axiomatise the logical structure on these sets of formulae.

The sets of formulae are defined inductively over the type structure as shown in Figure 4.1. The formulae also appearing in the formal system of conjunctive properties have the same interpretation in the extended system so the formula `f` represents the strongest property that implies all other properties, `t` is the trivial property satisfied by all expressions, the formula $\phi \times \psi$ describes a pair where the first (resp. the second) component satisfies the property ϕ (resp. ψ) and $\phi \to \psi$ means that the function maps arguments satisfying ϕ to results satisfying ψ. The introduction of the disjunction as a new logical connective requires new axioms and rules for defining the entailment relation on formulae containing disjunctions. This means that the

Formulae

- $\mathtt{f}, \mathtt{t} \in L(\sigma)$
- $\dfrac{\varphi, \psi \in L(\sigma)}{\varphi \vee \psi \in L(\sigma)}$
- $\dfrac{\varphi, \psi \in L(\sigma)}{\varphi \wedge \psi \in L(\sigma)}$
- $\dfrac{\varphi \in L(\sigma), \psi \in L(\tau)}{\varphi \times \psi \in L(\sigma \times \tau)}$
- $\dfrac{\varphi \in L(\sigma), \psi \in L(\tau)}{\varphi \to \psi \in L(\sigma \to \tau)}$

Logical rules

- $\varphi \leq \varphi$
- $\dfrac{\varphi \leq \psi, \psi \leq \chi}{\varphi \leq \chi}$
- $\varphi \wedge \psi \leq \varphi$
- $\dfrac{\varphi \leq \psi_1, \varphi \leq \psi_2}{\varphi \leq \psi_1 \wedge \psi_2}$
- $\varphi \wedge \psi \leq \psi$
- $\varphi \leq \psi \vee \varphi$
- $\dfrac{\varphi_1 \leq \psi, \varphi_2 \leq \psi}{\varphi_1 \vee \varphi_2 \leq \psi}$
- $\psi \leq \psi \vee \varphi$
- $\dfrac{\varphi \leq \psi, \psi \leq \varphi}{\varphi = \psi}$
- $\dfrac{\varphi = \psi}{\varphi \leq \psi, \psi \leq \varphi}$
- $\varphi \leq \mathtt{t}$
- $\mathtt{t}_{\sigma \to \tau} = \mathtt{t}_\sigma \to \mathtt{t}_\tau$
- $\mathtt{t}_{\sigma \times \tau} = \mathtt{t}_\sigma \times \mathtt{t}_\tau$
- $\mathtt{f} \leq \varphi$
- $\mathtt{t}_\sigma \to \mathtt{f}_\tau = \mathtt{f}_{\sigma \to \tau}$
- $\mathtt{f}_\sigma \times \mathtt{f}_\tau = \mathtt{f}_{\sigma \times \tau}$
- $\varphi \wedge (\psi_1 \vee \psi_2) = (\varphi \wedge \psi_1) \vee (\varphi \wedge \psi_2)$

Type-specific rules

- $(\varphi_1 \times \psi) \vee (\varphi_2 \times \psi) = (\varphi_1 \vee \varphi_2) \times \psi$
- $(\varphi \times \psi_1) \vee (\varphi \times \psi_2) = \varphi \times (\psi_1 \vee \psi_2)$
- $(\varphi_1 \times \psi_1) \wedge (\varphi_2 \times \psi_2) = (\varphi_1 \wedge \varphi_2) \times (\psi_1 \wedge \psi_2)$
- $\dfrac{\varphi_1 \leq \varphi_2, \psi_1 \leq \psi_2}{\varphi_1 \times \psi_1 \leq \varphi_2 \times \psi_2}$

- $(\varphi \to \psi_1) \wedge (\varphi \to \psi_2) = \varphi \to (\psi_1 \wedge \psi_2)$
- $(\varphi_1 \to \psi) \wedge (\varphi_2 \to \psi) = (\varphi_1 \vee \varphi_2) \to \psi$
- $\dfrac{\varphi_2 \leq \varphi_1, \psi_1 \leq \psi_2}{\varphi_1 \to \psi_1 \leq \varphi_2 \to \psi_2}$

Figure 4.1: Axiomatisation of disjunctive properties

Normal Form Theorem for the formal system changes significantly. Similar systems have been considered by Abramsky [Abr91b] and Barbaneri and Dezani-Ciancaglini [BDC91].

In preparation for the Normal Form Theorem for formulae we introduce the syntactic predicate Irr, which characterises the class of formulae that cannot be expressed as a disjunction of simpler formulae. Irr is defined as follows:

- $\mathsf{Irr}(\mathtt{f})$
- $\mathsf{Irr}(\mathtt{t})$
- $\dfrac{\mathsf{Irr}(\varphi) \quad \mathsf{Irr}(\psi)}{\mathsf{Irr}(\varphi \times \psi)}$
- $\dfrac{\mathsf{Irr}(\varphi_i)}{\mathsf{Irr}(\bigwedge_i(\varphi_i \to \psi_i))}$

Furthermore we write

$$\mathsf{Irr}(\sigma) = \{\varphi \in L(\sigma) \mid \mathsf{Irr}(\varphi)\}$$

for the set of irreducible formulae for a given type σ. Later we show that all properties can be written as a disjunction of formulae satisfying Irr.

The axiomatisation consists of two sets of rules: The logical rules define the generic logical connectives $\wedge, \vee, \mathtt{f}, \mathtt{t}$ in terms of the entailment relation $\leq$. The type-specific rules define the logic governing the properties of objects of product and function types. This axiomatisation defines a logical system

$$\mathcal{L}(\sigma) = (L(\sigma), \wedge, \vee, \mathtt{f}, \mathtt{t}, \leq, =)$$

where $=$ is provable equality. The axiomatisation is shown in Figure 4.1. Most of the rules have a straightforward logical reading. The rule for rewriting disjunctions in the argument position of a formula of function type can be motivated as follows: saying of a function f that it maps arguments satisfying either φ_1 or φ_2 to results satisfying ψ is the same as saying that f maps arguments satisfying φ_1 to ψ and f maps arguments satisfying φ_2 to ψ. This is the content of the rule

$$(\varphi_1 \to \psi) \wedge (\varphi_2 \to \psi) = (\varphi_1 \vee \varphi_2) \to \psi$$

Note that there is no rule for rewriting function formulae where the disjunction appears in the result position. A rule like

$$\varphi \to (\psi_1 \vee \psi_2) = \varphi \to \psi_1 \vee \varphi \to \psi_2$$

is not in general valid. A formal system that axiomatises precisely when this equality holds for formulae denoting strictness properties has been developed by Benton [Ben92b]. Our logic is thus weaker than the one considered there but simpler to deal with. In particular it is not clear how to obtain a normal form for formulae of function types in Benton's logic.

The formal logic satisfies the following *disjunctive normal form* theorem :

Theorem 4.2.1 *For all $\varphi \in L(\sigma)$ there exist $\psi_1, \ldots \psi_n \in \mathsf{Irr}(\sigma)$ such that*

$$\varphi = \bigvee_{i=1}^{n} \psi_i \qquad n \geq 1$$

Proof. By structural induction over the formation rules for formulae. The formulae $\mathtt{f}, \mathtt{t}$ are already in this form. Now assume inductively that $\varphi = \bigvee_i \varphi_i$ and $\psi = \bigvee_j \psi_j$. The formula $\varphi \vee \psi$ is already in disjunctive normal form and the formula $\varphi \wedge \psi$ can be converted into disjunctive normal form by repeated use of the distributive rule. For formulae describing products we note that

$$\bigvee_{i \in I} \varphi_i \times \bigvee_{j \in J} \psi_j = \bigvee_{(i,j) \in I \times J} (\varphi_i \times \psi_j)$$

and for formulae of function type we have that

$$(\bigvee_i \varphi_i) \to (\bigvee_j \psi_j)$$

can be rewritten into

$$\bigwedge_i (\varphi_i \to \bigvee_j \psi_j)$$

which satisfies the Irr predicate. ■

4.3 Partial order semantics of formulae

In this section we give a semantics to the formal logic defined above based on partial orders. In the previous chapter we saw that conjunctive properties are modelled by the ideals of the lattices used in the abstract interpretation. We let the Normal Form Theorem guide us in the search for a similar model for the disjunctive properties. The Normal Form Theorem states that every property is the finite disjunction of irreducible properties so the theorem splits the task in two: find a way of modelling irreducible properties and extend it to model disjunctions.

The irreducible properties are almost the same as the conjunctive properties axiomatised in Figure 3.2; the only difference being that arbitrary formulae, and not only irreducible ones, can occur in the result position of a formula of function type. This suggests that we can use the same kind of structures to model irreducibles as we used to model conjunctive properties. Thus the irreducible formulae of product type $\sigma \times \tau$ can be modelled by ideals in the cartesian product of the lattices modelling the irreducibles of σ and τ. The irreducible formulae of type $\sigma \to \tau$, which are of the form $\bigwedge(\varphi \to \psi)$ with φ irreducible, can be modelled by ideals in the lattice of monotone functions from the lattice modelling $\mathsf{Irr}(\sigma)$ to the lattice modelling the full set of τ-formulae.

In the conjunctive case we modelled a formula in $L(\sigma)$ by the set of elements in $\sigma^{\#}$ satisfying the property. For an irreducible property this set is an ideal. Carrying this idea over to the system of disjunctive formulae we observe that an element satisfies a disjunction if it satisfies one of the disjuncts, *i.e.*, if it belongs to one of the sets corresponding to the disjuncts. Hence we model disjunctive properties as finite unions of ideals. Such sets are characterised by being downwards closed subsets of the lattice and are called *lower sets*.

Definition 4.3.1 *A* lower subset *of a partial order P is a finite, non-empty subset S of P that is downwards closed,* i.e.*, it satisfies*

$$p_1 \sqsubseteq p_2 \quad \textit{and} \quad p_2 \in S \quad \Rightarrow \quad p_1 \in S.$$

We introduce the notation

$$\downarrow\{p_1, \ldots, p_n\} = \{p \mid p \sqsubseteq p_i \text{ for some } p_i\}$$

to write the downwards closure of a set of elements and denote by $\mathsf{D}(P)$ the set of lower subsets of P. This provides us with a technique for building distributive lattices from partial orders.

Lemma 4.3.2 *If P is a partial order with a least element 0 then $(\mathsf{D}(P), \cup, \cap, \subseteq)$ is a lattice with least element the lower set $\{0\}$ and greatest element P. The function $p \mapsto \downarrow\{p\}$ is an order-embedding.*

Proof. We just have to observe that the existence of a least element in P guarantees that the intersection of two lower sets is non-empty. The rest of the verification is straightforward. ■

$$
\begin{aligned}
[\![\mathsf{Int}]\!] &= \mathsf{D}(\mathbf{2}) \\
[\![\mathsf{Bool}]\!] &= \mathsf{D}(\mathbf{2}) \\
[\![\sigma \times \tau]\!] &= \mathsf{D}(\{(a,b) \mid \downarrow\{a\} \in [\![\sigma]\!], \downarrow\{b\} \in [\![\tau]\!]\} \\
[\![\sigma \to \tau]\!] &= \mathsf{D}([\![\sigma]\!] \multimap [\![\tau]\!])
\end{aligned}
$$

Figure 4.2: Types modelled as lower sets

The Lemma is a variation of a general result from lattice theory known as Birkhoff's Representation Theorem [DP90, Theorem 8.17]. This theorem states that every distributive lattice can be represented as the lattice $\mathsf{D}(P)$ of lower sets of a partial order P. Furthermore the P generating a distributive lattice D can be taken to be the set of join-irreducible elements of D[1].

When specifying the distributive lattice modelling a type σ we shall do it by defining a partial order $\mathcal{J}(\sigma)$ and then take the model $[\![\sigma]\!]$ to be $\mathsf{D}(\mathcal{J}(\sigma))$. The partial order $\mathcal{J}(\sigma)$ is thus intended as the model of the irreducible formulae of $\mathcal{L}(\sigma)$ and the lower set operation $\mathsf{D}(\ )$ is the extension to handle disjunctions. We say that $[\![\sigma]\!]$ is obtained by *Birkhoff completion* of $\mathcal{J}(\sigma)$.

Base types. An irreducible formula of base type is equivalent to either `t` or `f` so the two-point lattice $\mathbf{2}$ suffices. Thus $[\![\mathsf{Int}]\!] = [\![\mathsf{Bool}]\!] = \mathsf{D}(\mathbf{2})$.

Product. An irreducible formula of type $\sigma \times \tau$ is of form $\varphi_i \times \psi_i$ with φ_i, ψ_i irreducible. We shall therefore define

$$\mathcal{J}(\sigma \times \tau) = \mathcal{J}(\sigma) \times \mathcal{J}(\tau)$$

thus the lattice modelling $\sigma \times \tau$ consists of lower sets of pairs where the elements of the pairs correspond to irreducible properties.

$$[\![\sigma \times \tau]\!] = \mathsf{D}(\mathcal{J}(\sigma) \times \mathcal{J}(\tau)) = \mathsf{D}(\{(a,b) \mid \downarrow\{a\} \in [\![\sigma]\!], \downarrow\{b\} \in [\![\tau]\!]\}).$$

The first and second projection π_1 and π_2 that take out the first resp. the second component of a pair are extended to lower subsets of pairs by simply applying the relevant projection to each pair in the set. Due to the ordering on pairs this gives a

[1] An element p is *join-irreducible* if $p = p_1 \vee p_2 \ \Rightarrow\ p = p_1$ or $p = p_2$

lower set. Formally we define $\pi_1 : [\![\sigma \times \tau]\!] \to [\![\sigma]\!]$ by

$$\pi_1(\bigcup_{i=1}^{n} \downarrow\{(a_i, b_i)\}) = \bigcup_{i=1}^{n} \downarrow\{a_i\}.$$

and similarly for π_2.

Functions. An irreducible formula of function type has the form $\bigwedge(\varphi_i \to \psi)$ with every φ_i irreducible. This is the same shape as the conjunctive formulae of function type considered in the previous chapter. Intuitively, such a formula describes a monotone function that maps ideals (representing the irreducible formulae φ_i) into arbitrary lower sets (representing the ψ_i). Such a monotone function f can be extended to arbitrary lower sets by defining

$$f(\downarrow\{a_1, a_2\}) = f(\downarrow\{a_1\}) \cup f(\downarrow\{a_2\}).$$

A function satisfying this property is said to be *linear* and we say that f is extended *by linearity*. A linear function is completely determined by its value on ideals so this sets up a bijection between monotone functions from ideals to lower sets and linear functions from lower sets to lower sets. In particular, one linear function g is smaller than another linear function f in the pointwise ordering precisely when

$$g(\downarrow\{a\}) \subseteq f(\downarrow\{a\})$$

for all downwards closures of single elements. The notion of linearity can be defined for functions between any two join-semilattices.

Definition 4.3.3 *Let A,B be join-semilattices and let $f : A \to B$ be a function. Then f is* linear *if*

$$f(a_1 \sqcup a_2) = f(a_1) \sqcup f(a_2).$$

The set of linear functions from A to B is denoted by $A \multimap B$.

Thus it does not matter whether we use monotone functions defined on ideals or linear functions defined on lower sets to build our model of the function properties. We choose linear functions because it makes for an easier notation. Thus we define

$$\mathcal{J}(\sigma \to \tau) = [\![\sigma]\!] \multimap [\![\tau]\!]$$

and a formula of function type is then interpreted as a lower set of linear functions:

$$[\![\sigma \to \tau]\!] = \mathsf{D}([\![\sigma]\!] \multimap [\![\tau]\!]).$$

The foregoing is all summarised in Figure 4.2.

Let $\varphi, \varphi_1, \varphi_2 \in \mathcal{L}(\sigma)$ and $\psi \in \mathcal{L}(\tau)$.

$$
\begin{array}{lcl}
[\![]\!]_\sigma : L(\sigma) \to [\![\sigma]\!] & & \\
[\![\mathtt{t}]\!]_\sigma & = & 1_{[\![\sigma]\!]} \\
[\![\mathtt{f}]\!]_\sigma & = & 0_{[\![\sigma]\!]} \\
[\![\varphi_1 \wedge \varphi_2]\!]_\sigma & = & [\![\varphi_1]\!]_\sigma \cap [\![\varphi_2]\!]_\sigma \\
[\![\varphi_1 \vee \varphi_2]\!]_\sigma & = & [\![\varphi_1]\!]_\sigma \cup [\![\varphi_2]\!]_\sigma \\
[\![\varphi \times \psi]\!]_{\sigma\times\tau} & = & \{(a,b) \mid a \in [\![\varphi]\!]_\sigma, b \in [\![\psi]\!]_\tau\} = [\![\varphi]\!]_\sigma \times [\![\psi]\!]_\tau \\
[\![\varphi{\to}\psi]\!]_{\sigma\to\tau} & = & \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f([\![\varphi]\!]_\sigma) \subseteq [\![\psi]\!]_\tau\}
\end{array}
$$

Figure 4.3: Semantics of formulae

The logical formulae of $\mathcal{L}(\sigma)$ can be interpreted in $[\![\sigma]\!]$ using the function $[\![\ ]\!]_\sigma$ as shown in Figure 4.3. The exact connection between logic and semantics is given in the three theorems to follow. The Definability Theorem states that every downwards closure of a single element is denoted by an irreducible formula. From this it follows that every lower set is denoted by some formula so the interpretation $[\![]\!]$ defines a surjective function from formulae to lower sets.

Theorem 4.3.4 (Definability) *For all ideals $\downarrow\{a\} \in [\![\sigma]\!]$ there exists a $\varphi_a \in \mathcal{L}(\sigma)$ such that*

$$\mathsf{Irr}(\varphi_a) \quad \textit{and} \quad \downarrow\{a\} = [\![\varphi_a]\!]_\sigma$$

Proof. The proof is by induction over the structure of σ. For the base case $\sigma = \mathsf{Int}$ and $\sigma = \mathsf{Bool}$ we have that

$$\{0\} = [\![\mathtt{f}]\!] \qquad \text{and} \qquad \{0,1\} = [\![\mathtt{t}]\!].$$

For product type $\sigma \times \tau$ the downwards closures are of the form

$$\downarrow\{(a,b)\}$$

with $\downarrow\{a\} \in [\![\sigma]\!]$ and $\downarrow\{b\} \in [\![\tau]\!]$. By induction hypothesis we then have that there exist irreducibles φ and ψ such that

$$\downarrow\{a\} = [\![\varphi]\!] \qquad \text{and} \qquad \downarrow\{b\} = [\![\psi]\!]$$

and we then have from the definition of the semantics of product formulae that

$$[\![\varphi \times \psi]\!] = \{(a,b) \mid a \in {\downarrow}\{a\}, b \in {\downarrow}\{b\}\} = {\downarrow}\{(a,b)\}.$$

For a function type $[\![\sigma \to \tau]\!]$ we consider downwards closures of the form ${\downarrow}\{f\}$ with f a linear map from $[\![\sigma]\!]$ to $[\![\tau]\!]$. As observed above a function is smaller than f if and only if it is smaller than f on all downwards closures of single elements, hence

$${\downarrow}\{f\} = \{g \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid g \sqsubseteq f\} = \bigcap_{{\downarrow}\{a\} \in [\![\sigma]\!]} \{g \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid g({\downarrow}\{a\}) \subseteq f({\downarrow}\{a\})\}.$$

To every ${\downarrow}\{a\}$ there exists an irreducible $\varphi_{{\downarrow}\{a\}}$ such that ${\downarrow}\{a\} = [\![\varphi_{{\downarrow}\{a\}}]\!]$. The lower set $f({\downarrow}\{a\})$ can be written as a union of ideals belonging to $[\![\tau]\!]$ and the induction hypothesis can therefore be applied to obtain a disjunction of irreducibles, whose interpretation is $f({\downarrow}\{a\})$. We denote this disjunction by $\psi_{f({\downarrow}\{a\})}$. The equations above then yields that

$${\downarrow}\{f\} = [\![\bigwedge_{{\downarrow}\{a\} \in [\![\sigma]\!]} \varphi_{{\downarrow}\{a\}} \to \psi_{f({\downarrow}\{a\})}]\!]$$

and the latter formula is irreducible since the $\varphi_{{\downarrow}\{a\}}$ are. ■

It should be noted that in dealing with the function space we used the same technique as for the conjunctive case (Theorem 3.2.6) to represent a downwards closure of a function by finding a formula that essentially tabulates the function.

The Soundness Theorem states that any entailment that can be proved in the formal system is properly reflected in the semantics as an inclusion between subsets.

Theorem 4.3.5 (Soundness) *For all elements $\varphi, \psi \in \mathcal{L}(\sigma)$:*

$$\varphi \leq \psi \quad \Rightarrow \quad [\![\varphi]\!]_\sigma \subseteq [\![\psi]\!]_\sigma$$

Proof. By checking the validity of each axiom and inference rule. We validate a selection of the axioms here.

- $[\![\mathtt{t}_\sigma \to \mathtt{t}_\tau]\!]$

$= \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f({\downarrow}\{1_\sigma\}) \subseteq {\downarrow}\{1_\sigma\}\}$

$= [\![\sigma]\!] \multimap [\![\tau]\!] = [\![\mathtt{t}_{\sigma\to\tau}]\!]$

- $[\![\mathtt{t}_\sigma \to \mathtt{f}_\tau]\!]$

$= \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f({\downarrow}\{1_\sigma\}) \subseteq \{0_\tau\}\} = \{\lambda x.\{0_\tau\}\} = [\![\mathtt{f}_{\sigma\to\tau}]\!]$

- $[\![(\varphi_1 \times \psi) \vee (\varphi_2 \times \psi)]\!]$

$= \{(a,b) \mid a \in [\![\varphi_1]\!], b \in [\![\psi]\!]\} \cup \{(a,b) \mid a \in [\![\varphi_2]\!], b \in [\![\psi]\!]\}$

$= \{(a,b) \mid a \in [\![\varphi_1]\!] \cup [\![\varphi_2]\!], b \in [\![\psi]\!]\} = \{(a,b) \mid a \in [\![\varphi_1 \vee \varphi_2]\!], b \in [\![\psi]\!]\} = [\![(\varphi_1 \vee \varphi_2) \times \psi]\!]$

- $[\![(\varphi_1 \times \psi_1) \wedge (\varphi_2 \times \psi_2)]\!]$

$= \{(a,b) \mid a \in [\![\varphi_1]\!], b \in [\![\psi_1]\!]\} \cap \{(a,b) \mid a \in [\![\varphi_2]\!], b \in [\![\psi_2]\!]\}$

$= \{(a,b) \mid a \in [\![\varphi_1]\!] \cap [\![\varphi_2]\!], b \in [\![\psi_1]\!] \cap [\![\psi_2]\!]\}$

$= [\![\varphi_1 \wedge \varphi_2 \times \psi_1 \wedge \psi_2]\!]$

- $[\![(\varphi \to \psi_1) \wedge (\varphi \to \psi_2)]\!]$

$= \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f([\![\varphi]\!]) \subseteq [\![\psi_1]\!]\} \cap \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f([\![\varphi]\!]) \subseteq [\![\psi_2]\!]\}$

$= \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f([\![\varphi]\!]) \subseteq [\![\psi_1]\!] \cap [\![\psi_2]\!]\}$

$= \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f([\![\varphi]\!]) \subseteq [\![\psi_1 \wedge \psi_2]\!]\} = [\![\varphi \to \psi_1 \wedge \psi_2]\!]$

- $[\![(\varphi_1 \to \psi) \wedge (\varphi_2 \to \psi)]\!]$

$= \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f([\![\varphi_1]\!]) \subseteq [\![\psi]\!]\} \cap \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f([\![\varphi_2]\!]) \subseteq [\![\psi]\!]\}$

$= \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f([\![\varphi_1]\!] \cup [\![\varphi_2]\!]) \subseteq [\![\psi]\!]\} = [\![\varphi_1 \vee \varphi_2 \to \psi]\!]$

■

As a consequence of the Soundness Theorem we have that if two formulae are provably equal then they denote the same lower set.

The Completeness Theorem states that all inclusions between lower sets that hold in the semantics can be proved in the formal system. Since all lower sets in $[\![\sigma]\!]$ are denotable by a formula in $L(\sigma)$ this can be established by proving that an inclusion between the interpretation of two formulae corresponds to a logical entailment between the two formulae in the formal system.

Theorem 4.3.6 (Completeness) *For all elements $\varphi, \psi \in \mathcal{L}(\sigma)$:*

$$[\![\varphi]\!]_\sigma \subseteq [\![\psi]\!]_\sigma \quad \Rightarrow \quad \varphi \leq \psi$$

Proof. The strategy of the proof is to proceed by induction over the structure of σ. From the assumption that the theorem holds for all the constituent types of σ we prove that it holds for irreducible formulae of type σ. Given that we have proved completeness for irreducible formulae we can prove completeness for all formulae

as follows: we know that for arbitrary φ, ψ there exist φ_i, ψ_j satisfying $\mathsf{Irr}(\varphi_i)$ and $\mathsf{Irr}(\psi_j)$ such that

$$\varphi = \bigvee \varphi_i \qquad \psi = \bigvee \psi_j$$

For each φ_i and ψ_j there exist a_i, b_j such that

$$[\![\varphi_i]\!]_\sigma = {\downarrow}\{a_i\} \qquad [\![\psi_j]\!]_\sigma = {\downarrow}\{b_j\}$$

Completeness can then be proved as follows:

$$\begin{aligned}
[\![\varphi]\!]_\sigma \subseteq [\![\psi]\!]_\sigma &\Rightarrow \cup_i [\![\varphi_i]\!]_\sigma \subseteq \cup_j [\![\psi_j]\!]_\sigma \\
&\Rightarrow \cup_i {\downarrow}\{a_i\} \subseteq \cup_j {\downarrow}\{b_j\} \\
&\Rightarrow \forall i\ \exists j. {\downarrow}\{a_i\} \subseteq {\downarrow}\{b_j\} \\
&\Rightarrow \forall i\ \exists j. \varphi_i \leq \psi_j \\
&\Rightarrow \varphi = \bigvee \varphi_i \leq \bigvee \psi_j = \psi
\end{aligned}$$

So we have to prove completeness for irreducibles. For base types Int and Bool every irreducible is a conjunction of `t` and `f` and therefore provably equal to either `t` or `f`. The only non-trivial inclusion $[\![\mathtt{f}]\!] = \{0\} \subseteq \{0,1\} = [\![\mathtt{t}]\!]$ is proved by the fact $\mathtt{f} \leq \mathtt{t}$.

For a product type $\sigma \times \tau$ the proof is simple:

$$\begin{aligned}
[\![\varphi_1 \times \psi_1]\!] \subseteq [\![\varphi_2 \times \psi_2]\!] &\Rightarrow [\![\varphi_1]\!] \subseteq [\![\varphi_2]\!] \quad \text{and} \quad [\![\psi_1]\!] \subseteq [\![\psi_2]\!] \\
&\Rightarrow \varphi_1 \leq \varphi_2 \quad \text{and} \quad \psi_1 \leq \psi_2 \\
&\Rightarrow \varphi_1 \times \psi_1 \leq \varphi_2 \times \psi_2
\end{aligned}$$

For function types we have as assumption that

$$[\![\bigwedge_{i \in I} \varphi_i \to \psi_i]\!] \subseteq [\![\bigwedge_{j \in J} \varphi_j \to \psi_j]\!]$$

which implies that

$$\forall j \in J\ .\ [\![\bigwedge_{i \in I} \varphi_i \to \psi_i]\!] \subseteq [\![\varphi_j \to \psi_j]\!].$$

If we can prove that for fixed $j \in J$

$$[\![\bigwedge_{i \in I} \varphi_i \to \psi_i]\!] \subseteq [\![\varphi_j \to \psi_j]\!] \quad \Rightarrow \quad \bigwedge_{i \in I} \varphi_i \to \psi_i \leq \varphi_j \to \psi_j$$

the rules for $\wedge$ will give the desired implication. Leaving out the subscript on φ_j, ψ_j,

the proposition we have to prove can be stated as follows: if $\mathsf{Irr}(\varphi_i), \mathsf{Irr}(\varphi)$ then

$$[\![\bigwedge_{i\in I} \varphi_i \to \psi_i]\!] \subseteq [\![\varphi \to \psi]\!] \quad \Rightarrow \quad \bigwedge_{i\in I} \varphi_i \to \psi_i \leq \varphi \to \psi.$$

The assumption $\mathsf{Irr}(\varphi), \mathsf{Irr}(\varphi_i)$ means that there exist a, a_i such that $[\![\varphi]\!] = \downarrow\{a\}$ and $[\![\varphi_i]\!] = \downarrow\{a_i\}$ for all $i \in I$. The definition of the semantics of function type formulae translates the hypothesis to

$$\bigcap_{i\in I}\{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f(\downarrow\{a_i\}) \subseteq [\![\psi_i]\!]\} \subseteq \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f(\downarrow\{a\}) \subseteq [\![\psi]\!]\}. \quad (4.1)$$

Consider the set $I_0 = \{i \in I \mid \downarrow\{a\} \subseteq \downarrow\{a_i\}\}$. and let h denote the co-step function determined by the set of pairs

$$\{(\downarrow\{a_i\}, [\![\psi_i]\!]) \mid i \in I \setminus I_0\} \cup \{(\downarrow\{a_j\}, \cap_{i\in I_0}[\![\psi_i]\!]) \mid j \in I_0\}.$$

By the definition of co-step function the function h satisfies

$$h(\downarrow\{a\}) = \bigcap_{i\in I_0} [\![\psi_i]\!]$$

The linear extension of h belongs to the intersection of functions on the left hand side of the inequality 4.1 from which we get that

$$\bigcap_{i\in I_0} [\![\psi_i]\!] \subseteq [\![\psi]\!]$$

and by the induction hypothesis we can prove this inclusion in the formal system *i.e.*,

$$\bigwedge_{i\in I_0} \psi_i \leq \psi$$

Furthermore we have that

$$[\![\varphi]\!] = \downarrow\{a\} \subseteq \bigcap_{i\in I_0} \downarrow\{a_i\} = [\![\bigwedge_{i\in I_0} \varphi_i]\!]$$

so, again by completeness, we can prove

$$\varphi \leq \bigwedge_{i\in I_0} \varphi_i.$$

Putting these two facts together we can prove that

$$(\bigwedge_{i\in I_0} \varphi_i) \to (\bigwedge_{i\in I_0} \psi_i) \;\leq\; \varphi \to \psi.$$

$$\textbf{Conj}\quad \frac{\Gamma \vdash e : \psi_1 \quad \Gamma \vdash e : \psi_2}{\Gamma \vdash e : \psi_1 \wedge \psi_2} \qquad \textbf{Disj}\quad \frac{\Gamma[x \mapsto \phi_1] \vdash e : \psi \quad \Gamma[x \mapsto \phi_2] \vdash e : \psi}{\Gamma[x \mapsto \phi_1 \vee \phi_2] \vdash e : \psi}$$

$$\textbf{Taut}\quad \Gamma \vdash e : \mathtt{t} \qquad \textbf{Weak}\quad \frac{\Gamma \leq \Delta \quad \Delta \vdash e : \phi \quad \phi \leq \psi}{\Gamma \vdash e : \psi}$$

$$\textbf{Var}\quad \Gamma[x \mapsto \phi] \vdash x : \phi \qquad \textbf{Pair}\quad \frac{\Gamma \vdash e_1 : \phi_1 \quad \Gamma \vdash e_2 : \phi_2}{\Gamma \vdash (e_1, e_2) : \phi_1 \times \phi_2}$$

$$\textbf{Fst}\quad \frac{\Gamma \vdash e : \varphi \times \psi}{\Gamma \vdash \mathsf{fst}(e) : \varphi} \qquad \textbf{Snd}\quad \frac{\Gamma \vdash e : \varphi \times \psi}{\Gamma \vdash \mathsf{snd}(e) : \psi}$$

$$\textbf{Abs}\quad \frac{\Gamma[x \mapsto \phi] \vdash e : \psi}{\Gamma \vdash \lambda x.e : (\phi \to \psi)} \qquad \textbf{App}\quad \frac{\Gamma \vdash e_1 : (\phi \to \psi) \quad \Gamma \vdash e_2 : \phi}{\Gamma \vdash e_1 e_2 : \psi}$$

$$\textbf{Fix}\quad \vdash \mathsf{fix} : (\varphi \to \varphi) \to \varphi$$

$$\textbf{If-1}\quad \frac{\Gamma \vdash b : \mathtt{f}}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \mathtt{f}} \qquad \textbf{If-2}\quad \frac{\Gamma \vdash e_1 : \phi_1 \quad \Gamma \vdash e_2 : \phi_2}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \phi_1 \vee \phi_2}$$

Figure 4.4: Disjunctive strictness logic

The rules for $\to$ formulae furthermore gives that

$$\bigwedge_{i \in I_0} (\varphi_i \to \psi_i) \leq \bigwedge_{i \in I_0} ((\bigwedge_{i \in I_0} \varphi_i) \to \psi_i) \leq (\bigwedge_{i \in I_0} \varphi_i) \to (\bigwedge_{i \in I_0} \psi_i).$$

This allows the following deduction:

$$\bigwedge_{i \in I} (\varphi_i \to \psi_i) \leq \bigwedge_{i \in I_0} (\varphi_i \to \psi_i) \leq (\bigwedge_{i \in I_0} \varphi_i) \to (\bigwedge_{i \in I_0} \psi_i) \leq \varphi \to \psi$$

which was the entailment that we wanted to prove. ■

4.4 Disjunctive strictness analysis

The formal system axiomatising the disjunctive formulae will now be used to formulate an extension of the strictness logic presented in Figure 3.4. Thus we interpret the formulae as representing strictness properties of programs. The logic will be formulated in the same style as the conjunctive strictness logic with judgements of the form

$$\Gamma \vdash e : \varphi$$

where Γ is an environment associating variables with formulae, e a term of type σ and φ a formula in $L(\sigma)$. The logic is presented in Figure 4.4. It has three novel features. In the rule for if we can now combine the results of analysing each branch by a disjunction to obtain the strongest property that holds for both branches. This is demonstrated in the following example:

Example. Consider the function

$$\mathsf{f} \equiv \lambda x.\mathsf{if}\ B\ \mathsf{then}\ (x, 17)\ \mathsf{else}\ (17, x)$$

This is the function considered in the introduction. We verify that f satisfies the property:

$$\mathtt{f} \to (\mathtt{f} \times \mathtt{t} \vee \mathtt{t} \times \mathtt{f}).$$

Let Γ be the environment $[x \mapsto \mathtt{f}]$. Using **Var, Taut, Pair, If-2** and **Abs** we get the deduction:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : \mathtt{f}}\,\mathbf{Var} \quad \dfrac{}{\Gamma \vdash 17 : \mathtt{t}}\,\mathbf{Taut}}{\Gamma \vdash (x, 17) : \mathtt{f} \times \mathtt{t}}\,\mathbf{Pair} \quad \dfrac{\dfrac{}{\Gamma \vdash x : \mathtt{f}}\,\mathbf{Var} \quad \dfrac{}{\Gamma \vdash 17 : \mathtt{t}}\,\mathbf{Taut}}{\Gamma \vdash (17, x) : \mathtt{t} \times \mathtt{f}}\,\mathbf{Pair}}{\Gamma \vdash \mathsf{if}\ B\ \mathsf{then}\ (x, 17)\ \mathsf{else}\ (17, x)) : \mathtt{f} \times \mathtt{t} \vee \mathtt{t} \times \mathtt{f}}\,\mathbf{If-2}}{\vdash \lambda x.\mathsf{if}\ B\ \mathsf{then}\ (x, 17)\ \mathsf{else}\ (17, x) : \mathtt{f} \to (\mathtt{f} \times \mathtt{t} \vee \mathtt{t} \times \mathtt{f})}\,\mathbf{Abs}$$

The rule for the fixed point operator has been modified and is now axiomatised as a higher order function. The reason for this is mainly economical; by axiomatising it as a function we do not have to specify the behaviour of fix on more complex formulae, since this is determined by the axioms for functions. For example, it holds generally that

$$\bigwedge_{i\in I}(\varphi_i \to \varphi_i) \to \varphi_i \ \leq\ \bigwedge_{i\in I}((\varphi_i \to \varphi_i) \to (\bigvee_{i\in I} \varphi_i)) \ \leq\ \bigvee_{i\in I}(\varphi_i \to \varphi_i) \to \bigvee_{i\in I} \varphi_i$$

and since fix satisfies the formula to the left we get the following derived fact about fix :

$$\vdash \mathsf{fix}\ : \bigvee_{i\in I}(\varphi_i \to \varphi_i) \to \bigvee_{i\in I} \varphi_i$$

The formula $\bigvee_{i\in I}(\varphi_i \to \varphi_i)$ represents the knowledge that a function satisfies (at least) one of the properties $\varphi_i \to \varphi_i$. Hence the fixed point will satisfy (at least) one of the properties φ_i.

Disjunctions can also occur in the environment describing free variables in the ex-

pression. An environment where x is bound to the formula $\varphi \vee \psi$ indicates that x satisfies either φ or ψ for the whole of the evaluation of this expression. When analysing the components of the expression we must take care to ascribe the same property to x throughout. As an example consider analysing the expression (x, x) in an environment where x is bound to a disjunction $L \vee R$ (L and R could for example be the properties $\mathtt{f} \times \mathtt{t}$ and $\mathtt{t} \times \mathtt{f}$). Without the **Disj** rule we can only prove

$$[x : L \vee R] \vdash (x, x) : (L \vee R) \times (L \vee R)$$

which has normal form

$$(L \times L) \vee (L \times R) \vee (R \times L) \vee (R \times R).$$

However, by considering each disjunct in turn we can prove

$$\dfrac{\dfrac{}{[x : L] \vdash (x, x) : L \times L}\,\mathbf{Pair} \qquad \dfrac{}{[x : R] \vdash (x, x) : R \times R}\,\mathbf{Pair}}{[x : L \vee R] \vdash (x, x) : (L \times L) \vee (R \times R)}\,\mathbf{Disj}$$

which is a stronger property.

4.5 Disjunctive abstract interpretation

We now define an abstract interpretation that assigns disjunctive properties to expressions. Our main objective is to prove that the interpretation is a sound and complete model of the disjunctive program logic and is in this sense a *disjunctive* abstract interpretation.

The first problem is to find the abstract domains in which the expressions can be interpreted. In the conjunctive case, given the abstract domains beforehand, we could model the conjunctive properties as ideals in these domains. Here we have a model of the disjunctive properties as lower sets of some partial orders so one possibility would be to use *the maximal elements* of the lower sets as denotations. This would indeed allow us to model the properties as downwards closures of the denotations as we did in the conjunctive case. The problem with this approach is that both the ordering and the lattice operations are complicated to describe. The least upper bound of two such sets of maximal elements is not the union of the two sets since an element in one set can be smaller than an element in the other.

The proper way to compute the least upper bound is to form the union of the downwards closures of the maximal sets and then take the maximal elements of this

$$
\begin{array}{lcl}
[\![x^\sigma]\!]\rho & = & \rho(x) \\
[\![c^{\mathsf{Int}}]\!]\rho & = & [\![c^{\mathsf{Bool}}]\!]\rho = 1 \\
[\![(e_1, e_2)]\!]\rho & = & \bigcup \{[\![e_1]\!][x_i \mapsto \downarrow\{v_i\}] \times [\![e_2]\!][x_i \mapsto \downarrow\{v_i\}] \mid v_i \in \rho(x_i)\} \\
[\![\mathsf{fst}(e)]\!]\rho & = & \{a \mid (a, b) \in [\![e]\!]\rho\} \\
[\![\mathsf{snd}(e)]\!]\rho & = & \{b \mid (a, b) \in [\![e]\!]\rho\} \\
[\![\lambda x.e]\!]\rho & = & \bigcup \downarrow\{\lambda L. \bigcup_{l \in L} [\![e]\!][x_i \mapsto \downarrow\{v_i\}][x \mapsto \downarrow\{l\}] \mid v_i \in \rho(x_i)\} \\
[\![e_1 e_2]\!]\rho & = & \bigcup \{[\![e_1]\!][x_i \mapsto \downarrow\{v_i\}] \mathbin{@} [\![e_2]\!][x_i \mapsto \downarrow\{v_i\}] \mid v_i \in \rho(x_i)\} \\
[\![\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\rho & = & \begin{cases} \downarrow\{0\} & \text{if } [\![b]\!]\rho = \downarrow\{0_{\mathsf{Bool}}\} \\ [\![e_1]\!]\rho \cup [\![e_2]\!]\rho & \text{otherwise} \end{cases} \\
[\![\mathsf{fix}]\!] & = & \downarrow\{\lambda\downarrow\{f_1, \ldots, f_k\}. \bigcup_{i=1}^{k} \bigcup_{n=0}^{\infty} f_i^n(\{0\})\}
\end{array}
$$

Figure 4.5: Disjunctive abstract interpretation

union. But this suggests another possibility: use the lower sets as denotations. This would allow us to use the usual set theoretic notation, least upper bound is just set union and the ordering is set inclusion. We shall choose this latter option but we stress that the other approach is equally viable.

We define the abstract interpretation as a function

$$[\![\ ]\!] : \Lambda_T(\sigma) \rightarrow Env \rightarrow [\![\sigma]\!]$$

which, given an environment associating each free variable x^τ in t with a value from $[\![\tau]\!]$, maps λ-terms t of type σ into elements of $[\![\sigma]\!]$.

The environment ρ maps variables to lower sets. These lower sets represents disjunctions so if $\rho(x) = \downarrow\{a_1, a_2\} = \downarrow\{a_1\} \cup \downarrow\{a_2\}$ it means that x either satisfies the property $\downarrow\{a_1\}$ or the property $\downarrow\{a_2\}$. In the previous section we observed that we have to make the same choice of properties for the variables in the environment when analysing the sub-expressions of an expression. In the abstract interpretation this is relevant for the analysis of pairing and application, which have more than one sub-expression. Here we should analyse the sub-expressions first with x bound to $\downarrow\{a_1\}$ and then with x bound to $\downarrow\{a_2\}$ and take the combination of the results of these analyses. This is achieved by forming the union of the results obtained by analysing the expression with all possible choices of "simple" environments $[x \mapsto \downarrow\{v\}]$ for $v \in \rho(x)$.

Expressions of function type are interpreted as lower sets of functions. The application of a lower set of functions from $[\![\sigma \multimap \tau]\!]$ to an argument from $[\![\sigma]\!]$ is denoted by @ and defined by

$$\downarrow\{f_1, \ldots, f_n\} \mathbin{@} L = \bigcup_{i=1}^{n} f_i(L).$$

The interpretation of the fixed point operator is now so that it accepts a set of abstract functions, computes the fixed point of each function in turn and returns the union of these fixed points. Defined in this way it is the linear extension of the old interpretation of the fixed point. In a similar way, built-in functions like addition and multiplication are defined to be the linear extensions of their counterparts in the conjunctive abstract interpretation. The addition function plus is interpreted as follows:

$$[\![\mathsf{plus}]\!](\bigcup_{i=1}^{n} \downarrow\{(a_i, b_i)\}) = \bigcup_{i=1}^{n} \downarrow\{a_i \sqcap b_i\}$$

which captures that plus is bi-strict:

$$\begin{aligned} & [\![\mathsf{plus}]\!]\downarrow\{\{(1,0),(0,1)\}\} \\ = \;& \downarrow\{0\sqcap 1\}\cup\downarrow\{1\sqcap 0\} \\ = \;& \downarrow\{0\}\cup\downarrow\{0\} \\ = \;& \downarrow\{0\} \end{aligned}$$

Example. The new interpretation will detect that the function g from the introduction (Section 4.1) is strict. We first analyse the function f from the example in Section 4.4 and calculate $[\![\mathsf{f}]\!]$.

$$\begin{aligned} & [\![\lambda x.\mathsf{if}\ B\ \mathsf{then}\ (x,17)\ \mathsf{else}\ (17,x)]\!] \\ = \;& \downarrow\{\lambda L.\bigcup_{l\in L}[\![\mathsf{if}\ B\ \mathsf{then}\ (x,17)\ \mathsf{else}\ (17,x)]\!][x\mapsto\downarrow\{l\}]\} \\ = \;& \downarrow\{\lambda L.\bigcup_{l\in L}([\![(x,17)]\!][x\mapsto\downarrow\{l\}]\cup[\![(17,x)]\!][x\mapsto\downarrow\{l\}])\} \\ = \;& \downarrow\{\lambda L.\bigcup_{l\in L}(\downarrow\{(l,1)\}\cup\downarrow\{(1,l)\})\} \end{aligned}$$

and we thus get:

$$\begin{aligned} & [\![\mathsf{f}]\!] \mathbin{@} \downarrow\{0\} \\ = \;& \{(a,b) \mid a\in\downarrow\{0\}, b\in\downarrow\{1\}\}\cup\{(a,b) \mid a\in\downarrow\{1\}, b\in\downarrow\{0\}\} \\ = \;& \downarrow\{(0,1),(1,0)\}. \end{aligned}$$

Combining this with the interpretation of plus we get that $\mathsf{g} = \lambda x.\mathsf{plus}(\mathsf{f}(x))$ is strict:

$$\begin{aligned}
&[\![\lambda x.\mathsf{plus}(\mathsf{f}(x))]\!] \mathbin{@} {\downarrow}\{0\} \\
&= {\downarrow}\{\lambda L.\bigcup_{l\in L}[\![\mathsf{plus}(\mathsf{f}(x))]\!][x \mapsto {\downarrow}\{l\}]\} \mathbin{@} {\downarrow}\{0\} \\
&= [\![\mathsf{plus}]\!] \mathbin{@} ([\![\mathsf{f}]\!] \mathbin{@} {\downarrow}\{0\}) \\
&= [\![\mathsf{plus}]\!] \mathbin{@} {\downarrow}\{(0,1),(1,0)\} = {\downarrow}\{0\}.
\end{aligned}$$

We conclude this section with a lemma that will be of importance in the proof of the soundness of the logic. The lemma confirms that the interpretation handles disjunctions in the environment correctly.

Lemma 4.5.1 *Let e be an expression, ρ an environment defining the free variables in e and L_1 and L_2 lower sets. Then*

$$[\![e]\!]\rho[x \mapsto L_1 \cup L_2] = [\![e]\!]\rho[x \mapsto L_1] \cup [\![e]\!]\rho[x \mapsto L_2].$$

Proof. The proof is an induction over the structure of e. The cases where e is a variable or a constant are easy. For pairing we have

$$\begin{aligned}
&[\![(e_1,e_2)]\!]\rho[x \mapsto L_1 \cup L_2] \\
&= \bigcup\{[\![e_1]\!][x_i \mapsto v_i][x \mapsto l] \times [\![e_2]\!][x_i \mapsto v_i][x \mapsto l] \mid v_i \in \rho(x_i), l \in L_1 \cup L_2\} \\
&= \bigcup\{[\![e_1]\!][x_i \mapsto v_i][x \mapsto l] \times [\![e_2]\!][x_i \mapsto v_i][x \mapsto l] \mid v_i \in \rho(x_i), l \in L_1\} \\
&\quad \cup \bigcup\{[\![e_1]\!][x_i \mapsto v_i][x \mapsto l] \times [\![e_2]\!][x_i \mapsto v_i][x \mapsto l] \mid v_i \in \rho(x_i), l \in L_2\} \\
&= [\![(e_1,e_2)]\!]\rho[x \mapsto L_1] \cup [\![(e_1,e_2)]\!]\rho[x \mapsto L_2]
\end{aligned}$$

The case when e is an application is dealt with similarly. In the case of lambda abstraction we first assume that the variable abstracted over is different from x.

$$\begin{aligned}
&[\![\lambda y.e]\!]\rho[x \mapsto L_1 \cup L_2] \\
&= \bigcup {\downarrow}\{\lambda M.\bigcup_{m\in M}[\![e]\!][x_i \mapsto {\downarrow}\{v_i\}][x \mapsto {\downarrow}\{l\}][y \mapsto {\downarrow}\{m\}] \mid v_i \in \rho(x_i), l \in L_1 \cup L_2\} \\
&= \bigcup {\downarrow}\{\lambda M.\bigcup_{m\in M}[\![e]\!][x_i \mapsto {\downarrow}\{v_i\}][x \mapsto {\downarrow}\{l\}][y \mapsto {\downarrow}\{m\}] \mid v_i \in \rho(x_i), l \in L_1\} \\
&\quad \cup \bigcup {\downarrow}\{\lambda M.\bigcup_{m\in M}[\![e]\!][x_i \mapsto {\downarrow}\{v_i\}][x \mapsto {\downarrow}\{l\}][y \mapsto {\downarrow}\{m\}] \mid v_i \in \rho(x_i), l \in L_2\} \\
&= [\![\lambda y.e]\!]\rho[x \mapsto L_1] \cup [\![\lambda y.e]\!]\rho[x \mapsto L_2].
\end{aligned}$$

If $y = x$ then the interpretation of the lambda expression is independent of what

x is bound to in the environment. The rest of the cases are handled by the same technique. ■

4.6 Soundness and completeness

This section states the correspondence between the program logic and the abstract interpretation. The formulation of the correspondence will differ from the conjunctive case since the correspondence between properties and denotations is different. In the disjunctive case we chose to use the properties, represented as lower sets, as denotations. The relationship between logic and interpretation is then that the abstract interpretation of an expression is the strongest property that the logic can prove of that expression. Otherwise said: if the logic can prove a property about an expression then the abstract interpretation of the expression entails that property. The technique with which we prove this is similar to the one used in the proof of Theorem 3.3.2. We first prove that all deductions made in the logic are sound. Then we prove that the logic is complete, *i.e.*, that it can show all semantic properties of a given term.

Theorem 4.6.1 *Let e be a term of type σ and assume that Γ is an environment mapping the free variables of e to propositions. Let ρ_Γ be defined by $\rho_\Gamma(x) = [\![\Gamma(x)]\!]$. For φ a formulae belonging to $L(\sigma)$ we have:*

$$\Gamma \vdash e : \varphi \Rightarrow [\![e]\!]\rho_\Gamma \subseteq [\![\varphi]\!]$$

Proof. The proof is by induction over the structure of the derivation in the program logic, just like the proof of Theorem 3.3.2. As with that proof, this is done by checking the validity of each rule in the program logic.

The validity of the rules **Conj**, **Var** and **Taut** are straightforward. The soundness of rule **Disj** is Lemma 4.5.1. In the rule for pairing **Pair** we can assume that $[\![e_1]\!]\rho \subseteq [\![\varphi]\!]$ and $[\![e_2]\!]\rho \subseteq [\![\psi]\!]$. This implies

$$[\![(e_1, e_2)]\!]\rho = [\![e_1]\!]\rho \times [\![e_2]\!]\rho \subseteq [\![\varphi]\!] \times [\![\psi]\!] = [\![\varphi \times \psi]\!].$$

The rules for **Fst** and **Snd** follows directly from the definitions. For lambda abstraction **Abs** the assumption is that, with $\psi \in L(\tau)$

$$[\![e]\!]\rho[x^\sigma \mapsto [\![\varphi]\!]] \subseteq [\![\psi]\!]$$

The abstract interpretation is monotone in the environment so we get

$$\begin{aligned}
& [\![\lambda x^\sigma.e]\!]\rho \\
= \;& \bigcup\downarrow\{\lambda L.\bigcup_{l\in L}[\![e]\!][x_i \mapsto \downarrow\{v_i\}][x^\sigma \mapsto \downarrow\{l\}] \mid v_i \in \rho(x_i)\} \\
\subseteq \;& \bigcup\downarrow\{\lambda L.[\![e]\!][x_i \mapsto \downarrow\{v_i\}][x^\sigma \mapsto L] \mid v_i \in \rho(x_i)\} \\
\subseteq \;& \downarrow\{\lambda L.[\![e]\!]\rho[x^\sigma \mapsto L]\} \\
\subseteq \;& \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f([\![\varphi]\!]) \subseteq [\![\psi]\!]\} \qquad \text{by the assumption above.}
\end{aligned}$$

For function application the assumptions are that

$$[\![e_1]\!]\rho \subseteq \{f \mid f([\![\varphi]\!]) \subseteq [\![\psi]\!]\} \quad \text{and} \quad [\![e_2]\!]\rho \subseteq [\![\varphi]\!].$$

Since application is just interpreted as applying each function in $[\![e_1]\!]\rho$ to the lower set $[\![e_2]\!]\rho$ and taking the union of the results, this gives a set that is a subset of $[\![\psi]\!]$.

To prove the **Fix** axiom sound we must show that the interpretation $[\![\mathsf{fix}]\!]$ defines a set of higher order, linear functions that maps linear functions belonging to $[\![\varphi \to \varphi]\!]$ into the set $[\![\varphi]\!]$. The interpretation of fix is given by

$$[\![\mathsf{fix}]\!]\rho = \downarrow\{\lambda\downarrow\{f_1,\ldots,f_k\}.\bigcup_{i=1}^{k}\{\bigcup_{n=0}^{\infty} f_i^n(\{0\})\}\}$$

and the function

$$\lambda\downarrow\{f_1,\ldots,f_k\}.\bigcup_{i=1}^{k}\{\bigcup_{n=0}^{\infty} f_i^n(\{0\})\}$$

is obviously linear. When passed a set of linear functions $\{f_1,\ldots,f_k\} \subseteq [\![\varphi \to \varphi]\!]$ we know that for each f_i

$$\bigcup_{n=0}^{\infty} f_i^n(\{0\}) \subseteq [\![\varphi]\!]$$

since $\{0\} \subseteq [\![\varphi]\!]$ and taking the union of this over all the f_i still gives something smaller than $[\![\varphi]\!]$. This gives the desired property of $[\![\mathsf{fix}]\!]$. The rules for the conditional are proved sound by similar means. ■

Theorem 4.6.2 *Let e be an expression of type σ and let ρ and Γ be environments of values and formulae, respectively, so that $\rho(x) = [\![\Gamma(x)]\!]$ for all free variables x in e. Assume furthermore that $\rho(x)$ is of the form $\downarrow\{v\}$ for all x. Then for $\varphi \in L(\sigma)$*

$$[\![e]\!]\rho \subseteq [\![\varphi]\!] \;\Rightarrow\; \Gamma \vdash e : \varphi.$$

Proof. Completeness is proved by induction over the structure of the term e.

x^σ**:** Follows from the assumption on ρ and Γ.

c**:** We have $[\![c]\!]\rho = 1_{[\![\sigma]\!]} = [\![\mathtt{t}_\sigma]\!]$ and $\Gamma \vdash c : \mathtt{t}_\sigma$.

(e_1, e_2)**:** Since all variables in ρ are bound to ideals we have that the assumption can be simplified to

$$[\![(e_1, e_2)]\!]\rho = [\![e_1]\!]\rho \times [\![e_2]\!]\rho \subseteq [\![\varphi]\!].$$

Due to the Definability Theorem 4.3.4 we can find formulae φ_1 and φ_2 such that

$$[\![e_1]\!]\rho = [\![\varphi_1]\!] \qquad \text{and} \qquad [\![e_2]\!]\rho = [\![\varphi_2]\!]$$

hence by the induction hypothesis we can prove $\Gamma \vdash e_1 : \varphi_1$ and $\Gamma \vdash e_2 : \varphi_2$ and therefore $\Gamma \vdash (e_1, e_2) : \varphi_1 \times \varphi_2$. Since $[\![\varphi_1 \times \varphi_2]\!] \subseteq \varphi$ we have by completeness of the axiomatisation of $\leq$ that $\varphi_1 \times \varphi_2 \leq \varphi$ hence by the rule **Weak**

$$\Gamma \vdash (e_1, e_2) : \varphi$$

$\mathsf{fst}(e)$**:** If $[\![\mathsf{fst}(e)]\!]\rho = \{a \mid (a, b) \in [\![e]\!]\rho\} \subseteq [\![\varphi]\!]$ then $[\![e]\!]\rho \subseteq [\![\varphi]\!] \times [\![\mathtt{t}]\!] = [\![\varphi \times \mathtt{t}]\!]$ so by the induction hypothesis $\Gamma \vdash e : \varphi \times \mathtt{t}$ hence $\Gamma \vdash \mathsf{fst}(e) : \varphi$. Similar proof for snd.

$\lambda x.e$**:** Assume

$$[\![\lambda x.e]\!]\rho = \downarrow\{\lambda L. \bigcup_{l \in L} [\![e]\!]\rho[x \mapsto \downarrow\{l\}]\} \subseteq [\![\bigvee_{i \in I} (\bigwedge_{j \in J_i} \varphi_{ij} \to \psi_{ij})]\!].$$

with $\mathsf{Irr}(\varphi_{ij})$. This implies that there exists an $i \in I$ such that

$$\downarrow\{\lambda L. \bigcup_{l \in L} [\![e]\!]\rho[x \mapsto \downarrow\{l\}]\} \subseteq [\![\bigwedge_{j \in J_i} \varphi_{ij} \to \psi_{ij}]\!].$$

Since all φ_{ij} are irreducible we have for all $j \in J_i$ that $[\![e]\!]\rho[x \mapsto [\![\varphi_{ij}]\!]] \subseteq [\![\psi_{ij}]\!]$ so by induction hypothesis $\Gamma[x : \varphi_{ij}] \vdash e : \psi_{ij}$ by the the use of the rule **Abs** and from that $\Gamma \vdash \lambda x.e : \varphi_{ij} \to \psi_{ij}$ for every $j \in J_i$. By the rule **Conj** and the rule **Weak** we then derive

$$\Gamma \vdash \lambda x.e : \bigvee_{i \in I} (\bigwedge_{j \in J_i} \varphi_{ij} \to \psi_{ij}).$$

$e_1\ e_2$**:** Assume that $[\![e_1 e_2]\!]\rho = [\![e_1]\!]@[\![e_2]\!]\rho \subseteq [\![\psi]\!]$. Then, with $[\![e_1]\!]\rho = \downarrow\{f_1, \ldots, f_k\}$ and φ chosen such that $[\![e_2]\!]\rho = [\![\varphi]\!]$, the assumption tells us that $f_i([\![\varphi]\!]) \subseteq [\![\psi]\!]$ for all $i = 1, \ldots, k$. This implies that $[\![e_1]\!]\rho \subseteq [\![\varphi \to \psi]\!]$ hence by induction hypothesis we can prove $\Gamma \vdash e_1 : \varphi \to \psi$ and $\Gamma \vdash e_2 : \varphi$ so by the rule **App** we have $\Gamma \vdash e_1\ e_2 : \psi$.

fix : We have to prove that

$$\downarrow\{\lambda\downarrow\{f_1,\ldots,f_k\}.\bigcup_{i=1}^{k}\{\bigcup_{n=0}^{\infty} f_i^n(\{0\})\}\} \subseteq [\![(\varphi\to\varphi)\to\varphi]\!]$$

for arbitrary φ. This amounts to showing that the function we take the downwards closure of belongs to that set, which again amounts to showing that if $\downarrow\{f_1,\ldots,f_k\} \subseteq [\![\varphi\to\varphi]\!]$ then $\bigcup_{i=1}^{k}\{\bigcup_{n=0}^{\infty} f_i^n(\{0\})\} \subseteq [\![\varphi]\!]$. But if each $f_i \in [\![\varphi\to\varphi]\!]$ then for each $i=1,\ldots,k$ we have $\bigcup_{n=0}^{\infty} f_i^n(\{0\}) \subseteq [\![\varphi]\!]$ since $\{0\}\subseteq[\![\varphi]\!]$ and each f_i is monotone.

if b then e_1 else e_2: Assume $[\![\textsf{if } b \textsf{ then } e_1 \textsf{ else } e_2]\!] \subseteq [\![\varphi]\!]$. If $[\![b]\!]\rho = \{0\}$ then $\Gamma \vdash b : \mathtt{f}_{\mathsf{Bool}}$ and we can prove $\Gamma \vdash \textsf{if } b \textsf{ then } e_1 \textsf{ else } e_2 : \mathtt{f} \le \varphi$. Otherwise we have $[\![e_1]\!]\rho \cup [\![e_2]\!]\rho \subseteq [\![\varphi]\!]$ which implies $[\![e_1]\!]\rho \subseteq [\![\varphi]\!]$ and $[\![e_2]\!]\rho \subseteq [\![\varphi]\!]$ and by induction we have $\Gamma \vdash e_1 : \varphi$ and $\Gamma \vdash e_2 : \varphi$ so $\Gamma \vdash \textsf{if } b \textsf{ then } e_1 \textsf{ else } e_2 : \varphi$.

■

The proof of next theorem shows how the rule **Disj** enables us to extend the above results to arbitrary environments.

Theorem 4.6.3 (Completeness) *Let e be an expression of type σ and let ρ and Γ be environments of values and formulae, respectively, such that $\rho(x) = [\![\Gamma(x)]\!]$ for all free variables x in e. Then for all $\varphi \in L(\sigma)$*

$$[\![e]\!]\rho \subseteq [\![\varphi]\!] \;\Rightarrow\; \Gamma \vdash e : \varphi.$$

Proof. Assume $\Gamma = [x_1 : \bigvee_{j\in J_1}\varphi_{1j}, \ldots, x_n : \bigvee_{j\in J_n}\varphi_{nj}]$. Then for all possible choices of environments $[x_i \mapsto [\![\varphi_{ij}]\!]]$ with $j \in J_i$ we have

$$[\![e]\!][x_i \mapsto [\![\varphi_{ij}]\!]] \subseteq [\![e]\!]\rho \subseteq [\![\varphi]\!]$$

and hence by the previous theorem

$$[x_i : \varphi_{ij}] \vdash e : \varphi$$

Now fix the choice of φ_{ij} for $i = 1,\ldots,n-1$. Then

$$\forall j \in J_n \,.\, [x_1 : \varphi_1, \ldots, x_{n-1} : \varphi_{n-1}, x_n : \varphi_{nj}] \vdash e : \varphi$$

hence by the rule **Disj**

$$[x_1 : \varphi_1, \ldots, x_{n-1} : \varphi_{n-1}, x_n : \bigvee_{j\in J_n}\varphi_{nj}] \vdash e : \varphi.$$

By repeating this process for each x_i we finally obtain

$$[x_1 : \bigvee_{j \in J_1} \varphi_{1j}, \ldots, x_n : \bigvee_{j \in J_n} \varphi_{nj}] \vdash e : \varphi.$$

This concludes the proof of the completeness of the logic. ■

4.7 Relational program analysis

The need for expressing disjunctions of properties in a program analysis already emerged in the work on data flow analysis for imperative languages. We shall here summarise the notion of *independent attribute* and *relational* data flow analysis as introduced by Jones and Muchnick [JM81] and see how the ideas presented there leads to the use of the *tensor product* of lattices for constructing lattices describing elements of product type.

A typical example of a data flow analysis is the analysis for detecting whether the value of an integer variable is even or odd. In this analysis we have the set of basic properties $S = \{even, odd\}$ and the lattice of properties used in the flow analysis is the power set $\wp(S)$ ordered by inclusion. Here the element $\{even, odd\}$ is representing the disjunction of the properties $\{even\}$ and $\{odd\}$. The value of a variable at a given point in the program execution is described by an element in $\wp(S)$. The problem is how to describe a state, consisting of a vector of variables. In the independent attribute method a state is described by a vector of properties, one for each variable in the program. Thus the state is modelled by the lattice $\wp(S) \times \ldots \times \wp(S)$. Executing the command $\mathsf{x}_2 := \mathsf{x}_1$ in a state $(\{even, odd\}, \{even\})$ results in a state described by $(\{even, odd\}, \{even, odd\})$. Notice that this fails to represent that x_1 and x_2 have the same value. In the independent attribute method there are no means for expressing that two variables have the same property at a program point. This is possible in the relational method that models a state by an element of $\wp(S \times \ldots \times S)$, *i.e.*, as a relation between the basic properties. Here the state previously represented by $(\{even, odd\}, \{even\})$ would now be represented by $\{(even, even), (odd, even)\}$. Executing the command $\mathsf{x}_2 := \mathsf{x}_1$ results in the state $\{(even, even), (odd, odd)\}$ and we have now managed to capture the relationship between the value of x_1 and x_2.

Using $\wp(S \times \ldots \times S)$ as a lattice for representing relational properties works fine as long as the lattice of properties are given as a power set over some basic properties. More generally, we can consider the problem of constructing a lattice for modelling products from the lattices modelling each component. Such an operator should at least satisfy

$$\wp(S_1 \times \ldots \times S_n) \cong \wp(S_1) \star \ldots \star \wp(S_n).$$

Nielson [Nie84, Nie85] argues that the tensor product of lattices is the right operation for the generalisation. The tensor product $A \otimes B$ of two lattices can be constructed in several ways. One possibility is to construct it as a term model of a theory where the elements are terms

$$a_1 \otimes b_1 \sqcup \ldots \sqcup a_n \otimes b_n$$

where $a_i \in A$ and $b_i \in B$. The theory has equations that equate terms which represent the same information, so for example,

$$(a \otimes b_1) \sqcup (a \otimes b_2) = a \otimes (b_1 \sqcup b_2) \qquad \text{and} \qquad (a_1 \otimes b) \sqcup (a_2 \otimes b) = (a_1 \sqcup a_2) \otimes b$$

These equations formalise the fact that as long as only one component is described by a disjunction there is no loss of information by using the independent method instead of the relational. The full equational theory is shown in Nielson's thesis [Nie84, p. 105].

For practical purposes a more concrete representation of the tensor product is needed. Such a representation, based on downwards closed sets, has been used for abstract interpretation of functional languages by Nielson and Nielson [NN92] and Burn [Bur92]. Both papers use the following definition of tensor product of finite lattices:

$$L_1 \otimes L_2 = (\{Y \subseteq L_1 \times L_2 \mid Y = LC(Y) = CC_1(Y) = CC_2(Y)\}, \subseteq)$$

where

$$\begin{aligned} LC(Y) &= \downarrow Y (\text{ the downwards closure of } Y) \\ CC_1(Y) &= \{(l_1 \sqcup l_2, l) \mid (l_1, l), (l_2, l) \in Y\} \\ CC_2(Y) &= \{(l, l_1 \sqcup l_2) \mid (l, l_1), (l, l_2) \in Y\}. \end{aligned}$$

A term $a_1 \otimes b_1$ is modelled by the lower set $\downarrow\{(a_1, b_1)\}$. The closure conditions CC_1 and CC_2 are there to remove lower sets of the form $\downarrow\{(a, b_1), (a, b_2)\}$ since they represent the same information as $\downarrow\{(a, b_1 \sqcup b_2)\}$. In other words, this model chooses to represent the equivalence class containing $a \otimes b_1 \sqcup a \otimes b_2$ and $a \otimes (b_1 \sqcup b_2)$ by $a \otimes (b_1 \sqcup b_2)$.

Our interpretation of products can be seen as taking the other term as representative. We defined

$$[\![\sigma \times \tau]\!] = \mathsf{D}(\{(a, b) \mid \downarrow\{a\} \in [\![\sigma]\!], \downarrow\{b\} \in [\![\tau]\!]\}).$$

If $[\![e_1]\!]\rho = \downarrow\{a\}$ and $[\![e_2]\!]\rho = \downarrow\{b_1, b_2\}$ then $[\![(e_1, e_2)]\!]\rho = \downarrow\{(a, b_1), (a, b_2)\}$ which is the least upper bound of $\downarrow\{(a, b_1)\}$ and $\downarrow\{(a, b_2)\}$ in $[\![\sigma \times \tau]\!]$. In terms of tensor products we define the interpretation of $a \otimes b_1 \sqcup b_2$ in $[\![\sigma \times \tau]\!]$ by the interpretation of $a \otimes b_1 \sqcup a \otimes b_2$.

That

$$[\![\sigma \times \tau]\!] \cong [\![\sigma]\!] \otimes [\![\tau]\!]$$

is a direct consequence of a theorem of Bandelt [Ban80, Theorem 1.2.]. This theorem is applicable because our lattices are constructed as lower closed sets over a finite lattice. The construction used Nielson and Nielson [NN92] and Burn [Bur92] works for arbitrary lattices.

The analysis by Nielson and Nielson [NN92] is formulated for a first order combinator language. The analysis by Burn [Bur92] extends this by integrating tensor products into the framework of abstract interpretation of higher order functions. It is not as powerful as the analysis developed in this chapter because functions are still interpreted as monotone functions between lattices. This means that certain disjunctive properties modelled by our use of lower sets of monotone functions cannot be modelled. As an example consider the two functions

$$f_1 \equiv \lambda x^{\mathsf{Int}}.\lambda y^{\mathsf{Int}}.x \quad \text{and} \quad f_2 \equiv \lambda x^{\mathsf{Int}}.\lambda y^{\mathsf{Int}}.y.$$

Because least upper bounds of functions are computed pointwise when using monotone functions the abstract interpretation of an expression E that either evaluates to f_1 or f_2 is the function $\lambda x^{\mathsf{Int}}.\lambda y^{\mathsf{Int}}.x \sqcup y$. However, if the result of E is applied to two arguments, at least one of which is undefined, one of the functions will return undefined. Thus with the lower set interpretation we will detect the strictness of the function resulting from

$$(\lambda f.\lambda z.(f\ 17\ z) + (f\ z\ 17))\ E,$$

a property we cannot detect if we have to describe E by the function $\lambda x^{\mathsf{Int}}.\lambda y^{\mathsf{Int}}.x \sqcup y$.

4.8 Summary

We have extended the formal system that axiomatises properties of higher order functional programs to include disjunctions of properties and developed a model where properties are interpreted as lower sets. We extended the strictness logic based on the conjunctive properties to handle disjunctions of properties and showed how to construct an abstract interpretation that interprets expressions as lower sets. The correspondence between the logic and the abstract interpretation was established by proving that the property modelled by the abstract interpretation of an expression is the strongest property provable of the expression in the logic.

The work presented here falls in line with earlier work by Burn and Nielson on using tensor products of lattices to construct lattices that can represent disjunctive properties of elements of product types. It generalises the earlier methods by handling disjunctive properties of functions as well. The use of tensor products to represent disjunctions has earlier been justified by categorical arguments [Nie85]. One contribution of the work presented here is that the disjunctive strictness logic gives a concrete description of exactly what kind of disjunctive properties these analyses can capture.

Chapter 5

Uniform properties

5.1 Introduction

So far we have considered function types and product types. Another important class of data types is the recursively defined types such as lists and trees. The problem with analysing programs over these data types is to find a set of properties of such a data type that is small and still gives useful information. For example, listing a property for each component in such a data structure, as is done with the product type, is not feasible, since the resulting set of properties would be infinite. One way of cutting down the number of properties is to consider only *uniform* properties[1]. A property of a data structure is uniform if it only describes the content of the structure, hence *"contains an element equal to 42"* is a uniform property of a list but *"is sorted"* and *"the first element is undefined"* are not. For binary trees of integers, *"all nodes are greater than 5"* is a uniform property whereas *"is a heap"* is not. More generally, for a given property P of elements we shall consider the two uniform properties: *contains an element satisfying* P and *all elements satisfy* P.

We present a formal system for reasoning about uniform properties. The axiomatisation is adopted from that of the Plotkin powerdomain. Due to the proximity to the Plotkin powerdomain we can construct a model for the system of uniform properties based on convex sets of elements and we prove that the logical system is a sound and complete axiomatisation of the convex sets semantics. To prepare for the next chapter we also give an axiomatic description of the operations of inserting and extracting elements in structures described by uniform properties.

[1]There does not seem to exist a generally accepted definition of what a uniform property is. The characterisation given here is an attempt to take the essence of properties used in a variety of analyses for languages with recursive types.

5.2 A formal system of uniform properties

Uniform properties describe the content of a data structure, thus, from this point of view, a data structure is just a multi-set of data objects. We introduce two kinds of properties for describing non-empty sets of elements of a given type σ. If φ is a formula of type σ then $\forall\varphi$ holds for those sets where *all* elements satisfy φ. Similarly $\exists\varphi$ holds for those sets where at least one element satisfies φ. We let $\mathcal{P}(\sigma)$ denote the collection of sets of elements of type σ and define the set $L(\mathcal{P}(\sigma))$ of formulae over $\mathcal{P}(\sigma)$ as follows:

Formulae

- $\mathtt{t}, \mathtt{f} \in L(\mathcal{P}(\sigma))$
- $\dfrac{\varphi, \psi \in L(\mathcal{P}(\sigma))}{\varphi \wedge \psi, \varphi \vee \psi \in L(\mathcal{P}(\sigma))}$
- $\dfrac{\varphi \in L(\sigma)}{\forall\varphi, \exists\varphi \in L(\mathcal{P}(\sigma))}$

The entailment relation $\leq$ on uniform properties is defined by adding the following axioms and rules to the disjunctive logic defined in Section 4.2. The axiomatisation is adopted from the logical description of the Plotkin powerdomain by Abramsky [Abr91b].

Axioms

$$
\begin{array}{ll}
(\forall - \wedge) & \forall(\varphi \wedge \psi) = \forall\varphi \wedge \forall\psi \\
(\forall - \vee) & \forall(\varphi \vee \psi) \leq \forall\varphi \vee \exists\psi \\
(\exists - \wedge) & \forall\varphi \wedge \exists\psi \leq \exists(\varphi \wedge \psi) \\
(\exists - \vee) & \exists(\varphi \vee \psi) = \exists\varphi \vee \exists\psi \\
(\forall - \mathtt{t}) & \forall\mathtt{t} = \mathtt{t} \\
(\forall - \mathtt{f}) & \forall\mathtt{f} = \mathtt{f}
\end{array}
$$

Rules

$$
(\forall - \leq) \quad \frac{\varphi \leq \psi}{\forall\varphi \leq \forall\psi} \qquad (\exists - \leq) \quad \frac{\varphi \leq \psi}{\exists\varphi \leq \exists\psi}
$$

Irreducibility

- $\dfrac{\mathsf{Irr}(\varphi_i) \quad i \in I}{\mathsf{Irr}(\forall(\bigvee_{i\in I}\varphi_i) \wedge \bigwedge_{i\in I}(\exists\varphi_i))}$

The axioms and rules can be understood intuitively in terms of the content of a set. The axiom $\forall - \vee$ for example states that if all elements of a set satisfies either φ or ψ then either they all satisfy φ or there exists an element satisfying ψ. Similarly, $\exists - \wedge$ can be justified as follows: if all elements satisfy φ and there furthermore exists an element satisfying ψ then that element satisfies both φ and ψ so the set satisfies $\exists(\varphi \wedge \psi)$.

As a first example of the use of the axioms we have that

$$\forall\varphi = \forall(\mathtt{f} \vee \varphi) \leq \forall\mathtt{f} \vee \exists\varphi = \mathtt{f} \vee \exists\varphi = \exists\varphi$$

$$\mathtt{t} = \forall\mathtt{t} = \forall(\mathtt{f} \vee \mathtt{t}) \leq \forall\mathtt{f} \vee \exists\mathtt{t} = \mathtt{f} \vee \exists\mathtt{t} = \exists\mathtt{t}$$

proving the theorems

$$(\forall - \exists) \quad \forall\varphi \leq \exists\varphi \qquad\qquad (\exists - \mathtt{t}) \quad \exists\mathtt{t} = \mathtt{t}$$

Formulated in words these theorems state that *"if all elements in a set satisfy φ then there exists an element in that set satisfying φ"* and *"there exists an element in the set satisfying* $\mathtt{t}$*"*. Neither of these statements hold for the empty set $\emptyset$ so we see that the sets we are axiomatising are *non-empty*. The empty set can be accommodated by removing the axiom $(\forall - \mathtt{f})$, resulting in a logical description of "the Plotkin powerdomain with the empty set adjoined". This is explained in depth by Abramsky [Abr91a].

As was the case for the disjunctive logic we have a normal form theorem for the formal system of uniform properties.

Theorem 5.2.1 (Normal Form) *Every formulae in $L(\mathcal{P}(\sigma))$ is provably equal to a finite disjunction of formulae of the form*

$$\forall(\bigvee_{i=1}^{n} \varphi_i) \wedge \bigwedge_{i=1}^{n} (\exists\varphi_i)$$

where all $\varphi_i \in \mathsf{Irr}(\sigma)$.

Proof. This is proved in detail by Abramsky [Abr91b, Proposition 3.4.8] so we only give a sketch of the proof here. By distributivity we can bring any formula from $L(\mathcal{P}(\sigma))$ into the form

$$\bigvee(\bigwedge \forall(\chi') \wedge \bigwedge \exists\chi'').$$

where χ' and χ'' belong to $\mathcal{L}(\sigma)$.

By $\forall - \wedge$ this can be transformed into

$$\bigvee(\forall(\bigwedge \chi') \wedge \bigwedge \exists \chi'')$$

and by the Normal Form Theorem for the underlying types there exists irreducibles φ, ψ such that this is equivalent to

$$\bigvee(\forall(\bigvee \varphi) \wedge \bigwedge \exists \bigvee \psi)$$

Using $\exists - \vee$ and distributivity we obtain a formula of the form

$$\bigvee(\forall(\bigvee \varphi) \wedge \bigwedge \exists \psi). \qquad (*)$$

By repeated application of the equivalences

$$\forall(\varphi_1 \vee \varphi_2) = \forall\varphi_2 \vee (\forall(\varphi_1 \vee \varphi_2) \wedge \exists\varphi_1)$$
$$\forall\varphi_1 \wedge \exists\varphi_2 = \forall\varphi_1 \wedge \exists(\varphi_1 \wedge \varphi_2)$$

we can rewrite $(*)$ such that to every φ there exists a ψ such that $\psi \sqsubseteq \varphi$ and to every ψ we can find a φ such that $\psi \sqsubseteq \varphi$. This implies that

$$\bigwedge \exists\psi = \bigwedge \exists\psi \wedge \bigwedge \exists\varphi \text{ and } \forall(\bigvee \varphi) = \forall(\bigvee \varphi \vee \bigvee \psi)$$

so $(*)$ is equivalent to

$$\bigvee(\forall(\bigvee \varphi \vee \bigvee \psi) \wedge \bigwedge \exists\varphi \wedge \bigwedge \exists\psi)$$

which is of the desired normal form.

■

An irreducible formula

$$\forall(\bigvee_i \varphi_i) \wedge \bigwedge_i (\exists\varphi_i) \qquad \{\varphi_i\} \subseteq \mathsf{Irr}(L(\sigma))$$

describes those sets where every element satisfies $\bigvee_i \varphi_i$ and for every property φ_i there exists at least one element that satisfies this property. Informally, we can also view such a formula as a very concrete description of a set. Recalling that irreducible formulae correspond to join-irreducible elements in the model, such a formula describes the finite set that contains the elements corresponding to φ_i and only these elements. In the next section we show how this intuition leads to a model for the logic $\mathcal{L}(\mathcal{P}(\sigma))$.

5.3 Convex sets

We now proceed to find a model for the formal system of uniform properties. Since the uniform properties in $\mathcal{L}(\mathcal{P}(\sigma))$ are intended to describe sets of elements of type σ, we look for a model based on sets. The irreducible formulae of $\mathcal{L}(\mathcal{P}(\sigma))$ are of the form

$$\forall(\bigvee \varphi_i) \wedge \bigwedge \exists\varphi_i \qquad \text{with } \mathsf{Irr}(\varphi_i)$$

and as each irreducible $\varphi_i \in L(\sigma)$ corresponds to a join-irreducible element in $[\![\sigma]\!]$ this suggests that a model can be built based on finite subsets of $\mathcal{J}([\![\sigma]\!])$.

First we need to find an ordering on finite sets that reflects the entailment ordering between the uniform properties represented by the sets. The theory of powerdomains offers several ways of ordering sets of elements according to the kind of information the sets are modelling [GS90]. Given two sets $\{\varphi_i\}_{i\in I}, \{\psi_j\}_{j\in J} \subseteq \mathsf{Irr}(\sigma)$ we have

$$\forall(\bigvee_{i\in I} \varphi_i) \wedge \bigwedge_{i\in I} \exists\varphi_i \leq \forall(\bigvee_{j\in J} \psi_j) \wedge \bigwedge_{j\in J} \exists\psi_j$$

if and only if

- $\bigvee_{i\in I} \varphi_i \leq \bigvee_{j\in J} \psi_j$ and
- $\forall j \in J\ \exists i \in I\ .\ \varphi_i \leq \psi_j$

i.e., if and only if

- $\forall i \in I\ \exists j \in J\ .\ \varphi_i \leq \psi_j$ and
- $\forall j \in J\ \exists i \in I\ .\ \varphi_i \leq \psi_j$.

This last condition gives rise to the *Egli-Milner ordering* on subsets of a partial order.

Definition 5.3.1 *The Egli-Milner ordering $\sqsubseteq_{EM}$ on non-empty subsets of a partial order $(P, \sqsubseteq)$ is defined by*

$$S \sqsubseteq_{EM} T \equiv \forall s \in S\ \exists t \in T\ .\ s \sqsubseteq t \quad \textit{and} \quad \forall t \in T\ \exists s \in S\ .\ s \sqsubseteq t.$$

Furthermore define

$$S \approx_{EM} T \equiv S \sqsubseteq_{EM} T \textit{ and } T \sqsubseteq_{EM} S$$

The Egli-Milner ordering is in general only a preorder on subsets. As an example take the partial order $\mathbf{3} = \{0 \sqsubseteq 1 \sqsubseteq 2\}$, where we have $\{0,2\} \sqsubseteq_{EM} \{0,1,2\}$ and $\{0,1,2\} \sqsubseteq_{EM} \{0,2\}$. In order to obtain a partial order we consider only *convex* sets.

Definition 5.3.2 *A subset S of a partial order P is* convex *if*

$$\forall s_1, s_2 \in S\ \forall p \in P. s_1 \sqsubseteq p \sqsubseteq s_2 \Rightarrow p \in S$$

The convex closure *of a non-empty set S is the least (with respect to $\subseteq$) convex set that contains S. We write this set as $\overline{S}$.*

It is straightforward to see that for a subset S we always have $S \sqsubseteq_{EM} \overline{S}$ and $\overline{S} \sqsubseteq_{EM} S$ From this it follows that the partial order of convex sets, ordered by $\sqsubseteq_{EM}$, is order-isomorphic to the preorder of subsets ordered by $\sqsubseteq_{EM}$, quotiented by $\approx_{EM}$

In the chapter on disjunctive properties we defined a partial order interpretation of types such that $[\![\sigma]\!]$ is isomorphic to the Lindenbaum algebra of $\mathcal{L}(\sigma)$. In this model the irreducible properties correspond to the join-irreducible elements $\mathcal{J}([\![\sigma]\!])$. We now construct a lattice $[\![\mathcal{P}(\sigma)]\!]$ for modelling the properties $\mathcal{L}(\mathcal{P}(\sigma))$ as the Birkhoff completion of a partial order of convex sets. The convex sets will be subsets of $\mathcal{J}([\![\sigma]\!])$ such that if $a_i \in \mathcal{J}([\![\sigma]\!])$ models irreducible $\varphi_i \in L(\sigma)$ then

$$\forall(\bigvee_{i \in I} \varphi_i) \wedge \bigwedge_{i \in I} \exists \varphi_i$$

will be modelled by the convex closure of $\{a_i \mid i \in I\}$. The following series of lemmas show that $[\![\mathcal{P}(\sigma)]\!]$, ordered by the Egli-Milner ordering, is isomorphic to the Lindenbaum algebra $\mathcal{LA}(\mathcal{P}(\sigma))$.

Definition 5.3.3 *For P a partial order define $P_C(P)$, the partial order of convex sets of P, by*

$$P_C(P) = (\{\overline{S} \mid \emptyset \neq S \subseteq P\}, \sqsubseteq_{EM}).$$

Note that all the sets in $P_C(P)$ are non-empty.

We can then define the lattice modelling $L(\mathcal{P}(\sigma))$ as follows:

Definition 5.3.4 $[\![\mathcal{P}(\sigma)]\!] = \mathsf{D}(P_C(\mathcal{J}([\![\sigma]\!])))$.

The formulae of uniform properties in $\mathcal{P}(\sigma)$ are interpreted as denoting lower subsets of $P_C(\mathcal{J}(\sigma))$ by extending the semantic function $[\![\ ]\!]_\sigma$ to the type $\mathcal{P}(\sigma)$ as shown in Figure 5.1. The trivial property `t` here denotes the whole collection of convex sets and `f` is modelled by the downwards closure of the convex set $\{0_{[\![\sigma]\!]}\}$ which is just the singleton set $\{\{0_{[\![\sigma]\!]}\}\}$.

Define the function $[\![\ ]\!]_{\mathcal{P}(\sigma)} : L(\mathcal{P}(\sigma)) \to \mathsf{D}(P_C(\mathcal{J}(\sigma)))$ by

$$\begin{array}{lcl}
[\![\mathtt{t}]\!]_{\mathcal{P}(\sigma)} & = & P_C(\mathcal{J}(\sigma)) \\
[\![\mathtt{f}]\!]_{\mathcal{P}(\sigma)} & = & \{\{0_{\mathcal{J}(\sigma)}\}\} \\
[\![\forall\varphi]\!]_{\mathcal{P}(\sigma)} & = & \{C \in P_C(\mathcal{J}(\sigma)) \mid \forall c \in C \,.\, c \in [\![\varphi]\!]_\sigma\} \\
 & = & \{C \in P_C(\mathcal{J}(\sigma)) \mid C \subseteq [\![\varphi]\!]_\sigma\} \\
[\![\exists\varphi]\!]_{\mathcal{P}(\sigma)} & = & \{C \in P_C(\mathcal{J}(\sigma)) \mid \exists c \in C \,.\, c \in [\![\varphi]\!]_\sigma \neq \emptyset\} \\
 & = & \{C \in P_C(\mathcal{J}(\sigma)) \mid C \cap [\![\varphi]\!]_\sigma \neq \emptyset\} \\
[\![\psi_1 \wedge \psi_2]\!]_{\mathcal{P}(\sigma)} & = & [\![\psi_1]\!]_{\mathcal{P}(\sigma)} \cap [\![\psi_2]\!]_{\mathcal{P}(\sigma)} \\
[\![\psi_1 \vee \psi_2]\!]_{\mathcal{P}(\sigma)} & = & [\![\psi_1]\!]_{\mathcal{P}(\sigma)} \cup [\![\psi_2]\!]_{\mathcal{P}(\sigma)}
\end{array}$$

Figure 5.1: Semantics of uniform formulae

An essential fact of the logics presented earlier is that irreducible formulae are modelled by downwards closures of a single, join-irreducible element. The first lemma shows that this is still the case for $L(\mathcal{P}(\sigma))$.

Lemma 5.3.5 *Let* $\{\varphi_i\}_{i\in I} \subseteq \mathsf{Irr}(L(\sigma))$. *Then*

$$[\![\forall(\bigvee_i \varphi_i) \wedge \bigwedge_i (\exists\varphi_i)]\!]_{\mathcal{P}(\sigma)} = {\downarrow}\{\overline{\{a_i \mid {\downarrow}\{a_i\} = [\![\varphi_i]\!]_\sigma\}}\}.$$

Proof. With ${\downarrow}\{a_i\} = [\![\varphi_i]\!]_\sigma$ we have

$$\begin{array}{ll}
 & [\![\forall(\bigvee_i \varphi_i) \wedge \bigwedge_i (\exists\varphi_i)]\!]_{\mathcal{P}(\sigma)} \\
= & \{C \in P_C(\mathcal{J}(\sigma)) \mid C \subseteq [\![\bigvee_i \varphi_i]\!]_\sigma \text{ and } \forall i.C \cap [\![\varphi_i]\!]_\sigma \neq \emptyset\} \\
= & \{C \in P_C(\mathcal{J}(\sigma)) \mid C \subseteq {\downarrow}\{a_i \mid i \in I\} \text{ and } \forall i.C \cap {\downarrow}\{a_i\} \neq \emptyset\} \\
= & \{C \in P_C(\mathcal{J}(\sigma)) \mid C \sqsubseteq_{EM} \overline{\{a_i \mid i \in I\}}\}
\end{array}$$

■

Lemma 5.3.6 (Definability) *The function* $[\![\ ]\!]_{\mathcal{P}(\sigma)} : L(\mathcal{P}(\sigma)) \to \mathsf{D}(P_C(\mathcal{J}(\sigma)))$ *is surjective.*

Proof. For every L a lower set of $P_C(\mathcal{J}(\sigma))$ there exist convex sets $C_1, \ldots, C_n$ such that $L = {\downarrow}\{C_1, \ldots, C_n\} = {\downarrow}\{C_1\} \cup \ldots \cup {\downarrow}\{C_n\}$ To each $c \in C_i$ there is an irreducible

φ_{ji} such that $[\![\varphi_{ji}]\!]_\sigma = \downarrow\{c\}$ and from the lemma above we get that

$$[\![\bigvee_j \forall(\bigvee_i \varphi_{ji}) \wedge \bigwedge_i (\exists \varphi_{ji})]\!]_{\mathcal{P}(\sigma)} = \downarrow\{C_1\} \cup \ldots \cup \downarrow\{C_n\} = \downarrow\{C_1, \ldots, C_n\}$$

■

The following two lemmas are the usual Soundness and Completeness Theorems for the axiomatisation and its model. The structure of the proofs is the same as for the proofs of soundness and completeness in the earlier chapters.

Lemma 5.3.7 $\varphi \leq \psi \Rightarrow [\![\varphi]\!]_{\mathcal{P}(\sigma)} \subseteq [\![\psi]\!]_{\mathcal{P}(\sigma)}$.

Proof. By induction over the derivation of $\varphi \leq \psi$. This amounts to checking each axiom and rule defining $\leq$. For example we verify the axiom

$$(\forall - \wedge) \qquad \forall(\varphi \wedge \psi) = \forall\varphi \wedge \forall\psi$$

by noting that

$$[\![\forall(\varphi \wedge \psi)]\!]_{\mathcal{P}(\sigma)} = \{C \mid C \subseteq [\![\varphi]\!]_\sigma \cap [\![\psi]\!]_\sigma\} = \{C \mid C \subseteq [\![\varphi]\!]_\sigma\} \cap \{C \mid C \subseteq [\![\psi]\!]_\sigma\}.$$

To verify the axiom $(\forall - \vee)$ we first note that

$$[\![\forall(\varphi \vee \psi)]\!]_{\mathcal{P}(\sigma)} = \{C \mid C \subseteq [\![\varphi]\!]_\sigma \cup [\![\psi]\!]_\sigma\}.$$

Such a C is either fully contained in $[\![\varphi]\!]_\sigma$ or it intersects $[\![\psi]\!]_\sigma$ *i.e.*, such a C belongs to

$$\{C \mid C \subseteq [\![\varphi]\!]_\sigma \text{ or } C \cap [\![\psi]\!]_\sigma \neq \emptyset\} = [\![\forall\varphi \vee \exists\psi]\!]_{\mathcal{P}(\sigma)}.$$

The rest of the axioms are verified in a similar manner. The soundness of the rules

$$\frac{\varphi \leq \psi}{\forall\varphi \leq \forall\psi} \quad \text{and} \quad \frac{\varphi \leq \psi}{\exists\varphi \leq \exists\psi}$$

follows from the observation that if $[\![\varphi]\!] \subseteq [\![\psi]\!]$ then a set contained in $[\![\varphi]\!]$ is also contained in $[\![\psi]\!]$ and a set intersecting $[\![\varphi]\!]$ must also intersect $[\![\psi]\!]$. ■

Lemma 5.3.8 $[\![\varphi]\!]_{\mathcal{P}(\sigma)} \subseteq [\![\psi]\!]_{\mathcal{P}(\sigma)} \Rightarrow \varphi \leq \psi$.

Proof. The lemma is proved by induction over the type structure. The base cases were dealt with in earlier chapters, so here we can assume that the lemma holds

for $[\![\ ]\!]_\sigma$ and $\leq_\sigma$. We prove this for irreducible φ, ψ first. Let $\downarrow\{a_i\} = [\![\varphi_i]\!]_\sigma$ and $\downarrow\{b_j\} = [\![\psi_j]\!]_\sigma$. Then

$$\begin{aligned}
&[\![\forall(\bigvee_{i\in I}\varphi_i) \wedge \bigwedge_{i\in I}(\exists\varphi_i)]\!]_{\mathcal{P}(\sigma)} \subseteq [\![\forall(\bigvee_{j\in J}\psi_j) \wedge \bigwedge_{j\in J}(\exists\psi_j)]\!]_{\mathcal{P}(\sigma)} \\
\Leftrightarrow\ & \downarrow\{\overline{\{a_i \mid i \in I\}}\} \subseteq \downarrow\{\overline{\{b_j \mid j \in J\}}\} \\
\Leftrightarrow\ & \overline{\{a_i \mid i \in I\}} \sqsubseteq_{EM} \overline{\{b_j \mid j \in J\}} \\
\Leftrightarrow\ & \downarrow\{a_i \mid i \in I\} \subseteq \downarrow\{b_j \mid j \in J\} \text{ and } \forall j\ \exists i\ .\ a_i \sqsubseteq b_j \\
\Leftrightarrow\ & [\![\bigvee_i \varphi_i]\!]_\sigma \subseteq [\![\bigvee_j \psi_j]\!]_\sigma \text{ and } \forall j\ \exists i\ .\ [\![\varphi_i]\!]_\sigma \subseteq [\![\psi_j]\!]_\sigma
\end{aligned}$$

By induction hypothesis we can deduce that

$$\begin{aligned}
&\bigvee_i \varphi_i \leq \bigvee_j \psi_j \text{ and } \forall j\ \exists i\ .\ \varphi_i \leq \psi_j \\
\Rightarrow\ & \forall(\bigvee_i \varphi_i) \leq \forall(\bigvee_j \psi_j) \text{ and } \forall j\ \exists i\ .\ \exists\varphi_i \leq \exists\psi_j \\
\Rightarrow\ & \forall(\bigvee_i \varphi_i) \wedge \bigwedge_i \exists\varphi_i \leq \forall(\bigvee_j \psi_j) \wedge \bigwedge_j \exists\psi_j
\end{aligned}$$

For general formulae we can argue precisely as we did in the proof of Theorem 4.3.6 when we proved completeness of the formal system from completeness for irreducible properties. ■

The following theorem is then an immediate result of the preceeding lemmas.

Theorem 5.3.9 *The function* $[\![\ ]\!]_{\mathcal{P}(\sigma)} : L(\mathcal{P}(\sigma)) \to \mathsf{D}(P_C(\mathcal{J}(\sigma)))$ *induces an isomorphism between the Lindenbaum algebra* $\mathcal{LA}(\mathcal{P}(\sigma))$ *and the lower subsets of* $P_C(\mathcal{J}(\sigma))$*:*

$$\mathcal{LA}(\mathcal{P}(\sigma)) \equiv (\mathcal{L}(\mathcal{P}(\sigma)/=, \leq/=) \cong \mathsf{D}(P_C(\mathcal{J}(\sigma))).$$

The theorem enables us to give a model-theoretic proof of the following equivalence:

Corollary 5.3.10 *Let* $J \subseteq I$ *and assume* $\varphi_i \in \mathsf{Irr}(\sigma)$ *for all* $i \in I$*. Then*

$$\forall \bigvee_{i\in I}\varphi_i \wedge \bigwedge_{i\in J}\exists\varphi_i = \bigvee_{J\subseteq I'\subseteq I} \forall(\bigvee_{i\in I'}\varphi_i) \wedge \bigwedge_{i\in I'}\exists\varphi_i.$$

Proof. We show

$$[\![\forall \bigvee_{i\in I}\varphi_i \wedge \bigwedge_{i\in J}\exists\varphi_i]\!] = [\![\bigvee_{J\subseteq I'\subseteq I} \forall(\bigvee_{i\in I'}\varphi_i) \wedge \bigwedge_{i\in I'}\exists\varphi_i]\!].$$

and use the theorem (more precisely Lemma 5.3.8) to arrive at the conclusion of the

Corollary. Let $a_i \in \mathcal{J}(\sigma)$ be given by $\downarrow\{a_i\} = [\![\varphi_i]\!]$.

$\supseteq$. Enough to show that for any I' such that $J \subseteq I' \subseteq I$ we have $\overline{\{a_i \mid i \in I'\}} \in [\![\forall \bigvee_{i\in I} \varphi_i \wedge \bigwedge_{i\in J} \exists\varphi_i]\!]$. This follows from

$$\overline{\{a_i \mid i \in I'\}} \subseteq [\![\bigvee_{i\in I} \varphi_i]\!] = \bigcup_{i\in I} \downarrow\{a_i\}$$

and

$$\forall j \in J \,.\, \overline{\{a_i \mid i \in I'\}} \cap \downarrow\{a_j\} \neq \emptyset.$$

$\subseteq$ Assume that C convex satisfies

$$C \subseteq [\![\bigvee_{i\in I} \varphi]\!] = \bigcup_{i\in I} \downarrow\{a_i\}$$

and let I' be the biggest set of indices such that for every $i \in I'$ there exists a $c \in C$ such that $c \sqsubseteq a_i$. Since $C \cap \downarrow\{a_i\} \neq \emptyset$ for all $i \in J$ we have that $J \subseteq I'$ which means that $C \subseteq [\![\bigvee_{i\in I'} \varphi]\!]$ and $C \cap [\![\varphi_i]\!] \neq \emptyset$ for all $i \in J$. ■

In view of the correspondence between the irreducible formulae φ_i and the elements a_i we can motivate the corollary as follows: If we know of a set S that it contains at most the elements $\{a_i \mid i \in I\}$ and it contains at least $\{a_i \mid i \in J\}$ then we know it is one of the sets S satisfying $\{a_i \mid i \in J\} \subseteq S \subseteq \{a_i \mid i \in I\}$.

5.4 Examples

The interpretation of the type Int is the lower sets of the lattice $\mathcal{J}(\mathsf{Int}) = \{0 \sqsubseteq 1\}$ so the set of convex sets $\mathcal{J}(\mathcal{P}(\mathsf{Int}))$ is given by

$$\mathcal{J}(\mathcal{P}(\mathsf{Int})) = \{0\} \sqsubseteq_{EM} \{0,1\} \sqsubseteq_{EM} \{1\}$$

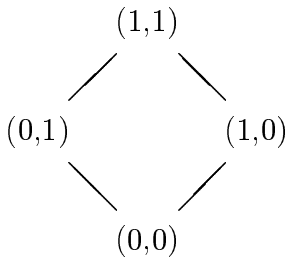
This is a linearly ordered set so the lattice of lower sets of $\mathcal{J}(\mathcal{P}(\mathsf{Int}))$ is isomorphic to $\mathcal{J}(\mathcal{P}(\mathsf{Int}))$. It is shown in Figure 5.2. Here the top element $\downarrow\{\{1\}\}$ contains no information; it just says that all elements satisfy the trivial property. Likewise the smallest element $\downarrow\{\{0\}\}$ states that all elements satisfies the strongest property (*e.g.*, the property "completely undefined" in strictness analysis or "static" in binding time analysis). The middle point $\downarrow\{\{0,1\}\}$ is the interpretation of the formula $\forall(\mathtt{t} \vee \mathtt{f}) \wedge \exists\mathtt{t} \wedge \exists\mathtt{f} = \forall\mathtt{t} \wedge \exists\mathtt{f}$. The minimal element 0 models the $\exists$ part and the maximal element 1 the $\forall$ part of the formula. To see the difference between the ordering $\sqsubseteq_{EM}$ and the subset ordering notice that by removing 1 from $\downarrow\{\{0,1\}\}$ we

$$\begin{array}{rcl} \downarrow\{\{1\}\} & = & \{\{0\},\{0,1\},\{1\}\} \\ \quad | & & \\ \downarrow\{\{0,1\}\} & = & \{\{0\},\{0,1\}\} \\ \quad | & & \\ \downarrow\{\{0\}\} & = & \{\{0\}\} \end{array}$$

Figure 5.2: The lattice $[\![\mathcal{P}(\mathsf{Int})]\!] = \mathsf{D}(P_C(\mathcal{J}(\mathsf{Int})))$

get a set that is smaller in the $\sqsubseteq_{EM}$ ordering, whereas by removing 0 from $\downarrow\{\{0,1\}\}$ we get a set that is bigger.

More interesting convex sets arise from the type $\mathsf{Int} \times \mathsf{Int}$. The lattice $\mathcal{J}(\mathsf{Int} \times \mathsf{Int}) = \mathcal{J}(\mathsf{Int}) \times \mathcal{J}(\mathsf{Int})$ is the diamond-shaped lattice



The whole lattice is of course a convex set and corresponds to the formula

$$\forall(\mathtt{t} \times \mathtt{t}) \wedge \exists(\mathtt{f} \times \mathtt{f}).$$

A formula in normal form equivalent to this formula is

$$\forall(\mathtt{t} \times \mathtt{t} \vee \mathtt{f} \times \mathtt{f}) \wedge \exists(\mathtt{f} \times \mathtt{f}) \wedge \exists(\mathtt{t} \times \mathtt{t}).$$

The property that all pairs in a set have at least one component that is modelled by 0 would itself be modelled by the downwards closure of the convex set $\{(0,1),(1,0)\}$. It corresponds to the formula

$$\forall(\mathtt{t} \times \mathtt{f} \vee \mathtt{f} \times \mathtt{t}) \wedge \exists(\mathtt{t} \times \mathtt{f}) \wedge \exists(\mathtt{f} \times \mathtt{t}).$$

In the Egli-Milner ordering we have $\{(0,1),(1,0),(0,0)\} \sqsubseteq_{EM} \{(0,1),(1,0)\}$. The information content of the set $\{(0,1),(1,0)\}$ differs from that of $\{(0,1),(1,0),(0,0)\}$ in that the latter requires that there exists an element $(0,0)$ in the set.

5.5 Operations on sets

As an introduction to the later axiomatisation of the operations on recursive data structures we define two operations on sets: *insert* that adds an element to a set and *extract* that picks an element from a set, and provide an axiomatic description of these operators using the formal system $\mathcal{L}(\mathcal{P}(\sigma))$. The two operators have type

$$\begin{aligned} \mathit{insert} &: \sigma \times \mathcal{P}(\sigma) \to \mathcal{P}(\sigma) \\ \mathit{extract} &: \mathcal{P}(\sigma) \to (\sigma \times \mathcal{P}(\sigma)) \end{aligned}$$

The following two inference rules define the logic for reasoning about *insert* and *extract*. Since this is only meant to illustrate the use of the formal system, we shall not state the rules in their full generality. We assume that there is a finite set $\{\varphi_j \mid j \in J\} \subseteq \mathsf{Irr}(L(\sigma))$ and that $\psi \in \mathsf{Irr}(L(\sigma))$.

$$\frac{d : \psi \quad S : \forall(\bigvee_j \varphi_j) \wedge \bigwedge_j \exists\varphi_j)}{\mathit{insert}(d, S) : \forall(\bigvee_j \varphi_j \vee \psi) \wedge \exists\psi \wedge \bigwedge_j \exists\varphi_j}$$

$$\frac{S : \forall(\bigvee_j \varphi_j) \wedge \bigwedge_j \exists\varphi_j}{\mathit{extract}(S) : \bigvee_{I \subseteq J, I \neq \emptyset} (\bigwedge_{i \in I} \varphi_i) \times \forall(\bigvee_{j \in J} \varphi_j) \wedge \bigwedge_{j \in J \setminus I} \exists\varphi_j}$$

To motivate the rule for the function *insert* let us assume that we have an element d satisfying ψ and a structure M satisfying $\forall \bigvee \varphi_i \wedge \bigwedge \exists\varphi_i$ and we seek a property describing $\mathit{insert}(d, M)$. Since d satisfies ψ and every element in M satisfies at least one of the φ_i, we can conclude that that every elements in $\mathit{insert}(d, M)$ satisfy $\psi \vee \bigvee \varphi_i$. Similarly, in M there exists an element satisfying φ_i for each i so in $\mathit{insert}(d, M)$ there exists, in addition to this, an element satisfying ψ. Thus M satisfies $\exists\psi \wedge \bigwedge \exists\varphi_i$.

The rule for *extract* is slightly more complicated. Assume that the multi-set M satisfies

$$\forall(\varphi_1 \vee \ldots \vee \varphi_n) \wedge \exists\varphi_1 \wedge \ldots \wedge \exists\varphi_n$$

and that $\mathit{extract}(M) = (d, M')$. The most precise property we can attribute to d in isolation is $\varphi_1 \vee \ldots \vee \varphi_n$. Considering the remaining part of the structure, M', we can only say that it satisfies $\forall(\varphi_1 \vee \ldots \vee \varphi_n)$ since d can be the only element that makes M satisfy any of the $\exists\varphi_i$ formula. However, by considering d and M' together we can do better. The element d makes a certain number of the $\exists\varphi_i$ true but the rest of the $\exists\varphi_i$ are due to elements in the remaining structure M'. Thus for

one non-empty subset $I \subseteq \{1, \ldots, n\}$ we have

$$d : \bigwedge_{i \in I} \varphi_i \quad \text{and} \quad S' : \bigwedge_{i \in \{1,\ldots,n\} \setminus I} \exists \varphi_i .$$

Since it cannot be determined for which subset I this holds, we have to form the disjunction of all such properties, indexed by the non-empty subsets of $\{1, \ldots, n\}$ (non-empty since d satisfies at least one of the properties).

The general rules would cover the cases where the formulae in the assumptions were disjunctions of the kind of formulae we consider here. For example, the rule for fold would have as premises

$$d : \bigvee_j \psi_j \quad \text{and} \quad S : \bigvee_i (\forall (\bigvee_j \varphi_{ij}) \wedge \bigwedge_j \exists \varphi_{ij})$$

These more general cases are accommodated by analysing each disjunct using the above rules and forming the disjunction of the results so obtained.

The logic for *insert* and *extract* can be modelled by functions over the lattices modelling the uniform properties. We define the two functions

$$[\![insert]\!] : [\![\sigma \times \mathcal{P}(\sigma)]\!] \multimap [\![\mathcal{P}(\sigma)]\!]$$
$$[\![extract]\!] : [\![\mathcal{P}(\sigma)]\!] \multimap [\![(\sigma \times \mathcal{P}(\sigma))]\!]$$

by giving their values on join-irreducibles:

$$[\![insert]\!](\downarrow\{(s, C)\}) = \downarrow\{\overline{C \cup \{s\}}\}$$
$$[\![extract]\!](\downarrow\{C\}) = \downarrow\{(\sqcap S, C') \mid C' \text{ convex}, S \subseteq C \;\&\; C \setminus S \subseteq C' \subseteq C\}$$

The rule for *extract* illustrates a problem with removing elements from a convex since each element can represent $\forall$-properties as well $\exists$-properties and we should only remove the elements describing $\exists$-properties (*cf.* the discussion above). In the rule for *extract* we specify that we should take the maximal (with respect to $\sqsubseteq_{EM}$) convex sets, but we have not been able to find a formula that calculates exactly these maximal sets.

We prove two lemmas that show that the logic for *insert* and *extract* is sound and complete with respect to the abstract functions $[\![insert]\!]$ and $[\![extract]\!]$.

Lemma 5.5.1 *Assume* $\mathsf{Irr}(\varphi)$ *and* $\mathsf{Irr}(\varphi_j)$ *for all* $j \in J$ *Then*

$$[\![insert]\!]([\![\varphi \times \forall(\bigvee_{j\in J} \varphi_i) \wedge \bigwedge_{j\in J} \exists\varphi_i]\!]) = [\![\forall(\varphi \vee \bigvee_{j\in J} \varphi_j) \wedge \exists\varphi \wedge \bigwedge_{j\in J} \exists\varphi_j]\!].$$

Proof. With $\downarrow\{a\} = [\![\varphi]\!]$ and $\downarrow\{a_i\} = [\![\varphi_i]\!]$ for all $j \in J$, we let C be the convex closure $\overline{\{a_j \mid j \in J\}}$. Then $\downarrow\{C\} = [\![\forall(\bigvee_{j\in J} \varphi_j) \wedge \bigwedge_{j\in J} \exists\varphi_j]\!]$ and the lemma follows from the observation

$$\overline{C \cup \{a\}} = \overline{\{a_j \mid j \in J\} \cup \{a\}} = [\![\forall(\varphi \vee \bigvee_{j\in J} \varphi_j) \wedge \exists\varphi \wedge \bigwedge_{j\in J} \exists\varphi_j]\!].$$

■

Lemma 5.5.2 *With* $\downarrow\{a_i\} = [\![\varphi_i]\!]$ *for all* $j \in J$ *and* $C = \downarrow\{a_j \mid j \in J\}$ *we have that*

$$[\![\bigvee_{\emptyset\neq I\subseteq J} (\bigwedge_{i\in I} \varphi_i) \times \forall(\bigvee_{j\in J} \varphi_j) \wedge \bigwedge_{j\in J\setminus I} \exists\varphi_j]\!] = \bigcup_{\emptyset\neq S\subseteq C} \downarrow\{(\sqcap S, C') \mid C \setminus S \subseteq C' \subseteq C, C' \text{ convex}\}.$$

Proof. "$\supseteq$". Assume S, C' satisfy $\emptyset \neq S \subseteq C$ and $C \setminus S \subseteq C' \subseteq C$. We have to find a non-empty $I \subseteq J$ such that

$$\sqcap S \in [\![\bigwedge_{j\in I} \varphi_j]\!] \quad \& \quad C' \in [\![\forall(\bigvee_{j\in J} \varphi_j) \wedge \bigwedge_{j\in J\setminus I} \exists\varphi_j]\!].$$

By choosing $I = \{i \mid \exists s \in S.s \sqsubseteq a_i\}$ we obtain that $K \neq \emptyset$ if $S \neq \emptyset$ since every $s \in S$ is bounded above by some a_i. Furthermore

$$\sqcap S \in \bigcap_{i\in I} [\![\varphi_i]\!]$$

because for all $i \in I$ there exists an $s \in S$ such that $s \sqsubseteq a_i$ so an element smaller than all $s \in S$ must necessarily also be smaller than all a_i for $i \in I$.

We then observe that C' belongs to

$$[\![\forall(\bigvee_{j\in J} \varphi_j)]\!] = \{D \mid D \text{ convex}, D \subseteq [\![\bigvee_{j\in J} \varphi_j]\!]\}$$

since C belongs to that set and $C' \subseteq C$. Finally, for $i \in J\setminus I$ we have $a_i \in C \setminus S \subseteq C'$ so $\forall i \in J \setminus I \,.\, \downarrow\{a_i\} \cap C' \neq \emptyset$ which means that $C' \in [\![\bigwedge_{J\setminus I} \exists\varphi_i]\!]$.

"$\subseteq$". Given non-empty $I \subseteq J$ we have to find $S \subseteq C$ satisfying the two requirements

- $[\![\bigwedge_{j\in I} \varphi_j]\!] \subseteq \downarrow\{\sqcap S\}$

- $[\![\forall(\bigvee_{j\in J}\varphi_j)\wedge\bigwedge_{j\in J\setminus I}\exists\varphi_j]\!]\subseteq\downarrow\{C' \text{ convex } \mid C\setminus S\subseteq C'\subseteq C\}$.

Define $S=\uparrow\{a_i \mid i\in I\}$. Then $\sqcap S=\sqcap\{a_i \mid i\in I\}$ which proves half of the requirement on S. For the second half we use that

$$[\![\forall(\bigvee_{j\in J}\varphi_j)\wedge\bigwedge_{j\in J\setminus I}\exists\varphi_j]\!]=[\![\bigvee_{J\setminus I\subseteq I'\subseteq J}\forall(\bigvee_{j\in I'}\varphi_j)\wedge\bigwedge_{j\in I'}\exists\varphi_j]\!]$$

so we only have to prove that for given I' satisfying $J\setminus I\subseteq I'\subseteq J$ there exists a convex C' satisfying $C\setminus S\subseteq C'\subseteq C$ such that $\overline{\{a_i \mid i\in I'\}}\sqsubseteq_{EM}C'$. We have that

$$\forall c\in C\setminus S\;\exists i\in J\setminus I\;.\;a_i\sqsubseteq c$$

which implies $\{a_i \mid i\in I'\}\sqsubseteq_{EM}\{a_i \mid i\in I'\}\cup(C\setminus S)$. Since a subset is equivalent to its convex closure under the ordering $\sqsubseteq_{EM}$ we then get that

$$\overline{\{a_i \mid i\in I'\}}\sqsubseteq_{EM}\overline{\{a_i \mid i\in I'\}\cup(C\setminus S)}$$

and we have found a convex set satisfying the requirements from above. ■

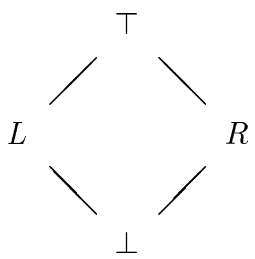
By combining this lemma and the definition of *extract* we get the following corollary:

Corollary 5.5.3 *Assume* $\mathsf{Irr}(\varphi_j)$ *for all* $j\in J$ *Then*

$$[\![extract]\!]([\![\forall(\bigvee_{j\in J}\varphi_i)\wedge\bigwedge_{j\in J}\exists\varphi_i]\!])=[\![\bigvee_{I\subseteq J,I\neq\emptyset}(\bigwedge_{i\in I}\varphi_i)\times\forall(\bigvee_{j\in J}\varphi_j)\wedge\bigwedge_{j\in J\setminus I}\exists\varphi_j]\!].$$

5.6 The cone powerdomain

A problem with using convex sets in abstract interpretation is that two convex sets do not necessarily have a least upper bound in the Egli-Milner ordering. Plotkin [Plo76] gives the following example. Let S be the lattice



and consider the product $S\times S$.

Amongst others we have the elements

$$(L, \bot) \quad (R, \bot) \quad (\bot, L) \quad (\bot, R)$$

and we can consider the two convex sets

$$\{(L, \bot), (R, \bot)\} \quad \text{and} \quad \{(\bot, L), (\bot, R)\}.$$

These two sets have the minimal upper bounds

$$\{(L, L), (R, R)\} \quad \text{and} \quad \{(L, R), (R, L)\}$$

which are incomparable.

One solution is to embed the convex sets into a lattice via Birkhoff completion. This is the approach taken here. Another solution is presented by Ferguson and Hughes [FH89] where they consider a particular kind of convex sets: the cones. A cone is defined as a convex set that is closed under least upper bound. They show that least upper bounds of convex sets exist and can be computed by the formula

$$P \sqcup Q = \{p \sqcup q \mid p \in P, q \in Q\}$$

which viewed from a logical point of view is natural, since this forms all the possible combinations of properties that can be made with one property from each set.

Do we lose anything by using cones instead of arbitrary convex sets? Ferguson and Hughes argues [FH89] that for their application *"... we do not really care about the distinction between sets ... differing only by points which are lubs of other points in the set"*. In general this is only sustainable when there is no loss of information by taking least upper bounds. For example, we lose precision if we have to convert the set $\{(1, 0), (0, 1)\}$ into the set $\{(1, 0), (0, 1), (1, 1)\}$ since we no longer know that at least one component will be undefined.

On the other hand, if $\{(1, 0), (0, 1)\}$ is converted into something like $\{(1, 0), (0, 1), (1, 0) \sqcup (0, 1)\}$ where the least upper bound operator allows us to retrieve $(1, 0), (0, 1)\}$ then the cone might just be another representation of the same information. A logical description of the cone powerdomain would be useful for comparing the two approaches, but we shall not venture deeper into this.

5.7 Summary

In this chapter we have axiomatised a class of properties called uniform properties. These properties were introduced with the purpose of describing the content of a data structure. We considered two kinds of uniform properties: *contains an element satisfying a property P* and *all elements satisfy a property P*. It can be argued that a property like *contains 17 elements equal to 42* should be called uniform as well since it only describes content. However, this would lead to infinite set of properties and we would have to find a way of cutting this down to a finite set.

We presented a model of the uniform properties based on convex sets and showed this model sound and complete with respect to the axiomatisation. The formal system and the model were then used to give an axiomatic resp. denotational description of the operations of inserting and extracting elements and we showed that these two descriptions are equivalent. In the next chapter we combine this with an axiomatisation of sum types to obtain a framework for analysing algebraic data structures.

Chapter 6

Recursive Data Structures

6.1 Introduction

The analysis of list structures has been considered by a number of people [Wad87, Lau89, EM91, HS91, Hun91, LM91, NN92]. Wadler in his original work [Wad87] suggested the following four-point domain for the analysis of programs over lists of integers. With each point in the domain we list the set of lists that the point represent.

$1\in$	all lists
$\mid$	
$0\in$	lists containing an undefined element and partial or infinite lists
$\mid$	
∞	all partial or infinite lists
$\mid$	
$\bot$	the undefined list

(Wadler writes $\top\in$ and $\bot\in$ where we write $1\in$ and $0\in$). The abstract interpretation of the list operations are defined over this domain such that *e.g.*, we have $\mathsf{cons}^{\#}\ 0\ \ 1\in = \ 0\in$ and $\mathsf{tail}^{\#}\ \infty\ =\ \infty$. In this chapter we shall use the system of uniform properties to extend Wadler's analysis to an analysis of uniform properties of recursive data structures. More precisely, we consider data structures which are

of *algebraic data type*, *i.e.*, a type of the form

$$T = \mu\alpha.D_1{\times}\alpha^{n_1} + \ldots + D_k{\times}\alpha^{n_k}.$$

Examples of this kind of data types are lists and binary trees.

After reviewing some standard material about algebraic data types we present a formal system for reasoning about properties of sum type. Based on this we define for each algebraic data type a formal system of uniform properties together with rules for reasoning about the operations on structures of such type. We then go on to develop a model for this logic based on the convex set model for uniform properties and show that this model is sound and complete with respect to the formal system. This is followed by an axiomatisation of the operations fold and unfold associated with recursively defined types together with an abstract interpretation of these operations that is shown to be a sound and complete model of the axiomatisation. We conclude with tabulating the abstract interpretations of some of the standard operations on lists and comparing them with the abstract operations defined by Wadler [Wad87].

6.2 Algebraic data types

In this section we define the syntax of the data types with which we shall be concerned. These types are often referred to as *algebraic* types, or "sum-of-products" types. They form the core of the type systems in most modern functional languages including ML and Haskell.

We assume that we are given a collection B of *base types* such as integers and boolean values. From these we build the *simple types* D by means of the type constructors $\times, +, \rightarrow$ and the one-point type $\mathbf{1}$. On top of this we construct the recursive data types T of the form $\mu\alpha.F(\alpha)$ where F is a sum of products possibly involving the recursion variable α. To keep things simple we restrict ourselves to consider types built using only one variable. The following grammar defines D and T:

$$\begin{aligned} T &= \mu\alpha.T^\alpha + \ldots + T^\alpha \\ T^\alpha &= \alpha \mid D \mid T^\alpha \times \ldots \times T^\alpha \\ D &= B \mid \mathbf{1} \mid D + D \mid D \times D \mid D \rightarrow D. \end{aligned}$$

The order of the factors in a T^α-product is irrelevant for our purposes so we shall

assume that all types are in the canonical form

$$T = \mu\alpha.D_1 \times \alpha^{n_1} + \ldots + D_k \times \alpha^{n_k} = \mu\alpha.\sum_i D_i \times \alpha^{n_i}$$

where D_i is a simple type and n_i is an integer value, possibly zero.

Examples Binary trees with boolean values at the leaves have the following canonical type

$$\mu\alpha.\texttt{Bool} + \mathbf{1} \times \alpha \times \alpha.$$

The type of lists of integers is as follows:

$$\mu\alpha.\texttt{Int} \times \alpha + \mathbf{1}.$$

This class of types is simple yet general enough to enable us to demonstrate our ideas. However, two important classes of recursively defined types are excluded by this definition: nested recursive types such as lists of lists, and types where the type variable appears as one or both of the arguments of the arrow type constructors, as in $\mu\alpha.\alpha \to \alpha$. We believe that it is fairly straightforward to generalise the principles presented here to the former kind of types but it is not clear whether a similar generalisation can be carried out for the latter kind of types.

The operations for constructing and decomposing recursive data structures are called fold and unfold. The typing rules for fold and unfold are

$$\frac{e : F(\mu\alpha.F(\alpha))}{\mathsf{fold}\ e : \mu\alpha.F(\alpha)} \qquad \frac{e : \mu\alpha.F(\alpha)}{\mathsf{unfold}\ e : F(\mu\alpha.F(\alpha))}.$$

Assume that e is of type lists of integers. If e evaluates to the list $[2,3]$ then the expression $\mathsf{fold}\ \mathsf{in}_1(1,e)$[1] evaluates to the list $[1,2,3]$. Unfolding that value would yield the tagged pair $\mathsf{in}_1(1,[2,3])$.

6.2.1 Semantics of recursive data structures

In order to find a domain that can model an algebraic data type we consider a type definition like the one above as an equation such a domain must satisfy. The standard approach to the solution of recursive domain equations $D = \mu\alpha.F(\alpha)$ in a category $\mathbf{C}$ of domains interprets the type expression F as an endofunctor on $\mathbf{C}$ and defines a solution to be a "fixed point" of F. Since F is a mapping on a category,

[1] The operator in is the sum type constructor; see Section 6.3

rather that a set, such a fixed point consists of a domain D and an isomorphism

$$\gamma : F(D) \cong D.$$

That a fixed point exists and how it can be obtained as the limit of a sequence of approximating domains is well understood but not in the scope of this thesis, so we refer to the literature [SP82, Tay86, Ten91]. The isomorphism defining the fixed point provides the semantics for the fold and the unfold operations, *i.e.*,

$$[\![\mathsf{fold}]\!] = \gamma \qquad [\![\mathsf{unfold}]\!] = \gamma^{-1}$$

For algebraic data types we interpret $\times$ and $+$ as cartesian product and separated sum of domains. Since γ is an order-isomorphism the bottom element in $R = \mu\alpha.\sum_i D_i{\times}\alpha^{n_i}$ is the only image under γ of the bottom element in the separated sum $\sum_i D_i{\times}R^{n_i}$, *i.e.*,

$$\gamma(\bot_{\sum_i D_i\times R^{n_i}}) = \bot_R \qquad \text{and} \qquad \gamma^{-1}(\bot_R) = \bot_{\sum_i D_i\times R^{n_i}}.$$

Any other element in $\mu\alpha.\sum_i D_i{\times}\alpha^{n_i}$ is the image of an element of the form $\mathsf{in}_i(d, e_1, \ldots, e_n)$ where $e_1, \ldots, e_n \in \mu\alpha.\sum_i D_i{\times}\alpha^{n_i}$.

6.2.2 Data structures as multi-sets

To provide some intuition for the way data structures are viewed here we briefly consider how their content can be described by multi-sets. The collection of elements in a data structure forms a *multi-set* since the same element can occur at several places in the structure. We define a function *flatten* to extract the multi-set of elements from a data structure.

A multi-set of elements from a set S is the same as a subset of S except for the fact that an element can occur more than once in a multi-set. Thus it is not a proper *set* since elements only occur once in a set, but rather a collection of elements from S where each element of S occurs a certain number of times. We denote by $\mathcal{M}(S)$ the set of multi-sets of S. As examples of multi-sets of natural numbers we have

$$\{1\} \quad \{1,1\} \quad \emptyset \quad \{1,2,3,4,\ldots\}$$

The operation of inserting an element in a multi-set can be defined by keeping count

of the multiplicity of each element. Thus

$$\mathit{insert}(1,\{1\}) = \{1,1\} \quad \mathit{insert}(1,\{1,1\}) = \{1,1,1\}$$

The union of two multi-sets can be defined by *insert*ing each element of one of the sets into the other. The result is a multi-set where each element occurs as many times as it occurs in the two multi-sets together. For example $\{1,1,2\} \cup \{1,2\} = \{1,1,1,2,2\}$.

Multi-sets can describe the content of an algebraic data structure. For D_i a domain we denote by $\mathcal{M}(D_i)$ the lattice of multi-sets of D_i, ordered by multi-set inclusion (*i.e.*, where multiplicity is taken into consideration). The function

$$\mathit{flatten}_i : \mu\alpha.\sum_i D_i \times \alpha^{n_i} \to \mathcal{M}(D_i)$$

extracts the set of elements belonging to the summand D_i from a data structure of type $\mu\alpha.\sum_i D_i \times \alpha^{n_i}$. The following definition of *flatten* makes use of the fact that any non-bottom element is the image of some non-bottom element under the isomorphism γ.

Definition 6.2.1 *Define the function* $\mathit{flatten}_i : \mu\alpha.\sum_i D_i \times \alpha^{n_i} \to \mathcal{M}(D_i)$ *by*

$$\begin{array}{lcl} \mathit{flatten}_i(\bot) & = & \emptyset \\ \mathit{flatten}_i(\gamma(\mathsf{in}_j(d, e_1, \ldots, e_n))) & = & \{d \mid i = j\} \cup \bigcup_{i=1}^{n} \mathit{flatten}_i(e_i) \end{array}$$

With this definition of *flatten* we have *e.g.*, that *flatten* maps the partial list $1 : 2 : \bot$ to the multi-set $\{1,2\}$ and the infinite list $[1,1,1,\ldots]$ to the multi-set $\{1,1,1,\ldots\}$ containing an infinite number of ones.

6.3 Sum types

Before we can consider recursive types we have to study the properties of sum types and provide a formal system of formulae for deducing properties of a sum type. An element of type $D_1 + D_2$ is an element of either D_1 or D_2 with a *tag* indicating which of the D_i the element is drawn from. Such an element is written $\mathsf{in}_i(d)$. More generally, we consider finite sums of types, written $\sum_{i \in I} D_i$.

We shall consider a formal system of sum types that can describe not only tagged elements, $\mathsf{in}_i(d)$, but a collection of such tagged elements. The reason for this gen-

Formulae

$$\bullet\ \mathtt{t}, \mathtt{f} \in L(\textstyle\sum_i \sigma_i) \qquad \bullet\ \dfrac{\varphi_j \in L(\sigma_j) \quad J \subseteq I}{\sum_j \mathsf{in}_j(\varphi_j) \in L(\sum_i \sigma_i)} \qquad \bullet\ \dfrac{\varphi, \psi \in L(\sum_i \sigma_i)}{\varphi \wedge \psi, \varphi \vee \psi \in L(\sum_i \sigma_i)}$$

Axioms

$$\textstyle\sum - \wedge \quad \bullet \quad \sum_{j \in J_1} \mathsf{in}_j(\varphi_j) \wedge \sum_{j \in J_2} \mathsf{in}_j(\psi_j) = \sum_{j \in J_1 \cap J_2} \mathsf{in}_j(\varphi_j \wedge \psi_j)$$

$$\textstyle\sum - \mathtt{f} \quad \bullet \quad \sum_{j \in \emptyset} \mathsf{in}_j(\varphi_j) = \mathtt{f}$$

$$\textstyle\sum - \mathtt{t} \quad \bullet \quad \sum_{j \in I} \mathsf{in}_j(\mathtt{t}) = \mathtt{t}$$

$$\textstyle\sum - \vee \quad \bullet \quad \sum_{j \in J} \mathsf{in}_j(\varphi_j) \vee \sum_{j \in J} \mathsf{in}_j(\psi_j) = \sum_{j \in J} \mathsf{in}_j(\varphi_i \vee \psi_j)$$

Rule

$$\textstyle\sum - \leq \quad \bullet \quad \dfrac{\varphi_j \leq \psi_j \qquad \forall j \in J_1 \subseteq J_2 \subseteq I}{\sum_{j \in J_1} \mathsf{in}_j(\varphi_j) \leq \sum_{j \in J_2} \mathsf{in}_j(\psi_j)}$$

Irreducibility

$$\bullet\ \dfrac{\varphi_j \in \mathsf{Irr}(\sigma_j) \qquad j \in J}{\sum_{j \in J} \mathsf{in}_j(\varphi_j) \in \mathsf{Irr}(\sum_{i \in I} \sigma_i)}$$

Figure 6.1: Axiomatisation of sum properties

eralisation is that we want to use the system to describe data structures where the elements can have different tags. With the generalised system we can list the tags occurring in the structure and for each tag give a property that the elements with this tag satisfy. The basic formulae of a sum type $\sum_{i \in I} D_i$ will be of the form $\sum_{i \in J} \mathsf{in}_i(\varphi_i)$ where $J \subseteq I$ and φ_i is a property of type D_i. Such a formula describes a collection of tagged elements where every element is drawn from one of the summands $\{D_i\}_{i \in J}$ and the elements of the j-th summand satisfies φ_j. General properties are con- and disjunctions of these basic properties.

It is important to notice that the $\sum$ connective is different from disjunction. If a collection of tagged elements satisfies the formula $\mathsf{in}_1(\varphi_1) \vee \mathsf{in}_2(\varphi_2)$ then all elements either belong to the first or the second summand; in either case they all have the same tag. In comparison, the formula $\mathsf{in}_1(\varphi_1) + \mathsf{in}_2(\varphi_2)$ does not preclude that there can be elements from both summands. It only guarantees that *if* an element comes

from the first (the second) summand then it satisfies φ_1 (φ_2). In general we have

$$\bigvee_{j\in J} \mathsf{in}_j(\varphi_j) \leq \sum_{j\in J} \mathsf{in}_j(\varphi_j)$$

but not the converse. The formal definition of the set of formulae $L(\sum_i \sigma_i)$ and its axiomatisation is shown in Figure 6.1.

Theorem 6.3.1 (Normal form for sums) *Every formula in $L(\sum_{i\in I} \sigma_i)$ is provably equivalent to a formula of the canonical form*

$$\bigvee_{k\in K} \sum_{j\in J_k} \mathsf{in}_j(\varphi_{jk})$$

where $\mathsf{Irr}(\varphi_{jk})$.

Proof. By the distributive laws we can reorganise a formula in $L(\sum_{i\in I} \sigma_i)$ into the form

$$\bigvee\Big(\bigwedge_{k\in K_i} \sum_{j\in J_k} \mathsf{in}_j(\varphi_{jk})\Big)$$

We can then perform the following transformations:

$$\begin{array}{lll}
 & \bigvee(\bigwedge_{k\in K_i} \sum_{j\in J_k} \mathsf{in}_j(\varphi_{jk})) & \\
= & \bigvee \sum_{j\in \cap J_k} \mathsf{in}_j(\bigwedge_{k\in K_i} \varphi_{jk}) & (\text{by } \textstyle\sum -\wedge) \\
= & \bigvee \sum_{j\in \cap J_k} \mathsf{in}_j(\bigvee_{l\in L} \psi_{jl}) & (\text{with } \mathsf{Irr}(\psi_{jl}), \text{ by normal form for } \mathcal{L}(\sigma_j)) \\
= & \bigvee \bigvee_{l\in L} \sum_{j\in \cap J_k} \mathsf{in}_j(\psi_{jl}) & (\text{by } \textstyle\sum -\vee)
\end{array}$$

which is of the desired normal form. ■

We can motivate the rule $\sum - \leq$ as follows: an element $\mathsf{in}_j(d)$ satisfies $\sum_{i\in I} \mathsf{in}_i(\varphi_i)$ if d satisfies the property φ_j. So if an element satisfies $\mathsf{in}_1(\varphi_1) + \mathsf{in}_2(\varphi_2)$ then it belongs to either the first or the second summand and *if* it belongs to the first summand then it satisfies φ_1 whereas *if* it belongs to the second summand it will satisfy φ_2. It is thus clear that a sum formula with few summands is a more accurate description of a value than a sum with many summands, since the former restricts the number of possible summands that the value can belong to. This is the explanation behind the entailment rule above.

$$(\mathsf{in}) \quad \frac{\Gamma \vdash e : \varphi}{\Gamma \vdash \mathsf{in}_j(e) : \mathsf{in}_j(\varphi)}$$

$$(\mathsf{case} \ -\mathtt{f}) \quad \frac{\Gamma \vdash e : \mathtt{f}}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{in_i(\overline{x_i})\ .\ e_i\} : \mathtt{f}}$$

$$(\mathsf{case} \ -\vee) \quad \frac{\Gamma \vdash e : \bigvee_{k\in K} \sum_{i\in J_k} \mathsf{in}_i(\varphi_{ik}) \quad \Gamma[\overline{x_i} : \varphi_{ik}] \vdash e_i : \psi_{ik} \quad J_k \neq \emptyset}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{in_i(\overline{x_i})\ .\ e_i\} : \bigvee_{k\in K} \bigvee_{i\in J_k} \psi_{ik}}$$

Figure 6.2: Rules for in and case.

6.3.1 Operations on sums

Data objects of sum type $\sum_{i\in I} \sigma_i$ are created using the constructors in_i where $i \in I$ and taken apart using the case construct. We shall here only give a restricted version of the case expression where we assume that the σ_i are of product type. The symbol $\overline{x}$ then stands for a vector of variables, one for each factor in the product. These x's are used to access the components of the product. Γ is a type environment that gives a type to each free variable in the terms. The typing rules for in_j and case are

$$\frac{\Gamma \vdash e : \sigma_i}{\Gamma \vdash \mathsf{in}_i(e) : \sum_{i\in I} \sigma_i}$$

$$\frac{\Gamma \vdash e : \sum_{i\in I} \sigma_i \quad \Gamma[\overline{x_i} : \sigma_i] \vdash e_i : \rho}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{in_i(\overline{x_i})\ .\ e_i\} : \rho}$$

When there are only two alternatives in a case expression we shall also write it

$$\mathsf{case}\ e\ \mathsf{of}\ in_1(\overline{x})\ .\ e_1\ [\!]\ in_2(\overline{y})\ .\ e_2$$

The rules for reasoning about sum types follow the typing rules. We let Γ denote an environment of formulae that associates a formula to each free variable. Recall that we only consider the case where each of the summands is a product type. We saw in Section 4.2 that irreducible formulae of product type are tuples $\varphi_1 \times \ldots \times \varphi_n$ of irreducible formulae so for $\overline{x}$ a vector of variables and φ such a tuple of formulae we write $\overline{x} : \varphi$ for the environment obtained by matching each variable in $\overline{x}$ with the corresponding irreducible φ_i.

In the rule for case the formula $\sum_{i\in J} \mathsf{in}_i(\varphi_i)$ tells us that one of the branches with index $i \in J$ will be chosen. So we analyse each branch and take the disjunction of

the results obtained. We note that as long as we can only build elements of sum type by using the in_j operation all formulae of sum type will be of form $\mathsf{in}_j(\varphi)$ and disjunctions thereof. The more general form of sum formulae is needed later to model the unfold operation adequately.

The case construct is a generalisation of the if construct that we considered in previous chapters. It is well known that the if construct can be expressed by means of case as follows

- The type Bool is defined to be $\mathbf{1} + \mathbf{1}$.
- Given fresh variables z_1, z_2 not occurring in e_1, e_2 we define

$$\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \equiv \mathsf{case}\ b\ \mathsf{of}\ \mathsf{in}_1(z_1)\ .\ e_1\ []\ \mathsf{in}_2(z_2)\ .\ e_2.$$

With this encoding we can *derive* the rules for if in the disjunctive logic in Section 4.4 from the rule for case. For example the first rule for if deals with the case where the b expression is undefined, *i.e.*, where we have been able to prove $\Gamma \vdash b : \mathtt{f}$. It follows from the rule case $-\,\mathtt{f}$:

$$\frac{\Gamma \vdash b : \mathtt{f}}{\Gamma \vdash \mathsf{case}\ b\ \mathsf{of}\ \mathsf{in}_1(z_1)\ .\ e_1\ []\ \mathsf{in}_2(z_2)\ .\ e_2 : \mathtt{f}}.$$

6.4 Recursive types

A uniform property was characterised as a property that only depends on the content, and not on the structure, of the data structure. That means that we can consider a recursive structure as a multi-set of elements and reason about it as we would reason about sets. We can use the type structure to refine this description so that a structure is viewed as a collection of sets, one for each type the elements can have. The elements in the type lists of integers given by $\mu\alpha.\mathtt{Int} \times \alpha + \mathbf{1}$ will thus be described by two sets: the set of integers in the list and the set of values terminating the list. The latter set will either be the one-point set, in case the list is finite, or the empty set, in case the list is partial or infinite.

We can use the formal system of uniform properties as a basis for a logic of recursive data structures but we have to deal with the empty set explicitly since this is not handled by the uniform properties. With the logic for sum we can express that there is definitely no value in a summand, so by organising our collection of sets as a sum we can express that the set of elements of a given type is empty by stating that there is no set for this type in the sum.

This is the intuition behind the following definition of formulae of recursive type. Define the set $L(\mu\alpha.\sum_i D_i{\times}\alpha^{n_i})$ of formulae denoting properties of recursive type as follows:

Definition 6.4.1 (Formulae for recursive types)

$$L(\mu\alpha.\sum_i \sigma_i{\times}\alpha^{n_i}) = L(\sum_i \mathcal{P}(\sigma_i)).$$

Let us look at some examples from the list of integers type $\mu\alpha.\mathtt{Int}\times\alpha{+}\mathbf{1}$. Assuming that we have two properties for integer values, *viz.* $\mathtt{f}_{\mathtt{Int}}$ meaning that the value is undefined and $\mathtt{t}_{\mathtt{Int}}$ meaning that the value might be defined, we can form properties like the following for lists of integers.

- $\mathtt{f}_{\mathcal{P}(\mathtt{Int})+\mathcal{P}(\mathbf{1})}$ This property states that there is no content from either summand. It is only satisfied by the undefined list $\bot_{\mu\alpha.\mathtt{Int}\times\alpha+\mathbf{1}}$.
- $\mathsf{in}_1(\forall\mathtt{t}_{\mathtt{Int}})$ This describes all the lists whose ending is undefined, *i.e.*, partial or infinite lists.
- $\mathsf{in}_1(\exists\mathtt{f}_{\mathtt{Int}})$ This describes all the lists that contain an undefined element *and* whose ending is undefined.
- $\mathsf{in}_1(\forall\mathtt{f}_{\mathtt{Int}}) + \mathsf{in}_2(\forall\mathtt{t}_{\mathbf{1}})$ This property is satisfied by all finite, infinite or partial lists, all the elements of which are undefined.
- $\mathsf{in}_1(\forall\mathtt{t}_{\mathtt{Int}}) \vee (\mathsf{in}_1(\exists\mathtt{f}_{\mathtt{Int}}) + \mathsf{in}_2(\forall\mathtt{t}_{\mathbf{1}}))$ Either the end of the list is undefined or the list contains an undefined element. This is Wadler's $\bot \in$ element.
- $\mathsf{in}_1(\forall\mathtt{t}_{\mathtt{Int}}) + \mathsf{in}_2(\forall\mathtt{t}_{\mathbf{1}})$ The weakest property that holds for all lists. Corresponds to the top element in the 4-point domain.

6.4.1 The rules for fold and unfold

The following rules define how the operations fold and unfold interact with the properties for recursive structures. The axiomatisation follows that of other functions in that we only specify fold and unfold's behaviour on irreducible formulae and let the generic axioms for functions define the extensions to the general case of disjunctions of irreducibles.

The irreducible formulae in $\mathcal{L}(\sum_{i\in I}\mathcal{P}(\sigma_i))$ are of the form

$$\sum_{j\in J}\mathsf{in}_j(\forall \bigvee_{i\in I_j}\varphi_{ij} \wedge \bigwedge_{i\in I_j}\exists\varphi_{ij}) \qquad \varphi_{ij}\in \mathsf{Irr}(L(\sigma_k))$$

Assume $i \notin J$.

$$(\mathsf{fold}-1) \quad \mathsf{fold} \ : \mathsf{in}_i(\varphi \times \sum_{j\in J} \mathsf{in}_j(\chi_h)) \to \mathsf{in}_i(\forall\varphi \wedge \exists\varphi) + \sum_{j\in J} \mathsf{in}_j(\chi_h)$$

$$\begin{aligned}(\mathsf{fold}-2) \quad \mathsf{fold} \ : \ & \mathsf{in}_i(\varphi \times \mathsf{in}_i(\forall \bigvee_{k\in K_i} \varphi_{ik} \wedge \bigwedge_{k\in K_i} \exists\varphi_{ik}) + \sum_{j\in J} \mathsf{in}_j(\chi_h)) \\ & \to \quad \mathsf{in}_i(\forall(\varphi \vee \bigvee_{k\in K_i} \varphi_{ik}) \wedge \exists\varphi \wedge \bigwedge_{k\in K_i} \exists\varphi_{ik}) + \sum_{j\in J} \mathsf{in}_j(\chi_h)\end{aligned}$$

$$(\mathsf{fold}-\textstyle\sum) \quad \frac{\mathsf{fold} \ : \psi_i \to \sum_{k\in K_i} \chi_k \qquad (i \in I)}{\mathsf{fold} \ : \sum_{i\in I} \psi_i \to \bigvee_{i\in I} \sum_{k\in K_i} \chi_k}$$

Figure 6.3: Rules for fold

The argument to fold is described by the even more complex formulae in $\mathcal{L}(\sum_{i\in I} \sigma_i \times \sum_{i\in I} \mathcal{P}(\sigma_i))$ consisting of a property of the element to be "folded" into the structure, in addition to a property of the structure itself. When adding an element from D_j satisfying property φ to a data structure we have to modify that part of the formula that describes the set of elements from summand D_j. Assume this set satisfies the formula

$$\forall \bigvee_{i\in I} \varphi_i \wedge \bigwedge_{i\in I} \exists\varphi_i.$$

Inserting an element satisfying φ creates a set that satisfies

$$\forall(\varphi \vee \bigvee_{i\in I} \varphi_i) \wedge \exists\varphi \wedge \bigwedge_{i\in I} \exists\varphi_i.$$

This is expressed in the rule for fold shown in Figure 6.3. The fold-operation can only change the set of D_i-elements, hence the formulae describing the other summands are left unchanged. In the rules we let variables χ_h range over those irreducible formulae of $\mathcal{L}(\mathcal{P}(\sigma_k))$ that do not change.

The property describing the argument to unfold will be a disjunction of irreducible properties of the form

$$\sum_{j\in J} \mathsf{in}_j(\forall(\bigvee_{i\in I_j} \varphi_{ij}) \wedge \bigwedge_{i\in I_j} \exists\varphi_{ij}).$$

Again we only axiomatise unfold on these simpler formulae and let the general framework define the extension to disjunctions of these. Unfolding such a structure gives a tagged pair $\mathsf{in}_j(d, e')$ where d belongs to the set of elements described by the j'th summand in the formula above. We want to determine the properties that d and the set of elements in e' satisfy. Since we have no information about the order in

$$\vdash \mathsf{unfold} : \sum_{j\in\emptyset} \mathsf{in}_j(\chi_j) \to \mathtt{f} = \mathtt{f} \to \mathtt{f}$$

$$\vdash \mathsf{unfold} : \sum_{j\in J} \mathsf{in}_j(\forall \bigvee_{i\in I_j} \varphi_{ij} \wedge \bigwedge_{i\in I_j} \exists\varphi_{ij}) \quad \to \qquad (j \neq \emptyset)$$
$$\bigvee_{j\in J} \bigvee_{\emptyset\neq I\subseteq I_j} \mathsf{in}_j(\bigwedge_{i\in I} \varphi_{ij} \times \mathsf{in}_j(\forall \bigvee_{i\in I_j} \varphi_{ij} \wedge \bigwedge_{i\in I_j\setminus I} \exists\varphi_{ij}) + \sum_{J\setminus\{j\}} \mathsf{in}_j(\forall \bigvee_{i\in I_j} \varphi_{ij} \wedge \bigwedge_{i\in I_j} \exists\varphi_{ij}))$$

Figure 6.4: Rules for unfold.

which the elements appear in the data structure this corresponds directly to choosing randomly an element from a set and we can use the same logic for unfold as we used for the *extract* operation on sets in Section 5.5. That is, d belongs to one of the summands indexed by J, k say, and satisfies a certain subset of the properties φ_{ik}. The rest of the structure satisfies the same $\forall$-properties, but we can no longer guarantee that there exist elements satisfying the properties that d satisfies, since d could have been the only element in the list satisfying these properties. Finally, if the argument to unfold is undefined then so is the result hence unfold satisfies $\mathtt{f} \to \mathtt{f}$. This gives the unfold-rules shown in Figure 6.4.

Example. The property $\mathsf{in}_1(\forall \mathtt{t}_{\mathtt{Int}})$ is satisfied by all partial or infinite integer lists where the elements could be defined. Assume we have that $\Gamma \vdash e : \mathsf{in}_1(\forall \mathtt{t}_{\mathtt{Int}} \wedge \exists \mathtt{t}_{\mathtt{Int}})$ for some expression e of list type. Sticking the value 1 in front of the list obtained by evaluating e is done with the expression $\mathsf{fold}\ \mathsf{in}_1((1, e))$ and we would expect to be able to prove the same property for this expression.

First we get the following property in normal form of the expression $\mathsf{in}_1((1, e))$

$$\Gamma \vdash \mathsf{in}_1((1, e)) : \mathsf{in}_1(\mathtt{t}_{\mathtt{Int}} \times \mathsf{in}_1(\forall \mathtt{t}_{\mathtt{Int}} \wedge \exists \mathtt{t}_{\mathtt{Int}}))$$

Applying the fold-rule and omitting type subscripts we obtain

$$\Gamma \vdash \mathsf{fold}\ \mathsf{in}_1((1, e)) : \mathsf{in}_1(\forall(\mathtt{t} \vee \mathtt{t}) \wedge \exists \mathtt{t} \wedge \exists \mathtt{t}) = \mathsf{in}_1(\forall \mathtt{t} \wedge \exists \mathtt{t})$$

Example. As an example of the use of the unfold rule we consider unfolding a partial or infinite list which contains at least one undefined element. This property of a list is described by the formula

$$\mathsf{in}_1(\exists \mathtt{f}) \equiv \mathsf{in}_1(\forall(\mathtt{t} \vee \mathtt{f}) \wedge \exists \mathtt{t} \wedge \exists \mathtt{f}).$$

The rule for unfold applied to the formula in normal form gives that the result of

the unfold operation satisfies:

$$\mathsf{in}_1(\mathtt{t} \times \mathsf{in}_1(\forall(\mathtt{t} \vee \mathtt{f}) \wedge \exists\mathtt{f}) \vee \mathsf{in}_1(\mathtt{f} \times \mathsf{in}_1(\forall(\mathtt{t} \vee \mathtt{f}) \wedge \exists\mathtt{t}) \vee \mathsf{in}_1((\mathtt{t} \wedge \mathtt{f}) \times \mathsf{in}_1(\forall(\mathtt{t} \vee \mathtt{f})))$$

which is equivalent to the formula

$$\mathsf{in}_1(\mathtt{t} \times \mathsf{in}_1(\exists\mathtt{f})) \vee \mathsf{in}_1(\mathtt{f} \times \mathsf{in}_1(\forall\mathtt{t})).$$

This formula asserts that the unfold operation uncovers either an element which might be defined, in which case the remainder of the lists still contains an undefined element, or an element which is undefined in which case we cannot say anything about the elements in the rest of the list since this could have been the only undefined element in the whole list.

6.5 Analysing length and map

As an example of the use of the logic we prove some properties of the functions length and map.

6.5.1 Computing the length of an infinite list

We shall prove that taking the length of an infinite list yields an undefined result. The function length is defined by

$$\mathsf{length} = \mathsf{fix}\ \lambda L.\lambda l.\mathsf{case}\ \mathsf{unfold}\ l\ \mathsf{of}\ \mathsf{in}_1(x, xs)\ .\ 1 + L\ xs\ []\ \mathsf{in}_2(y)\ .\ 0$$

The infinite list where nothing is known about the elements is characterised by the formula $\mathsf{in}_1(\forall\mathtt{t})$. We want to prove that

$$\mathsf{length} : \mathsf{in}_1(\forall\mathtt{t}) \to \mathtt{f}.$$

The fix-rule requires us to prove that under the hypothesis Γ:

$$\Gamma \equiv [L : \mathsf{in}_1(\forall\mathtt{t}) \to \mathtt{f}, l : \mathsf{in}_1(\forall\mathtt{t})]$$

the result of evaluating the case-expression is undefined (satisfies the property denoted by $\mathtt{f}$). The description of the unfolding of the list l is simplified by the fact that there are only properties for one of the summands from the list type definition.

Using the rules for unfold we get

$$\Gamma \vdash \mathsf{unfold}\ l : \mathsf{in}_1(\mathtt{t} \times \mathsf{in}_1(\forall \mathtt{t}))$$

Proceeding with the rule for analysing case-expression we have to analyse the body of the expression. Since there is no property for the second clause, all we have to prove is

$$\Gamma[x : \mathtt{t}, xs : \mathsf{in}_1(\forall \mathtt{t})] \vdash 1 + L\ xs : \mathtt{f}.$$

This fact is easily verified using the rules for addition and function application and it allows us to conclude that

$$\Gamma \vdash \mathsf{case}\ \mathsf{unfold}\ l\ \mathsf{of}\ \mathsf{in}_1(x, xs)\ .\ 1 + L\ xs\ []\ \mathsf{in}_2(y)\ .\ 0 : \mathtt{f}$$

which was the goal.

6.5.2 Mapping a strict function over a list containing an undefined element

The map-function is defined as follows

$$\mathtt{map} = \mathsf{fix}\ \lambda M.\lambda f.\lambda l.\mathsf{case}\ \mathsf{unfold}\ l\ \mathsf{of}\ \ \mathsf{in}_1(x, xs)\ .\ \mathsf{fold}\ \mathsf{in}_1(fx, M\ f\ xs)\ []\ \mathsf{in}_2(y)\ .\ \mathsf{fold}\ \mathsf{in}_2(\bullet)$$

We want to verify that mapping a strict function over a list containing an undefined element gives a list that also contains an undefined element. When lists are defined by $\mu\alpha.B \times \alpha + \mathbf{1}$, the set of lists with an undefined element is described by the formula $\mathsf{in}_1(\exists \mathtt{f}) + \mathsf{in}_2(\forall \mathtt{t})$ and a strict function on the integers by $\mathtt{f} \to \mathtt{f}$. We define the environment Γ by

$$[M : (\mathtt{f} \to \mathtt{f}) \to \mathsf{in}_1(\exists \mathtt{f}) + \mathsf{in}_2(\forall \mathtt{t}) \to \mathsf{in}_1(\exists \mathtt{f}) + \mathsf{in}_2(\forall \mathtt{t}), f : \mathtt{f} \to \mathtt{f}, l : \mathsf{in}_1(\exists \mathtt{f}) + \mathsf{in}_2(\forall \mathtt{t})]$$

and must show that

$$\Gamma \vdash \mathsf{case}\ \mathsf{unfold}\ l\ \mathsf{of}\ \ \mathsf{in}_1(x, xs)\ .\ \mathsf{fold}\ \mathsf{in}_1(fx, M\ f\ xs)\ []\ \mathsf{in}_2(y)\ .\ \mathsf{fold}\ \mathsf{in}_2(\bullet)$$

satisfies $\mathsf{in}_1(\exists \mathtt{f}) + \mathsf{in}_2(\forall \mathtt{t})$. We first present $l : \mathsf{in}_1(\exists \mathtt{f}) + \mathsf{in}_2(\forall \mathtt{t})$ in its normal form $\mathsf{in}_1(\forall(\mathtt{t} \vee \mathtt{f}) \wedge \exists \mathtt{t} \wedge \exists \mathtt{f}) + \mathsf{in}_2(\forall \mathtt{t} \wedge \exists \mathtt{t})$ so we can submit it to the unfold-rule. According

to this rule $\mathsf{unfold}\ l$ satisfies the disjunction

$$\mathsf{in}_1(\mathtt{f} \times \mathsf{in}_1(\forall\mathtt{t}) + \mathsf{in}_2(\forall\mathtt{t})) \vee \mathsf{in}_1(\mathtt{t} \times \mathsf{in}_1(\exists\mathtt{f}) + \mathsf{in}_2(\forall\mathtt{t})) \vee \mathsf{in}_2(\mathtt{t} \times \mathsf{in}_1(\exists\mathtt{f}) + \mathsf{in}_2(\forall\mathtt{t}))$$

This gives three alternative properties for the argument to case: the two first match the first branch, the last one matches the second branch. These matchings result in three different extensions to the environment Γ which we shall denote by Γ_1, Γ_2 and Γ_3.

$$\Gamma_1 = \Gamma[x : \mathtt{f}, xs : \mathsf{in}_1(\exists\mathtt{t})] \qquad \Gamma_2 = \Gamma[x : \mathtt{t}, xs : \mathsf{in}_1(\exists\mathtt{f})] \qquad \Gamma_3 = \Gamma[y : \mathtt{t}]$$

It can be shown using rules from Figure 4.4 that

$$\Gamma_1 \vdash \mathsf{in}_1(fx, M\ f\ xs) : \mathsf{in}_1(\mathtt{f} \times \mathsf{in}_1(\forall\mathtt{t}) + \mathsf{in}_2(\forall\mathtt{t})$$
$$\Gamma_2 \vdash \mathsf{in}_1(fx, M\ f\ xs) : \mathsf{in}_1(\mathtt{t} \times \mathsf{in}_1(\exists\mathtt{f}) + \mathsf{in}_2(\forall\mathtt{t})$$
$$\Gamma_3 \vdash \mathsf{in}_2(\bullet) : \mathsf{in}_2(\mathtt{t})$$

and hence, using the rule for fold,

$$\Gamma_1 \vdash \mathsf{fold}\ \mathsf{in}_1(fx, M\ f\ xs) : \mathsf{in}_1(\exists\mathtt{f}) + \mathsf{in}_2(\forall\mathtt{t})$$
$$\Gamma_2 \vdash \mathsf{fold}\ \mathsf{in}_1(fx, M\ f\ xs) : \mathsf{in}_1(\exists\mathtt{f}) + \mathsf{in}_2(\forall\mathtt{t})$$
$$\Gamma_3 \vdash \mathsf{fold}\ \mathsf{in}_2(\bullet) : \mathsf{in}_2(\forall\mathtt{t}) \leq \mathsf{in}_1(\exists\mathtt{f}) + \mathsf{in}_2(\forall\mathtt{t})$$

so we can conclude that the whole case-expression satisfies $\mathsf{in}_1(\exists\mathtt{f}) + \mathsf{in}_2(\forall\mathtt{t})$.

6.6 Abstract interpretation of sums

In this section we calculate the *Lindenbaum* algebra of the logical system $\mathcal{L}(\sum_{i\in I}\sigma_i)$. This algebra forms a distributive lattice over which the abstract interpretation of the two operations associated with the sum type is defined.

6.6.1 Lattices for sum types

A property describing an element of a sum type $\sum_{i\in I}\sigma_i$ is a sum of properties $\sum_{j\in J}\mathsf{in}_j(\varphi_j)$ where J is the (index) set of summands that the element can belong to and φ_j the property satisfied by the element in case it belongs to σ_j. We can write such a sum of properties as an I–tuple of properties if we add an extra symbol

Assume $J \subseteq \{1, \ldots, n\}$.

$$
\begin{array}{lcl}
[\![\ ]\!] : \mathcal{L}(\sum_{i=1}^{n} \sigma_i) \to \mathsf{D}(\prod_{i=1}^{n} (\mathcal{J}(\sigma_i))_\perp) & & \\
[\![\sum_J \mathsf{in}_j(\varphi_j)]\!]_{\sum_i \sigma_i} & = & \{(e_1, \ldots, e_n) \mid e_i = \perp \text{ or } e_i \in [\![\varphi_i]\!]\} \\
[\![\mathtt{t}]\!]_{\sum \sigma_i} & = & \mathcal{J}(\sigma_1)_\perp \times \ldots \times \mathcal{J}(\sigma_n)_\perp \\
[\![\mathtt{f}]\!]_{\sum \sigma_i} & = & \downarrow\{(\perp, \ldots, \perp)\}
\end{array}
$$

Figure 6.5: Semantics of sum formulae.

that expresses *"no property of this summand in the sum"*. As already argued in Section 6.3, a formula stating that the value is definitely not in summand j, say, logically implies a formula saying that *if* the value is from summand j then it satisfies some property φ. This means that the new symbol is least in the implication ordering, so we can add this extra symbol by lifting the lattice $\mathcal{LA}(\sigma_i)$, adding a new bottom element $\perp$. The interpretation of a sum type, which is the Lindenbaum algebra of the formal system for that sum type, then becomes:

$$[\![\sum_{i \in I} \sigma_i]\!] = \mathcal{LA}(\sum_{i \in I} \sigma_i) = \mathsf{D}(\prod_{i \in I} (\mathcal{J}(\sigma_i))_\perp)$$

where $\prod$ is the cartesian product of lattices.

The semantics of a formula $\sum_J \mathsf{in}_j(\varphi_j) \in L(\sum_I \sigma_i)$ with φ_j irreducible is the downwards closure of the tuple where all components with index in $I \setminus J$ are equal to $\perp$ and the j'th component for $j \in J$ is d_j where $\downarrow\{d_j\} = [\![\varphi_j]\!]$. The least upper bound is union of lower sets of tuples, which is calculated by applying union componentwise.

Notation: When dealing with these sum domains we shall frequently encounter the situation where some components of a tuple are described explicitly and the rest are defined to be $\perp$, so we introduce a way of writing such a tuple. For a cartesian product of lattices $\prod_{i=1,\ldots,n}(D_i)_\perp$ and elements $\{d_j\}_{j \in J}$ with $J \subseteq \{1, \ldots, n\}$ we define

$$\prod_{j \in J} d_j \equiv (e_1, \ldots, e_n) \quad \text{where } d_j = e_j, \forall j \in J, \text{ and } e_i = \perp \text{ otherwise.}$$

In particular, $\prod_{j \in \emptyset} d_j$ is the tuple $(\perp, \ldots, \perp)$.

The semantics of the formulae in $L(\sum_I \sigma_i)$ is given in Figure 6.5. Con– and disjunction are still interpreted as union and intersection of lower sets. With this semantics we can give a precise description of the relationship between the formal system of sum properties and the partial orders just defined by showing that the semantics

$[\![\ ]\!]$ defines an order-isomorphism between the Lindenbaum algebra $\mathcal{LA}(\sum_{i=1}^{n}\sigma_i)$ and the model $\mathsf{D}(\prod_{i=1}^{n}(\mathcal{J}(\sigma_i))_\perp)$.

Lemma 6.6.1 *The function* $[\![\ ]\!]$ *is surjective on* $\mathsf{Idl}(\prod_{i=1}^{n}(\mathcal{J}(\sigma_i))_\perp)$.

Proof. Let $\downarrow\{(e_1,\ldots,e_n)\}$ be an ideal in $\prod_{i\in I}(\mathcal{J}(\sigma_i))_\perp$ and let J denote the set $\{j\in I \mid e_j\neq\perp\}$. By hypothesis there exists $\varphi_j\in\mathsf{Irr}(\mathcal{L}(\sigma_j))$ such that $[\![\varphi_j]\!]=\downarrow\{e_j\}$. Then

$$[\![\sum_{j\in J}\mathsf{in}_j(\varphi_j)]\!]=\{(d_1,\ldots,d_n)\mid d_j\in\downarrow\{e_j\}\text{ or }d_j=\perp\}=\downarrow\{(e_1,\ldots,e_n)\}.$$

■

Since disjunction is interpreted as union every lower set, being a union of ideals, in $\prod_{i=1}^{n}(\mathcal{J}(\sigma_i))_\perp$ is in the image of $[\![\ ]\!]$.

Lemma 6.6.2 *Let* $\varphi,\psi\in L(\sum_{i=1}^{n}\sigma_i)$. *Then*

$$\varphi\leq\psi\Rightarrow[\![\varphi]\!]\subseteq[\![\psi]\!].$$

Proof. As in the proof of Theorem 4.6.1 this lemma is proved by checking that each rule defining $\leq$ is sound. The rest is then an induction on the length of the proof of $\varphi\leq\psi$.

- $\sum-\wedge$

$$\begin{aligned}
&[\![\sum_{j\in J_1}\mathsf{in}_j(\varphi_j)]\!]\cap[\![\sum_{j\in J_2}\mathsf{in}_j(\psi_j)]\!]\\
=\ &\{(e_1,\ldots,e_n)\mid e_j=\perp\text{ or }(e_j\in[\![\varphi_j]\!]\text{ and }e_j\in[\![\psi_j]\!])\}\\
=\ &\{(e_1,\ldots,e_n)\mid e_j=\perp\text{ or }\exists j\in J_1\cap J_2.(e_j\in[\![\varphi_j]\!]\cap[\![\psi_j]\!])\}\\
=\ &[\![\textstyle\sum_{j\in J_1\cap J_2}\mathsf{in}_j(\varphi_j\wedge\psi_j)]\!]
\end{aligned}$$

- $\sum-\mathtt{f}$.

$$[\![\sum_{j\in\emptyset}\mathsf{in}_j(\varphi_j)]\!]=\{(\perp,\ldots,\perp)\}=[\![\mathtt{f}_{\sum_I\sigma_i}]\!]$$

- $\sum-\mathtt{t}$.

$$[\![\sum_{j=1}^{n}\mathsf{in}_j(\mathtt{t})]\!]=\{(e_1,\ldots,e_n)\mid e_i=\perp\text{ or }e_i\in[\![\mathtt{t}]\!]\}=\mathcal{J}(\sigma_1)_\perp\times\ldots\times\mathcal{J}(\sigma_n)_\perp=[\![\mathtt{t}_{\sum_{i=1}^{n}\sigma_i}]\!]$$

- $\sum -\vee$.

$$
\begin{aligned}
&[\![\sum_{j\in J} \mathsf{in}_j(\varphi_j \vee \psi_j)]\!] \\
&= \{(e_1,\ldots,e_n) \mid e_j \in [\![\varphi_j]\!] \cup [\![\psi_j]\!] \text{ or } e_j = \bot\} \\
&= \{(e_1,\ldots,e_n) \mid e_j \in [\![\varphi_j]\!] \text{ or } e_j \in [\![\psi_j]\!] \text{ or } e_j = \bot\} \\
&= \{(e_1,\ldots,e_n) \mid e_j \in [\![\varphi_j]\!] \text{ or } e_j = \bot\} \cup \{(e_1,\ldots,e_n) \mid e_j \in [\![\psi_j]\!] \text{ or } e_j = \bot\} \\
&= [\![\textstyle\sum_{j\in J} \mathsf{in}_j(\varphi_j)]\!] \vee [\![\textstyle\sum_{j\in J} \mathsf{in}_j(\psi_j)]\!]
\end{aligned}
$$

- $\sum - \leq$. Follows from the fact that if $[\![\varphi_j]\!] \subseteq [\![\psi_j]\!]$ for all $j \in J_1$ then

$$\{(e_1,\ldots,e_n) \mid e_j \in [\![\varphi_j]\!] \text{ or } e_j = \bot\} \subseteq \{(e_1,\ldots,e_n) \mid e_j \in [\![\psi_j]\!] \text{ or } e_j = \bot\}.$$

■

Lemma 6.6.3 *Let $\varphi, \psi \in L(\sum_I \sigma_i)$. Then*

$$[\![\varphi]\!] \subseteq [\![\psi]\!] \Rightarrow \varphi \leq \psi.$$

Proof. We first prove this for irreducible φ, ψ. Assume that

$$[\![\sum_{j\in J_1} \mathsf{in}_j(\varphi_j)]\!] \subseteq [\![\sum_{j\in J_2} \mathsf{in}_j(\psi_j)]\!].$$

Applying the rules for $[\![\ ]\!]$ we get from this that

$$\{(e_1,\ldots,e_n) \mid e_j \in [\![\varphi_j]\!] \text{ or } e_j = \bot\} \subseteq \{(e_1,\ldots,e_n) \mid e_j \in [\![\psi_j]\!] \text{ or } e_j = \bot\}$$

which implies that

$$\forall j \in J_1 \,.\, [\![\varphi_j]\!] \subseteq [\![\psi_j]\!]$$

We then invoke the induction hypothesis to get

$$\forall j \in J_1 \,.\, \varphi_j \leq \psi_j$$

and by an application of the rule $(\sum - \leq)$ we obtain

$$\sum_{j\in J_1} \mathsf{in}_j(\varphi_j) \leq \sum_{j\in J_2} \mathsf{in}_j(\psi_j).$$

For the general case we argue as in Theorem 4.6.1: If

$$\bigcup_{k\in K} [\![\sum_{j\in J_k} \mathsf{in}_j(\varphi_{jk})]\!] \subseteq \bigcup_{l\in L} [\![\sum_{j\in J_l} \mathsf{in}_j(\psi_{jl})]\!]$$

then for all $k \in K$ there is some $l \in L$ such that

$$[\![\sum_{j\in J_k} \mathsf{in}_j(\varphi_{jk})]\!] \subseteq [\![\sum_{j\in J_l} \mathsf{in}_j(\psi_{jl})]\!]$$

so, by the result for irreducibles above we get

$$\forall k \in K\ \exists l \in L\ .\ \sum_{j\in J_k} \mathsf{in}_j(\varphi_{jk}) \leq \sum_{j\in J_l} \mathsf{in}_j(\psi_{jl}).$$

From this it follows by the rules for disjunction that

$$\bigvee_{k\in K} \sum_{j\in J_k} \mathsf{in}_j(\varphi_{jk}) \leq \bigvee_{l\in L} \sum_{j\in J_l} \mathsf{in}_j(\psi_{jl}).$$

which proves completeness of the formal system in the general case. ■

We sum this up in the following Theorem

Theorem 6.6.4 *The function*

$$[\![\ ]\!] /{=} :\ \mathcal{LA}(\sum_{i\in I} \sigma_i) \to \mathsf{D}(\prod_{i\in I} (\mathcal{J}(\sigma_i))_\perp)$$

is an order isomorphism.

6.6.2 Abstract interpretation of in and case

We now define the abstract interpretation of in and case and show that this interpretation is sound and complete with respect to the logic for reasoning about sums defined in Section 6.3.

The abstract function $[\![\mathsf{in}_j]\!]$ maps an element from $[\![\sigma_j]\!]$ to a tuple where all components, except the j'th, are equal to the bottom element:

$$\begin{aligned} [\![\mathsf{in}_j(e)]\!]\rho &= \downarrow\{\textstyle\prod_{\{j\}} d_k \mid [\![e]\!]\rho = \downarrow\{d_1, \ldots, d_n\} \text{ and } k = 1, \ldots, n\} \\ &= \downarrow\{\textstyle\prod_{\{j\}} d \mid d \in [\![e]\!]\rho\} \end{aligned}$$

With the case expression we first analyse the expression of sum type. The outcome of this analysis is either that the value of e is undefined, in which case the result of the whole case expression is undefined, or a set of alternatives of which we analyse each in turn and form the union of the results so obtained.

$$[\![\mathsf{case}\ e\ \mathsf{of}\ \{\overline{x_i}\ .\ e_i\}]\!]\rho = \begin{cases} \{0\} & \text{if } [\![e]\!]\rho = (\bot,\ldots,\bot) \\ \bigcup\{[\![e_i]\!]\rho[\overline{x_i} : \downarrow\{d_{ij}\}] & \text{if } [\![e]\!]\rho = \downarrow\{\prod_{I_j} d_{ij} \mid j \in J\}\} \end{cases}$$

The following theorems prove soundness and completeness of the logic for in and case with repect to the abstract interpretation. The proofs should be viewed as extensions of the inductive proofs of the Soundness and Completeness Theorems for the disjunctive logic (Theorems 4.6.1 and 4.6.3).

Theorem 6.6.5 *The rules for* in *and* case *are sound with respect to the abstract interpretation* $[\![\ ]\!]$.

Proof. Throughout the proof we assume $\rho_\Gamma(x) = [\![\Gamma(x)]\!]$. The rule for in is

$$\frac{\Gamma \vdash e : \sigma_i}{\Gamma \vdash \mathsf{in}_i(e) : \sum_{i\in I} \sigma_i}$$

So we assume that $[\![e]\!]\rho_\Gamma \subseteq [\![\varphi]\!]$. Then

$$\begin{aligned} [\![\mathsf{in}_j(e)]\!]\rho_\Gamma &= \downarrow\{\textstyle\prod_j d \mid d \in [\![e]\!]\rho_\Gamma\} \\ &\subseteq \downarrow\{\textstyle\prod_j d \mid d \in [\![\varphi]\!]\} \\ &= [\![\mathsf{in}_j(\varphi)]\!] \end{aligned}$$

which proves the soundness of the in rule.

The soundness of the rule case $-\mathtt{f}$ is immediate since if $[\![e]\!]\rho_\Gamma \subseteq [\![\mathtt{f}]\!] = \{(\bot,\ldots,\bot)\}$ we get that

$$[\![\mathsf{case}\ e\ \mathsf{of}\ \{\overline{x_i}\ .\ e_i\}]\!]\rho_\Gamma = \{0\} \subseteq [\![\mathtt{f}]\!].$$

The other rule for case is

$$\frac{\Gamma \vdash e : \bigvee_{k\in K} \sum_{j\in J_k} \mathsf{in}_j(\varphi_{jk}) \quad \Gamma[\overline{x_j} : \varphi_{jk}] \vdash e_j : \psi_{jk}}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{in_j(\overline{x_j})\ .\ e_j\} : \bigvee_{k\in K} \bigvee_{j\in J_k} \psi_{jk}}$$

so the assumptions here are

$$[\![e]\!]\rho_\Gamma \subseteq [\![\bigvee_{k\in K} \sum_{j\in J_k} \mathsf{in}_j(\varphi_{jk})]\!] \qquad \text{and} \qquad \forall k \in K\ \forall j \in J_k\ .\ [\![e_j]\!]\rho_{\Gamma[\overline{x_j}:\varphi_{jk}]} \subseteq [\![\psi_{jk}]\!].$$

These assumptions can be rewritten using the definition of $[\![\ ]\!]$. The φ_{jk} are irreducible so to each φ_{jk} there exists a c_{jk} such that $[\![\varphi_{jk}]\!] = \downarrow\{c_{jk}\}$. Let $d_{il} \in \mathcal{J}(\sigma_i)$ be defined by $[\![e]\!]\rho_\Gamma = \bigcup_{l\in L} \downarrow\{\prod_{i\in I_l} d_{il}\}$. The first assumption then implies

$$\begin{aligned}
[\![e]\!]\rho_\Gamma &= \bigcup_{l\in L} \downarrow\{\prod_{i\in I_l} d_{il}\} \\
&\subseteq [\![\bigvee_{k\in K} \sum_{j\in J_k} \mathsf{in}_j(\varphi_{jk})]\!] \\
&= \bigcup_{k\in K} [\![\sum_{j\in J_k} \mathsf{in}_j(\varphi_{jk})]\!] \\
&= \bigcup_{k\in K} \downarrow\{\prod_{j\in J_k} c_{jk}\}
\end{aligned}$$

For the second assumption we have that $\rho_{\Gamma[\overline{x_j}:\varphi_{jk}]} = \rho_\Gamma[\overline{x_j} \mapsto \downarrow\{c_{jk}\}]$ so it rewrites to

$$\forall k \in K\ \forall j \in J_k\ .\ [\![e_j]\!]\rho_\Gamma[\overline{x_j} \mapsto \downarrow\{c_{jk}\}] \subseteq [\![\psi_{jk}]\!].$$

The first assumption has as implication

$$\forall l \in L\ \exists k \in K\ .\ \prod_{i\in I_l} d_{il} \sqsubseteq \prod_{j\in J_k} c_{jk}$$

i.e.,

$$\forall l \in L\ \exists k \in K\ .\ I_l \subseteq J_k \ \text{ and } \ \forall i \in I_l\ .\ d_{il} \sqsubseteq c_{ik}.$$

Since the semantic function $[\![\ ]\!]$ is monotone in the environment this implies

$$\forall l \in L\ \exists k \in K\ .\ I_l \subseteq J_k \ \text{ and } \ \forall i \in I_l\ .\ [\![e_i]\!]\rho_\Gamma[\overline{x_i} \mapsto \downarrow\{d_{il}\}] \subseteq [\![e_i]\!]\rho_\Gamma[\overline{x_i} \mapsto \downarrow\{c_{ik}\}]$$

from which we deduce

$$\forall l \in L\ \exists k \in K\ .\ \bigcup_{i\in I_l} \{[\![e_i]\!]\rho_\Gamma[\overline{x_i} \mapsto \downarrow\{d_{il}\}]\} \subseteq \bigcup_{j\in J_k} \{[\![e_j]\!]\rho_\Gamma[\overline{x_j} \mapsto \downarrow\{c_{jk}\}]\}.$$

We then use the second hypothesis to get

$$\forall l \in L\ \exists k \in K\ .\ \bigcup_{i\in I_l} \{[\![e_i]\!]\rho_\Gamma[\overline{x_i} \mapsto \downarrow\{d_{il}\}]\} \subseteq \bigcup_{j\in J_k} [\![\psi_{jk}]\!]$$

and we arrive at the conclusion that

$$\bigcup_{l\in L} \bigcup_{i\in I_l} \{[\![e_i]\!]\rho_\Gamma[\overline{x_i} \mapsto \downarrow\{d_{il}\}]\} \subseteq \bigcup_{k\in K} \bigcup_{j\in J_k} [\![\psi_{jk}]\!]$$

which is the semantic equivalent to the conclusion of the rule for case. ■

Theorem 6.6.6 (Completeness for in.) *Let environments ρ_Γ, Γ be defined such that $\rho_\Gamma(x) = [\![\Gamma(x)]\!]$. Then*

$$[\![\mathsf{in}_i(e)]\!]\rho_\Gamma \subseteq [\![\bigvee_{k\in K} \sum_{j\in J_k} \mathsf{in}_j(\varphi_{jk})]\!] \;\Rightarrow\; \Gamma \vdash \mathsf{in}_i(e) : \bigvee_{k\in K} \sum_{j\in J_k} \mathsf{in}_j(\varphi_{jk}).$$

Proof. Assume

$$[\![\mathsf{in}_i(e)]\!]\rho_\Gamma \subseteq [\![\bigvee_{k\in K} \sum_{j\in J_k} \mathsf{in}_i(\varphi_{jk})]\!]$$

and

$$[\![e]\!]\rho_\Gamma = \downarrow\{d_1, \ldots, d_n\}.$$

From this we get

$$\bigcup_{l=1}^{n} \downarrow\{\Pi_{\{i\}}\, d_l\} \subseteq \bigcup_{k\in K} \downarrow\{\Pi_{j\in J_k}\, d \mid d \in [\![\varphi_{jk}]\!]\}$$

which implies that

$$\forall l = 1, \ldots, n\ \exists k \in K\ .\ i \in J_k \ \text{ and } \ d_l \in [\![\varphi_{ik}]\!]$$

This means that there is a subset $K_0 \subseteq K$ such that $i \in J_k$ for all $k \in K_0$ and every d_i satisfy at least one of the $\varphi_{ik}, k \in K_0$. For the abstract interpretation of e this entails

$$[\![e]\!]\rho \subseteq \bigcup_{k\in K_0} [\![\varphi_{ik}]\!].$$

Applying the induction hypothesis yields

$$\Gamma \vdash e : \bigvee_{k\in K_0} \varphi_{ik}$$

and using the rule for in we get

$$\Gamma \vdash \mathsf{in}_i(e) : \mathsf{in}_i(\bigvee_{k\in K_0} \varphi_{ik}) = \bigvee_{k\in K_0} \mathsf{in}_i(\varphi_{ik})$$

Since $i \in J_k$ for all $k \in K_0$ we have the following inequalities:

$$\bigvee_{k\in K_0} \mathsf{in}_i(\varphi_{ik}) \leq \bigvee_{k\in K_0} \sum_{j\in J_k} \mathsf{in}_i(\varphi_{ik}) \leq \bigvee_{k\in K} \sum_{j\in J_k} \mathsf{in}_i(\varphi_{ik}).$$

i.e., $\Gamma \vdash \mathsf{in}_i(e) : \bigvee_{k\in K} \sum_{j\in J_k} \mathsf{in}_i(\varphi_{ik})$. ■

Theorem 6.6.7 (Completeness for case.) *Let environments ρ_Γ, Γ be defined such*

that $\rho_\Gamma(x) = [\![\Gamma(x)]\!]$. *Then*

$$[\![\mathsf{case}\ e\ \mathsf{of}\ \{\overline{x_i}\ .\ e_i\}]\!]\rho_\Gamma \subseteq [\![\bigvee_{k\in K} \psi_k]\!] \ \Rightarrow\ \Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{\overline{x_i}\ .\ e_i\} : \bigvee_{k\in K} \psi_k.$$

Proof. There are two cases to consider, depending on the value of e. The first case is where the value of e is undefined which means that the whole case expression is undefined. We show that this is provable in the program logic too.

So assume that

$$[\![e]\!]\rho = (\bot, \ldots, \bot) = [\![\sum_{i\in\emptyset} \varphi_i]\!].$$

By the induction hypothesis this means that we can prove

$$\Gamma \vdash e : \sum_{i\in\emptyset} \varphi_i = \mathtt{f}$$

By the case $-\mathtt{f}$ rule we then get

$$\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{\overline{x_i}\ .\ e_i\} : \mathtt{f}$$

from which we can prove any property about case e of $\{\overline{x_i}\ .\ e_i\}$ using the weakening rule.

For the second case the assumption is

$$[\![e]\!]\rho_\Gamma = \bigcup_{l\in L} \downarrow\{\textstyle\prod_{j\in J_l} d_{jl}\} \quad \text{and} \quad \forall l \in L\ .\ \bigcup_{j\in J_l} \{[\![e_j]\!]\rho_\Gamma[\overline{x_j} : \downarrow\{d_{jl}\}]\} \subseteq [\![\bigvee_{k\in K} \psi_k]\!].$$

Define φ_{jl} by $[\![\varphi_{jl}]\!] = \downarrow\{d_{jl}\}$. We can now use the induction hypothesis to get

$$\Gamma \vdash e : \bigvee_{l\in L} \sum_{j\in J_l} \varphi_{jl} \quad \text{and} \quad \forall l \in L\ \forall j \in J_l\ .\ \Gamma[\overline{x_j} : \varphi_{jl}] \vdash e_j : \bigvee_{k\in K} \psi_k$$

and the rule for case gives that

$$\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{\overline{x_i}\ .\ e_i\} : \bigvee_{k\in K} \psi_k.$$

which proves the completeness of the logic for case. ■

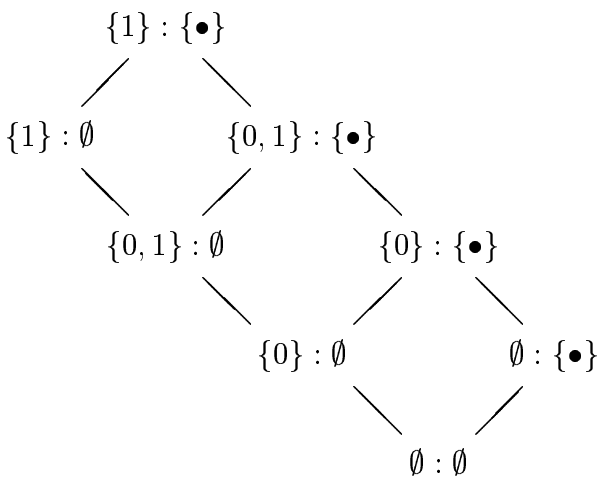


Figure 6.6: The partial order $\mathcal{J}(\mathsf{Int\ List})$

6.7 Lattices for recursive types

Using the semantics for sum types we can describe the Lindenbaum algebra of any algebraic data type. By definition we have that $\mathcal{LA}(\mu\alpha.\sum_i \sigma_i \times \alpha^{n_i}) = \mathcal{LA}(\sum_i \mathcal{P}(\sigma_i))$ so by modelling the last sum by a product we can extend the lower-set interpretation $[\![\ ]\!]$ of types to algebraic types. When lifting a lattice of convex sets we shall use the empty set $\emptyset$ as the bottom element so we extend the ordering $\sqsubseteq_{EM}$ by stipulating that $\emptyset \sqsubseteq_{EM} C$ for all convex sets C. We could have worked with an abstract bottom element for $P_C(\mathcal{J}(\sigma_i))_\perp$ but introducing the concrete set $\emptyset$ enables us to use some of the usual set operations as operations on the lattices. Furthermore $\emptyset$ is a natural choice of symbol to represent that there is no element from that summand in a structure.

Definition 6.7.1 (Lattices for algebraic data types)

$$[\![\mu\alpha.\sum_i \sigma_i \times \alpha^{n_i}]\!] = \mathsf{D}(\prod_{i\in I} P_C(\mathcal{J}(\sigma_i))_\emptyset).$$

As an example of a lattice describing a recursively defined data type we shall look at the lattice describing the type for integer lists defined by $\mathsf{Int\ List} \equiv \mu\alpha.\mathsf{Int}\times\alpha + \mathbf{1}$. The lattice modelling integer lists is built up from the lattices modelling sets of integers and sets of elements from the unit type $\mathbf{1}$. Sets of integers were considered in Section 5.4. The interpretation of $\mathbf{1}$ is defined by setting $\mathcal{J}(\mathbf{1})$ to be the one-point lattice, written as $\{\bullet\}$. The only non-empty convex subset of $\mathcal{J}(\mathbf{1})$ is $\{\bullet\}$ ifself.

Following Definition 6.7.1 the lattice modelling integer lists is defined to be the lower sets of:

$$\mathcal{J}(\mathsf{Int\ List}) = (P_C(\mathcal{J}(\mathsf{Int})))_\emptyset \times (P_C(\{\bullet\}))_\emptyset$$

The elements in $\mathcal{J}(\mathsf{Int\ List})$ are pairs (C, E) where the first component describes the set of elements in the lists and the second component describes what the lists ends in. We shall write such a pair in a more suggestive style as $C : E$ corresponding to the way lists are usually written. The partial order $\mathcal{J}(\mathsf{Int\ List})$ is shown in Figure 6.6.

The full lattice $[\![\mathsf{Int\ List}]\!]$ is obtained by forming all unions of the sets of this lattice. Thus we would have an element like

$$\downarrow\{\{1\} : \emptyset, \{0, 1\} : \{\bullet\}\}$$

representing the disjunctive property that either the list is infinite or partial or it contains an undefined element. This is the property corresponding to the property $\bot \in$ in the four-point domain of Wadler [Wad87]. The $\bot$ property in the four-point domain is here represented by $\downarrow\{\emptyset : \emptyset\}$, the $\top \in$ property is represented by $\downarrow\{\{1\} : \{\bullet\}\}$ and finally the property ∞ corresponding to all partial or infinite lists is represented by $\downarrow\{\{1\} : \emptyset\}$. So we have managed to reconstruct the elements from the four-point domain using uniform properties. However, the lattice so constructed contains elements that represent the same property of integer lists *e.g.*, the elements $\downarrow\{\{1\} : \emptyset\}$ and $\downarrow\{\{0, 1\} : \{\bullet\}\}$. This deficiency of uniform properties was already pointed out by Ernoult and Mycroft [EM91]. The presence of elements representing the same property does not jeopardise the correctness of the abstract interpretation. The same phenomenon has appeared in other analyses as reported by Cousot and Cousot [CC79] and by Hunt [Hun91] who also describe how to deal with this problem.

6.8 Abstract interpretation of fold and unfold

The abstract interpretations of the two operations associated with recursive data strucutres, fold and unfold, are defined over the lattices constructed in Section 6.7. In the definition, which is given in Figure 6.7 we shall describe the action of $[\![\mathsf{fold}]\!]$ and $[\![\mathsf{unfold}]\!]$ on ideals. The action on arbitrary lower sets is as usual defined to be the linear extension.

We now prove that extending the disjunctive logic from Chapter 4 with the rules for fold and unfold from Section 6.4 gives a logic which is sound and complete with respect to the abstract interpretation from Section 4.5 extended to fold and unfold as above.

$$[\![\mathsf{fold}]\!] : \mathsf{D}(\prod_I (\mathcal{J}(\sigma_i) \times \prod_I (P_C(\mathcal{J}(\sigma_i))_\emptyset))_\perp) \multimap \mathsf{D}(\prod_{i\in I} P_C(\mathcal{J}(\sigma_i))_\emptyset)$$

$$[\![\mathsf{fold}]\!](\downarrow\{\prod_{j\in J}(a_j,(S_1^j,\ldots,S_n^j))\}) \;=\; \bigcup_{j\in J}\downarrow\{(S_1^j,\ldots,\overline{S_j^j\cup\{a_j\}},\ldots,S_n^j)\}$$

$$[\![\mathsf{unfold}]\!] : \mathsf{D}(\prod_{i\in I} P_C(\mathcal{J}(\sigma_i))_\emptyset) \multimap \mathsf{D}(\prod_I (\mathcal{J}(\sigma_i) \times \prod_I (P_C(\mathcal{J}(\sigma_i))_\emptyset))_\perp)$$

$$[\![\mathsf{unfold}]\!](\downarrow\{(S_1,\ldots,S_n)\})$$

$$= \begin{cases} \{\perp,\ldots,\perp\} & \text{if } S_1=\ldots=S_n=\emptyset \\ \bigcup_{i=1}^{n}\bigcup_{\emptyset\neq S\subseteq S_i}\downarrow\{\prod_{\{i\}}(\bigsqcap S,(S_1,\ldots,C,\ldots,S_n^j)) \mid S_i\setminus S\subseteq C\subseteq S_i, C \text{ convex}\} & \text{otherwise.} \end{cases}$$

Figure 6.7: Abstract interpretation of fold and unfold.

Theorem 6.8.1 *The logical rules for* fold *are sound with respect to the interpretation* $[\![\ ]\!]$.

Proof. There are three rules to check. The first rule deals with the case where we fold an element from summand j onto a structure that contains no elements from this summand. Let J be an index set such that $i \notin J$. For the first rule we have to prove that

$$[\![\mathsf{fold}]\!]([\![\mathsf{in}_i(\varphi \times \sum_{j\in J}\mathsf{in}_j(\psi_j))]\!]) \subseteq [\![\mathsf{in}_i(\forall\varphi\wedge\exists\varphi)+\sum_{j\in J}\mathsf{in}_j(\psi_j)]\!].$$

The argument is the lower set

$$\downarrow\{\textstyle\prod_{\{i\}}(a_i,\prod_{j\in J}(S_j)) \mid \downarrow\{a\}=[\![\varphi]\!],\downarrow\{S_j\}=[\![\psi_j]\!]\}$$

and the condition $i \notin J$ implies that this is mapped by $[\![\mathsf{fold}]\!]$ to

$$\downarrow\{\textstyle\prod_{j\in J\cup\{i\}} S_j \mid S_i=\{a\},\forall j\in J\,.\,\downarrow\{S_j\}=[\![\psi_j]\!]\}$$

which is equal to

$$[\![\mathsf{in}_i(\forall\varphi\wedge\exists\varphi)+\sum_{j\in J}\mathsf{in}_j(\psi_j)]\!].$$

The second rule concerns folding an element from summand j onto a structure that already contains elements from this summand. J is still an index set such that $i \notin J$.

Let C_i, D_i be defined by

$$\downarrow\{C_i\} = [\![\forall \bigvee_{k\in K} \varphi_k \wedge \bigwedge_{k\in K} \varphi_k]\!],$$

$$\downarrow\{D_i\} = [\![\forall\varphi \vee \bigvee_{k\in K} \varphi_k \wedge \exists\varphi \wedge \bigwedge_{k\in K} \varphi_k]\!].$$

We have to show that $[\![\mathsf{fold}]\!]$ maps

$$[\![\mathsf{in}_i(\varphi \times \mathsf{in}_i(\forall \textstyle\bigvee_{k\in K} \varphi_k \wedge \bigwedge_{k\in K} \varphi_k) + \sum_{j\in J} \mathsf{in}_j(\psi_j))]\!] =$$
$$\downarrow\{\textstyle\prod_{\{i\}}(a, \prod_{j\in J\cup\{i\}} C_i) \mid \downarrow\{a\} = [\![\varphi]\!], \forall j \in J \downarrow\{C_j\} = [\![\psi_j]\!]\}$$

into the set

$$[\![\mathsf{in}_i(\forall(\varphi \vee \textstyle\bigvee_{k\in K} \varphi_k) \wedge \exists\varphi \wedge \bigwedge_{k\in K} \varphi_k) + \sum_{j\in J} \mathsf{in}_j(\psi_j)]\!] =$$
$$\downarrow\{\textstyle\prod_{j\in J\cup\{i\}} D_j \mid \forall j \in J \,.\, \downarrow\{D_j\} = [\![\psi_j]\!] = \downarrow\{C_j\}\}.$$

The definition of $[\![\mathsf{fold}]\!]$ gives that the argument is mapped into

$$\downarrow\{\textstyle\prod_{j\in J\cup\{i\}} C'_j \mid C'_i = \overline{\{a\} \cup C_i}, \forall j \in J \,.\, C'_j = C_j\}$$

so all we have to show is that

$$\overline{\{a\} \cup C_i} \in [\![\forall\varphi \vee \bigvee_{k\in K} \varphi_k \wedge \exists\varphi \wedge \bigwedge_{k\in K} \varphi_k]\!]$$

which follows from $a \in [\![\varphi]\!]$ and $C_i \in [\![\forall \bigvee_{k\in K} \varphi_k \wedge \bigwedge_{k\in K} \varphi_k]\!]$.

Finally the validity of the third rule for fold is proved as follows. The assumption is that the following set inclusion holds:

$$\forall j \in J \,.\, [\![\mathsf{fold}]\!](\downarrow\{\textstyle\prod_{\{j\}}(a_j, (S_1^j, \ldots, S_n^j))\}) = \downarrow\{(S_1^j, \ldots, \overline{S_j^j \cup \{a_j\}}, \ldots, S_n^j)\} \subseteq [\![\nu_j]\!].$$

From this we get that

$$\begin{aligned} [\![\mathsf{fold}]\!](\downarrow\{\textstyle\prod_{j\in J}(a_j, (S_1^j, \ldots, S_n^j))\}) &= \textstyle\bigcup_{j\in J} \downarrow\{(S_1^j, \ldots, \overline{S_j^j \cup \{a_j\}}, \ldots, S_n^j)\} \\ &\subseteq \textstyle\bigcup_{j\in J} [\![\nu_j]\!] = [\![\bigvee_{j\in J} \nu_j]\!] \end{aligned}$$

which is the conclusion of the third rule

■

Theorem 6.8.2 (Completeness for fold.) *Assume* $\mathsf{Irr}(\varphi_j), \mathsf{Irr}(\varphi_{ij}), \mathsf{Irr}(\psi_l)$. *Then (subscripts on dis- and conjunctions omitted):*

$$[\![\mathsf{fold}]\!]([\![\bigvee \sum_{j\in J} \mathsf{in}_j(\varphi_j \times \sum_{i\in I_j} \mathsf{in}_i(\forall \bigvee \varphi_{ij} \wedge \bigwedge \exists \varphi_{ij}))]\!]) \subseteq [\![\bigvee \sum_{l\in L} \mathsf{in}_l(\psi_l)]\!]$$
$$\Rightarrow \quad \vdash \mathsf{fold} : \bigvee \sum_{j\in J} \mathsf{in}_j(\varphi_j \times \sum_{i\in I_j} \mathsf{in}_i(\forall \bigvee \varphi_{ij} \wedge \bigwedge \varphi_{ij})) \;\to\; \bigvee \sum_{l\in L} \mathsf{in}_l(\psi_l).$$

Proof. Let a_j, S_i^j, T_l be so that

$$\downarrow\{a_j\} = [\![\varphi_j]\!], \quad \downarrow\{S_i^j\} = [\![\forall \bigvee \varphi_{ij} \wedge \bigwedge \exists \varphi_{ij}]\!] \quad \text{and} \quad \downarrow\{T_l\} = [\![\psi_l]\!].$$

The assumption is then that

$$[\![\mathsf{fold}]\!](\bigcup \downarrow\{\prod_{j\in J}(a_j, \prod_{i\in I_j} S_i^j)\}) \subseteq \bigcup \downarrow\{\prod_{l\in L} T_l\}.$$

Since $[\![\mathsf{fold}]\!]$ is linear (*i.e.*, distributes over unions) this implies that for every downwards closure in the union

$$[\![\mathsf{fold}]\!](\downarrow\{\prod_{j\in J}(a_j, \prod_{i\in I_j} S_i^j)\}) = \bigcup_{j\in J} \downarrow\{(S_1^j, \ldots, \overline{S_j^j \cup \{a_j\}}, \ldots, S_n^j)\} \subseteq \bigcup \downarrow\{\prod_{l\in L} T_l\}$$

that is, for each $j \in J$ there is a tuple $\prod_{l\in L} T_l$ such that

$$(S_1^j, \ldots, \overline{S_j^j \cup \{a_j\}}, \ldots, S_n^j) \sqsubseteq \prod_{l\in L} T_l.$$

From this we see that for all $S_i^j \neq \emptyset$, that is, for all S_i^j such that $i \in I_j$, the same i must appear in the index set L and if $i \neq j$ then $S_i^j \sqsubseteq_{EM} T_i$. The Completeness Theorem for the axiomatisation of uniform properties then allows us to conclude that

$$\forall j \in J\, \forall i \in I_j \setminus \{j\}\,. \quad \Rightarrow \quad \forall \bigvee \varphi_{ij} \wedge \bigwedge \exists \varphi_{ij} \leq \psi_i$$

We now have to consider what happens at index j where we actually make some changes to the set S_j^j. There are two possibilities: Either

1) $S_j^j = \emptyset$. In that case $\forall \varphi_j \wedge \exists \varphi_j \;\leq\; \psi_j$

or

2) $S_j^j \neq \emptyset$. In that case $\forall(\varphi_j \vee \bigvee \varphi_{ij}) \wedge \exists \varphi_j \wedge \bigwedge \exists \varphi_{ij} \;\leq\; \psi_j$

In case 1) we have

$$\mathsf{in}_j(\forall \varphi_j \wedge \exists \varphi_j) + \sum_{i\in I_j} \mathsf{in}_i(\forall \bigvee \varphi_{ij} \wedge \bigwedge \exists \varphi_{ij}) \;\leq\; \sum_{l\in L} \mathsf{in}_l(\psi_l)$$

and in case 2) we have

$$\mathsf{in}_j(\forall(\varphi_j \vee \bigvee \varphi_{ij}) \wedge \exists\varphi_j \wedge \bigwedge \exists\varphi_{ij}) + \sum_{i\in I_j} \mathsf{in}_i(\forall \bigvee \varphi_{ij} \wedge \bigwedge \exists\varphi_{ij}) \;\leq\; \sum_{l\in L} \mathsf{in}_l(\psi_l)$$

We then use the fact that we can always weaken the assumption about the result of a function to conclude that we can prove, in our program logic,

$$\vdash \mathsf{fold} : \sum_{j\in J} \mathsf{in}_j(\varphi_j \times \sum_{i\in I_j} \mathsf{in}_i(\forall \bigvee \varphi_{ij} \wedge \bigwedge \exists\varphi_{ij})) \;\to\; \bigvee \sum_{l\in L} \mathsf{in}_l(\psi_l).$$

for each of the disjuncts in the formula describing the argument to fold. We can then combine these facts into a conjunction with one conjunction for each disjunct:

$$\vdash \mathsf{fold} : \bigwedge[\sum_{j\in J} \mathsf{in}_j(\varphi_j \times \sum_{i\in I_j} \mathsf{in}_i(\forall \bigvee \varphi_{ij} \wedge \bigwedge \varphi_{ij})) \;\to\; \bigvee \sum_{l\in L} \mathsf{in}_l(\psi_l)].$$

and from the equality from the axiomatisation of functions

$$\bigwedge(\varphi_i \to \psi) = (\bigvee \varphi_i) \to \psi$$

we then obtain that

$$\vdash \mathsf{fold} : (\bigvee \sum_{j\in J} \mathsf{in}_j(\varphi_j \times \sum_{i\in I_j} \mathsf{in}_i(\forall \bigvee \varphi_{ij} \wedge \bigwedge \varphi_{ij}))) \;\to\; \bigvee \sum_{l\in L} \mathsf{in}_l(\psi_l).$$

which was what we wanted to prove. ■

We now turn to prove soundness and completeness for the unfold operation. The following lemma, which again builds on the corresponding lemma for the *extract* operation from Section 5.5, contains the essence of the proof.

Lemma 6.8.3 *Assume* $\mathsf{Irr}(\varphi_{ij})$ *and* $j \neq \emptyset$. *Then*

$$[\![\mathsf{unfold}]\!]([\![\textstyle\sum_{j\in J} \mathsf{in}_j(\forall \bigvee_{i\in I_j} \varphi_{ij} \wedge \bigwedge_{i\in I_j} \exists\varphi_{ij})]\!]) \quad =$$
$$[\![\bigvee_{j\in J} \bigvee_{\emptyset\neq I\subseteq I_j} \mathsf{in}_j(\bigwedge_{i\in I} \varphi_{ij} \times \mathsf{in}_j(\forall \bigvee_{i\in I_j} \varphi_{ij} \wedge \bigwedge_{i\in I_j\setminus I} \exists\varphi_{ij}) + \sum_{J\setminus\{j\}} \mathsf{in}_j(\forall \bigvee_{i\in I_j} \varphi_{ij} \wedge \bigwedge_{i\in I_j} \exists\varphi_{ij}))]\!].$$

Proof. Let us first introduce $S_1, \ldots, S_n$ such that

$$[\![\sum_{j\in J} \mathsf{in}_j(\forall \bigvee_{i\in I_j} \varphi_{ij} \wedge \bigwedge_{i\in I_j} \exists\varphi_{ij})]\!] = \downarrow\{(S_1, \ldots, S_n)\}$$

We first note that for all $j \in \{1, \ldots, n\} \setminus J$ we have $S_j = \emptyset$ so when applying the

definition of $[\![\mathsf{unfold}]\!]$ we get

$$\begin{aligned}
&[\![\mathsf{unfold}]\!](\downarrow\{(S_1,\ldots,S_n)\}) \\
&= \bigcup_{i=1}^{n}\bigcup_{S\subseteq S_i}\downarrow\{\prod_{\{i\}}(\sqcap S,(S_1,\ldots,C,\ldots,S_n^j)) \mid S_i\setminus S\subseteq C\subseteq S_i, C \text{ convex}\} \\
&= \bigcup_{i\in J}\bigcup_{S\subseteq S_i}\downarrow\{\prod_{\{i\}}(\sqcap S,(S_1,\ldots,C,\ldots,S_n^j)) \mid S_i\setminus S\subseteq C\subseteq S_i, C \text{ convex}\}
\end{aligned}$$

Lemma 5.5.2 states that

$$\bigcup_{\emptyset\neq S\subseteq C}\downarrow\{(\sqcap S,C') \mid C\setminus S\subseteq C'\subseteq C, C' \text{ convex}\} = [\![\bigvee_{\emptyset\neq I\subseteq J}(\bigwedge_{i\in I}\varphi_i)\times\forall(\bigvee_{j\in J}\varphi_j)\wedge\bigwedge_{j\in J\setminus I}\exists\varphi_j]\!].$$

From this equality it follows that

$$\begin{aligned}
&\bigcup_{\emptyset\neq S\subseteq C}\downarrow\{(\sqcap S,(S_1,\ldots,C',\ldots,S_n)) \mid C\setminus S\subseteq C'\subseteq C, C' \text{ convex}\} \\
&= [\![\bigvee_{\emptyset\neq I\subseteq J}(\bigwedge_{i\in I}\varphi_i)\times\mathsf{in}_j(\forall(\bigvee_{j\in J}\varphi_j)\wedge\bigwedge_{j\in J\setminus I}\exists\varphi_j)+\sum_{J\setminus\{j\}}\mathsf{in}_j(\forall\bigvee_{i\in I_j}\varphi_{ij}\wedge\bigwedge_{i\in I_j}\exists\varphi_{ij})]\!].
\end{aligned}$$

which proves the lemma.

■

Theorem 6.8.4 *The logical rules for* unfold *are sound and complete with respect to the interpretation* $[\![\mathsf{unfold}]\!]$.

Proof. The soundness part of the theorem is the "$\subseteq$" of the set equality in Lemma 6.8.3 above, together with the observation that

$$[\![\mathsf{unfold}]\!](\downarrow\{(\emptyset,\ldots,\emptyset)\}) = \downarrow\{(\bot,\ldots,\bot)\}.$$

verifying the soundness of the rule $\vdash \mathsf{unfold} : \mathtt{f}\to\mathtt{f}$.

Completeness amounts to showing that if

$$[\![\mathsf{unfold}]\!]([\![\bigvee\sum_{j\in J}\mathsf{in}_j(\forall\bigvee_{i\in I_j}\varphi_{ij}\wedge\bigwedge_{i\in I_j}\exists\varphi_{ij})]\!])\subseteq[\![\chi]\!]$$

then we can prove

$$\vdash \mathsf{unfold} : \bigvee\sum_{j\in J}\mathsf{in}_j(\forall\bigvee_{i\in I_j}\varphi_{ij}\wedge\bigwedge_{i\in I_j}\exists\varphi_{ij}) \;\to\; \chi.$$

But from the hypothesis and Lemma 6.8.3 it follows that

$$[\![\bigvee_{j\in J}\bigvee_{\emptyset\neq I\subseteq I_j}\mathsf{in}_j(\bigwedge_{i\in I}\varphi_{ij}\times\mathsf{in}_j(\forall\bigvee_{i\in I_j}\varphi_{ij}\wedge\bigwedge_{i\in I_j\setminus I}\exists\varphi_{ij})+\sum_{J\setminus\{j\}}\mathsf{in}_j(\forall\bigvee_{i\in I_j}\varphi_{ij}\wedge\bigwedge_{i\in I_j}\exists\varphi_{ij}))]\!]\subseteq[\![\chi]\!]$$

and by the completenes result for the axiomatisation of the entailment relation $\leq$ on sums and sets (Theorem 6.6.4 and Theorem 5.3.9), we can conclude that

$$\bigvee_{j\in J}\bigvee_{\emptyset\neq I\subseteq I_j}\mathsf{in}_j(\bigwedge_{i\in I}\varphi_{ij}\times\mathsf{in}_j(\forall\bigvee_{i\in I_j}\varphi_{ij}\wedge\bigwedge_{i\in I_j\setminus I}\exists\varphi_{ij})+\sum_{J\setminus\{j\}}\mathsf{in}_j(\forall\bigvee_{i\in I_j}\varphi_{ij}\wedge\bigwedge_{i\in I_j}\exists\varphi_{ij}))\leq\chi$$

Thus the judgement

$$\vdash \mathsf{unfold} : \bigvee\sum_{j\in J}\mathsf{in}_j(\forall\bigvee_{i\in I_j}\varphi_{ij}\wedge\bigwedge_{i\in I_j}\exists\varphi_{ij})\ \rightarrow\ \chi.$$

can be proved by an application of the rule for unfold followed by an application of the rule **Weak**.

■

This completes the proof of the correspondence between the logic for reasoning about the fold and unfold operations and their abstract interpretation.

6.9 Operations on lists

We now calculate the abstract interpretation of some of the standard operations on lists. To keep things simple we shall consider operations on integer lists. In addition to length and map studied in Section 5.4 we have *e.g.*, functions

$$\begin{array}{lcl}
\mathsf{cons} & : & \mathsf{Int}\times\mathsf{Int\ List}\rightarrow\mathsf{Int\ List}\\
 & \equiv & \lambda p.\mathsf{fold}\ \mathsf{in}_1(p)\\
\mathsf{head} & : & \mathsf{Int\ List}\rightarrow\mathsf{Int}\\
 & \equiv & \lambda l.\mathsf{case}\ \mathsf{unfold}\ l\ \mathsf{of}\ \mathsf{in}_1(x,xs)\ .\ x\ []\ \mathsf{in}_2(y)\ .\ \bot_\sigma\\
\mathsf{tail} & : & \mathsf{Int\ List}\rightarrow\mathsf{Int\ List}\\
 & \equiv & \lambda l.\mathsf{case}\ \mathsf{unfold}\ l\ \mathsf{of}\ \mathsf{in}_1(x,xs)\ .\ xs\ []\ \mathsf{in}_2(y)\ .\ \bot_{\mu\alpha.\sigma\times\alpha+\mathbf{1}}
\end{array}$$

where we use undefined ($\perp$) to model the occurrence of an error. The bottom element is not a part of the language but stands for any expression with an undefined value. Let us first calculate the abstract function for cons:

$$\begin{aligned}
[\![\mathsf{cons}]\!] &: \mathsf{D}(\mathbf{2} \times (P_C(\mathbf{2})_\emptyset \times P_C(\mathbf{1})_\emptyset)) \multimap \mathsf{D}(P_C(\mathbf{2})_\emptyset \times P_C(\mathbf{1})_\emptyset) \\
[\![\mathsf{cons}]\!]{\downarrow}\{(c,(C,E))\} &= [\![\mathsf{fold}]\!]@([\![\mathsf{in}_1]\!]@{\downarrow}\{(c,(C,E))\}) \\
&= [\![\mathsf{fold}]\!]@{\downarrow}\{((c,(C,E)),\perp)\} \\
&= {\downarrow}\{(\overline{C \cup \{c\}},E)\}
\end{aligned}$$

The following table shows what happens when adding an undefined resp. a defined element to an integer list described by the elements from the four-point domain. In the following table the abstract values of the list argument is listed vertically and the abstract values of the element being added to the list are listed horizontally.

$[\![\mathsf{cons}]\!]$	$\downarrow\{1\}$	$\downarrow\{0\}$
$\downarrow\{\{1\} : \{\bullet\}\}$	$\downarrow\{\{1\} : \{\bullet\}\}$	$\downarrow\{\{1\} : \emptyset, \{0,1\} : \{\bullet\}\}$
$\downarrow\{\{1\} : \emptyset, \{0,1\} : \{\bullet\}\}$	$\downarrow\{\{1\} : \emptyset, \{0,1\} : \{\bullet\}\}$	$\downarrow\{\{1\} : \emptyset, \{0,1\} : \{\bullet\}\}$
$\downarrow\{\{1\} : \emptyset\}$	$\downarrow\{\{1\} : \emptyset\}$	$\downarrow\{\{1\} : \emptyset\}$
$\downarrow\{\emptyset : \emptyset\}$	$\downarrow\{\{1\} : \emptyset\}$	$\downarrow\{\{0\} : \emptyset\}$

This is the table reported by Wadler [Wad87] except for the value in the lower right corner. When adding an undefined element to an undefined list, Wadler can only say that the ending is undefined, *i.e.*, that the result is ∞. Since we can represent $\forall$-properties we are able to express the additional fact that all elements of the resulting list are undefined.

The definitions of the functions head and tail share a common pattern by first unfolding an argument and then take action according to the type of the unfolded structure. The following equation will be used in computing $[\![\mathsf{head}]\!]$ and $[\![\mathsf{tail}]\!]$.

Assume that e_1, e_2 are of type τ and that $[\![e]\!]\rho = {\downarrow}\{C, E\}$. Then

$$[\![\mathsf{case\ unfold}\ e\ \mathsf{of}\ \mathsf{in}_1(x, xs)\ .\ e_1\ []\ \mathsf{in}_2(y)\ .\ e_2]\!]\rho$$
$$= \begin{cases} \{0_{[\![\tau]\!]}\} & \text{if } C = E = \emptyset \\ \bigcup\limits_{c \in C} [\![e_1]\!]\rho[x \mapsto {\downarrow}\{c\}, xs \mapsto {\downarrow}\{(\max(C, C \setminus \{c\}), E)\}] \cup \bigcup\limits_{\bullet \in E} [\![e_2]\!]\rho[y \mapsto \bullet]. \end{cases}$$

From this it is straightforward to calculate the result of applying the abstract functions $[\![\mathsf{head}]\!]$ and $[\![\mathsf{tail}]\!]$ to the downwards closure of a single element. As usual the extension to arbitrary lower sets is defined by linearity.

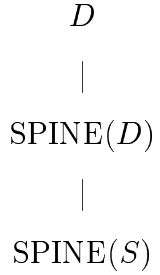
$$
\begin{aligned}
&[\![\mathsf{head}]\!] : \mathsf{D}(P_C(\mathbf{2})_\emptyset \times P_C(\mathbf{1})_\emptyset) \multimap \mathsf{D}(\mathbf{2}) \\
&[\![\mathsf{head}]\!]{\downarrow}\{(C,E)\} \;=\; \{0_{\mathsf{Int}}\} \cup \bigcup_{c \in C} {\downarrow}\{c\} \\
&[\![\mathsf{tail}]\!] : \mathsf{D}(P_C(\mathbf{2})_\emptyset \times P_C(\mathbf{1})_\emptyset) \multimap \mathsf{D}(P_C(\mathbf{2})_\emptyset \times P_C(\mathbf{1})_\emptyset) \\
&[\![\mathsf{tail}]\!]{\downarrow}\{(C,E)\} \;=\; \{(\emptyset,\emptyset)\} \cup \bigcup_{c \in C} \{(\max(C, C \setminus \{c\}), E)\}
\end{aligned}
$$

The following table gives the value of the abstract functions at the arguments corresponding to the elements in four-point domain. This is exactly the table for head and tail from Wadler's article [Wad87].

l	$[\![\mathsf{head}]\!]$	$[\![\mathsf{tail}]\!]$
$\downarrow\{\{1\} : \{\bullet\}\}$	$\downarrow\{1\}$	$\downarrow\{\{1\} : \{\bullet\}\}$
$\downarrow\{\{1\} : \emptyset, \{0,1\} : \{\bullet\}\}$	$\downarrow\{1\}$	$\downarrow\{\{1\} : \{\bullet\}\}$
$\downarrow\{\{1\} : \emptyset\}$	$\downarrow\{1\}$	$\downarrow\{\{1\} : \emptyset\}$
$\downarrow\{\emptyset : \emptyset\}$	$\downarrow\{0\}$	$\downarrow\{\emptyset : \emptyset\}$

6.10 Uniform properties and binding time analysis

Hunt and Sands [HS91] argue that binding time analysis can be formulated in terms of a constancy analysis and suggest the following three point domain for lists of booleans:



The elements represent equivalence classes of lists: D relates all lists to each other, SPINE(D) relates lists of the same length (including partial lists of the same length and infinite lists) and SPINE(S) only relates a list to itself. With these elements we can describe various degrees of constancy of a function f. For example, if f : BoolList $\rightarrow$ BoolList satisfies $f(D) = \text{SPINE}(S)$ then all lists are mapped into the same one-point equivalence class, *i.e.*, f is constant.

We mention this particular analysis because it gives an example of a collection of properties of a recursive data structure that cannot be constructed solely from uniform properties. The structure of the list (here: the length) forms an important part of the properties and cannot be described only by its content. However, this does not mean that we cannot find a program logic equivalent to this abstract interpretation, but only that we cannot use the framework constructed here. Instead we would have to construct the rules for fold and unfold in an *ad hoc* fashion using the technique from Chapter 2 to construct the abstract domains.

6.11 Summary

In this chapter we have presented an axiomatisation of uniform properties for recursive sum-of-products types that incorporates into one framework the type of properties considered by Wadler [Wad87] with those considered by Launchbury [Lau89] and Hunt and Sands [HS91]. We saw that this generalises the properties used in Wadler's strictness analysis but that it could not capture the structural information present in the properties used in Hunt and Sands binding time analysis. Also, the lattices constructed contain elements that represent the same properties of lists; a problem avoided in Wadler's construction.

The axiomatisation was used to define logical rules describing the operations fold and unfold on recursive data structures. We described the abstract interpretation corresponding to the program logic presented and proved that the abstract interpretation and the logical rules are equally strong in finding properties of the fold and unfold operations. Finally we compared the abstract interpretation of operations cons, head and tail with that given by Wadler and noticed that we have obtained a slight improvement with respect to the precision of the interpretation.

An important task that remains to be done is to prove that the analysis proposed here is correct. In order to prove the correctness of the analysis we would have to prove the soundness of the rules for fold and unfold with respect to the multi-set semantics for the recursive data structures which again is given in terms of the standard semantics and the *flatten* function from Section 6.2. The proof would

require a *concretisation* function γ that would map a uniform formula to the set of multi-sets it represents. For example, for e a closed expression of type $\mu\alpha\ .\ \sigma \times \alpha$ and with $S[\![\ ]\!]$ being the standard semantics we would have to prove the correctness property

$$\vdash e : \varphi \qquad \text{implies} \qquad \mathit{flatten}(S[\![e]\!]) \in \gamma(\varphi) \subseteq \mathcal{M}([\![\sigma]\!]).$$

To formalise this for recursively defined algebraic data structures in general would be a substantial task in itself. We shall leave this as work for the future.

Chapter 7

Conclusions

This chapter gives a summary of the thesis and assesses the results obtained. This is followed by a brief account of some topics related to what has been presented in the thesis together with some suggestions for future work.

7.1 Summary and appraisal

In this thesis we have studied the relationship between abstract interpretation and non-standard type systems. The idea underlying our approach is that an abstract interpretations can be viewed as a model of a program logic specified by a type system. By pursuing this idea we have been able to characterise the relationship between existing analyses and to develop new analyses by extending existing ones.

In Chapter 2 we studied the connection between the lattices used as domains in abstract interpretation and the formal systems of properties underlying a type system. We argued that the ideals in the lattices could model properties and that the elements of the lattices could be construed as filters of properties. This way of passing between logical systems and lattices was then formalised categorically and seen to be a special case of the theory of Stone Duality. By specialising proofs from this theory we proved that the lattices in abstract interpretation correspond to formal systems where we can express conjunctions of properties.

Underlying the approach taken here is the simple idea that properties should be understood as sets of elements and an element should be identified with the set of properties it satisfies. This makes the results presented applicable to a wide range of problems. Furthermore, our approach is not biased towards either the type system view or the abstract interpretation view. Since the results are based on a duality

between the lattices and the logical systems we can translate a logical system into a corresponding lattice and *vice versa*.

In Chapter 3 we gave an axiomatisation of the lattices used in abstract interpretation of the simply typed lambda calculus. With this axiomatisation we then defined a conjunctive program logic for strictness analysis of simply typed lambda calculus and proved it equivalent to the abstract interpretation strictness analysis of Burn, Hankin and Abramsky. An important conclusion here is that in order to attain the power of this abstract interpretation a type systems must be able to express conjunctive properties. It is furthermore shown that by changing the program logic slightly we can obtain a binding time analysis for typed lambda calculus.

It is important to emphasise that the results obtained only talk about the relationship between the analyses. In particular we have not studied the relationship between program logics and the standard semantics, *i.e.*, we have not provided a correctness proof for the logic. However, by the correspondence between logic and abstract interpretation we can, at least in the case of strictness and binding time analysis, rely on earlier correctness results for the abstract interpretation. By proving correctness via an abstract interpretation, as is done here, rather than directly with respect to a standard semantics we get the extra comparison between the two analysis techniques and their capability of detecting the properties we are analysing for.

In Chapter 4 we extend the conjunctive program logic to include disjunctions of properties. This is primarily motivated by the desire to express properties of data structures not expressible in the conjunctive logic. We prove a normal form theorem for the set of disjunctive formulae and guided by this define a model of the properties in which these are modelled by lower sets of lattices. We formulate an abstract interpretation where the abstract denotations are such lower sets and we show that the abstract interpretation of a term is the lower set modelling the strongest property provable about the term in a disjunctive program logic. This extends earlier work by Nielson, done in the setting of a first order language, to higher order functions. The chapter concludes with showing that our use of disjunctions corresponds to the use of tensor products in other abstract interpretation frameworks.

The approach to proving the equivalence between logic and denotation taken in Chapter 4 is slightly different from the one used in the earlier chapters since here we use the properties (the lower sets) directly as denotations instead of using the elements of these lower sets. The other approach is conceptually clearer but it would be more complicated to apply it here, hence the change. The analysis we arrive at handles more properties than existing abstract interpretations since it is able to represent disjunctive properties of function type. Whether this added strength is

important enough to justify a more complicated analysis remains to be seen. One restriction to consider would be to restrict the use of disjunctive properties to non-functional types thus arriving at analyses similar to existing analyses based on tensor products.

In Chapter 5 we consider a new class of properties called uniform properties. These properties are intended for analysing recursive data structures with respect to the elements they contain. We have two kinds of properties: *there exists an element in the structure satisfying a given property* and *all elements in the structure satisfy a given property*. We use the axiomatisation of the Plotkin powerdomain from Domain Logic to derive a formal system for reasoning about these properties and present a model of the uniform properties based on convex sets. The use of uniform properties is illustrated by giving an abstract semantics to operations of inserting and extracting elements of sets.

Uniform properties should be useful not only for our application to strictness analysis but also in data flow analysis of languages involving complex data structures where it would not be feasible to list the properties of each element in the structure. Although it can be argued that the approach to uniformity taken here is too restrictive we are convinced that this is a good starting point for constructing more elaborate systems of properties. The use of the Plotkin powerdomain in giving a semantic description of concurrent processes suggests that the uniform properties could also be used as a foundation for analysing the behaviour of concurrent languages.

In Chapter 6 the formal system of uniform properties is combined with a logic for reasoning about sum types to form a logic for strictness analysis of functions defined over recursive data structures. A data structure is now modelled by a collection of convex sets each describing the set of elements of a particular type. We axiomatise the operations fold and unfold associated with recursive data structures and show how to define a corresponding abstract interpretation. The analysis obtained can be seen as a reconstruction of Wadler's strictness analysis for lists based on the four point domain in terms of uniform properties. It can find more properties than Wadler's system due to the fact that we can express the property that all elements of a structure are undefined, but this happens at the price of a more complicated domain. Also the domain so constructed contains distinct elements that represent the same property of lists, this was avoided in Wadler's original construction. Finally, we see that the uniform properties cannot be used to construct the properties used in the binding time analysis considered earlier since these properties involve information about the structure of the data structure as well.

The logic for fold and unfold presented here gives a general axiomatisation of these operations. Practical applications might consider restricted versions of these rules.

Even if the full system is not put to use in an implementation the logical view and its correspondence with powerdomain theory can still be used to gain an understanding and a characterisation of other analyses of recursive data structures. For example if we only consider those of the uniform properties that talk about existence of elements then they can be modelled by the Smyth powerdomain which consists of upwards closed sets rather than convex sets. This again can explain the choice of upwards closed sets in Nielson and Nielson's generalisation of Wadler's analysis [NN92].

7.2 Related topics and further work

7.2.1 Polymorphism

Most functional languages in use have some degree of polymorphism built into their type system. For example, the function length considered in Chapter 6 works the same way regardless of what type the elements of the argument list have so it is reasonable to assign length a type like $\forall\alpha.\alpha\ \mathsf{List} \to \mathsf{Int}$ to express this uniform behaviour. But what should an analysis for a monomorphic language like the typed lambda calculus do when it is to analyse such a function? Clearly it cannot analyse the function at all the monomorphic instances of its type since the number of instances is infinite as soon as we introduce type constructors. One solution is to figure out which instances of the function are used in a program and then analyse each of them. A more economic solution is to exploit the strong regularity inherent in the polymorphism to try and deduce properties valid for all instances from properties valid for single instances. This is the idea behind polymorphic invariance introduced by Abramsky [Abr86]. There are two kinds of polymorphic invariance to be considered:

- polymorphic invariance of a property, *i.e.*, one instance of a function has a given property if and only if all instances have that property.
- polymorphic invariance of an analysis, *i.e.*, a particular analysis technique can find a property at one instance if and only if it can find that property at all instances.

The second kind of invariance is useful since we can then analyse the functions at their simplest instances (usually some base type) knowing that we will not obtain any more information by analysing other instances. The first kind of invariance is less useful since it might still be the case that the analysis can detect a property at

one instance but not at another and we have no means of predicting which instances we should choose to analyse. An extension of the second approach which calculates the abstract function at more complex types from the abstract function at the base types is considered by Hughes and Baraki [Hug88, BH89, Bar91].

The polymorphic invariance of strictness analysis by abstract interpretation was first shown by Abramsky [Abr86]. In a later paper with the author [AJ91] it was shown how this could be proved using a semantics for polymorphism based on Reynolds' Abstraction Theorem [Rey83]. In this semantics polymorphic types maps relations to relations and a polymorphic functions $f_\alpha : R(\alpha) \to S(\alpha)$ are shown to make the following diagram of relations

$$\begin{array}{ccc} A & & R(A) \xrightarrow{f_A} S(A) \\ \Big\downarrow r & & \Big\downarrow R(r) \quad \subseteq \quad \Big\downarrow S(r) \\ B & & R(B) \xrightarrow{f_B} S(B) \end{array}$$

semi-commute in the sense that relation $f_B \circ R(r)$ is included in the relation $S(r) \circ f_A$. The application to strictness analysis works by defining relations that relate the element denoting *"undefined"* at one type instance to that of another and then use semi-commutativity of the diagram to prove that $f_A(0_{R(A)}) = 0_{S(A)}$ if and only if $f_A(0_{R(B)}) = 0_{S(B)}$.

A refinement of this kind of invariance properties was studied by Benton [Ben92a] where it is observed that there are properties of polymorphic functions (such as *"maps strict functions to strict functions"* which are not polymorphically invariant over all instances but only *"at all instances for which that property makes sense"*[Ben92a, section 5]. This observation leads to consider polymorphic invariance results at subsets of all instances. Benton's approach is particularly relevant to this thesis because it proves invariance for a program logic rather than an abstract interpretation. The two approaches are reconciled with the results of this thesis since we can use one invariance result to prove the other. In the end it is a question of whether one prefers to argue on logics or their models.

7.2.2 Implementations

A reason often heard for *not* using abstract interpretation is that it is too slow. Some non-standard type systems, on the other hand, have been implemented efficiently by modifying the algorithm W [DM82] for ML type inference [Gom90, Hen91, Hen92].

In light of the results in this thesis this cannot come as a surprise. The type systems considered in *loc. cit.* do not have conjunctions of types and are therefore less expressive than an analysis based on abstract interpretation. A more reasonable comparison would be between abstract interpretation and type inference for conjunctive types. The latter is, however, an almost uncharted area, probably due to the fact mentioned in Section 3.5 that inferring principal intersection types for untyped lambda terms is undecidable. A semi-decidable algorithm based on unification is presented by Ronchi della Rocca [Roc88]. Bakel considers [vB92] two restrictions of the intersection type discipline but since neither of them is decidable this has not lead to further algorithmic results. The conjunctive analysis for typed lambda calculus presented here is decidable so it makes sense to ask for a type inference algorithm here. However, until some progress is made on inferring conjunctive types the alternative to an exhaustive search through the set of properties a term can satisfy is abstract interpretation.

7.2.3 Backwards analysis in logical form

Hughes [Hug87] has presented a technique for strictness analysis that is based on determining the *context* in which an expression is evaluated. A context describes what will be demanded of the result of an expression during the rest of the computation. For example, a context for an expression of type $\sigma \times \tau$ can say that only the first component of the resulting pair will be used. The analysis works by propagating the context of the program down to each variable occurring in the program, so if the program

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

is evaluated in a context where only the value of the first component will be demanded the analysis will tell us that the value of variable y will be demanded but not that of variable x. The function is therefore strict in y. Since the information is being propagated from the result of the program back to the input of the program this kind of analysis is usually called backwards analysis. The contexts used in backwards strictness analysis were formalised by Hughes and Wadler [WH87] using the notion of projection over a domain. An example of another kind of analysis formulated in backwards style is the usage count analysis for compile-time garbage collection presented in the author's joint work with Mogensen [JM90].

Let us consider what a logic corresponding to a backwards analysis would include. We can still work with judgements of the form

$$\Gamma \vdash e : \gamma$$

where γ is a formula describing the context and Γ is an environment mapping variables to contexts. These judgements should now be understood in the "backwards" manner: if the context of e satisfies γ then the context of variable x will satisfy $\Gamma(x)$. We would have rules like

$$[x : \gamma] \vdash x : \gamma \qquad \frac{\Gamma \vdash e : \gamma \times \mathtt{t}}{\Gamma \vdash \mathsf{fst}(e) : \gamma}$$

where we have taken $\mathtt{t}$ to mean the context that does not demand anything. The rule **Weak** that allowed us to strengthen the assumptions about variables and weaken the conclusions about a term must be "turned around" to say that

$$\frac{\Gamma \leq \Delta \quad \Gamma \vdash e : \gamma \quad \delta \leq \gamma}{\Delta \vdash e : \delta}$$

since our assumption is now on the term and the conclusion about the variables. When an expression contains more than one sub-expression we have to find a way of combining the contexts that a variable occurs in in the various sub-expressions. This operator is usually denoted & and must be defined for every new kind of context The rule for pairing would look like

$$\frac{\Gamma \vdash e_1 : \gamma \quad \Delta \vdash e_2 : \delta}{\Gamma \,\&\, \Delta \vdash (e_1, e_2) : \gamma \times \delta}.$$

For example, in the usage count semantics [JM90] the & operator was defined to be addition of the usage count of each context so that would give a derivation like

$$\frac{[x : 1] \vdash x : 1 \qquad [x : 2] \vdash x : 2}{[x : 3] \vdash (x, x) : 1 \times 2}.$$

A function is viewed by a backwards analysis as a context transformer that maps a context for the result of a function application to a context for the argument. A function type $\gamma \to \delta$ should then be understood as context δ being transformed into γ. Application would be described by the rule

$$\frac{\Gamma \vdash e_1 : \gamma \to \delta \qquad \Delta \vdash e_2 : \gamma}{\Gamma \,\&\, \Delta \vdash e_1\ e_2 : \delta}.$$

These ideas are not meant to be definite claims on how a backwards analysis in logical form necessarily should be formed. Rather they are intended to serve as a starting point for further investigations. Given the problems that backwards analyses have in dealing with higher order functions it would be worthwhile to see whether the type based approach has something better to offer here.

7.2.4 Other logical frameworks

Recently there has been some interest in using other logical frameworks for defining program logics. We shall briefly consider two of them: Wadler's linear types and Wright's Boolean Strictness Types.

Based on Girard's linear logic [Gir87] Wadler develops a series of type systems where the rules are taken from linear logic with the purpose of detecting which variables in a program get bound to shared data [Wad91]. The set of *linear types* is given by

$$U, V := X \mid !(U) \mid U \multimap V \mid U \otimes V \mid U \oplus V$$

A ! in front of a type means that the value of this type might be shared whereas a type without a ! indicates that the value is unshared[1]. For example the function *Twice* will be typed

$$\lambda f.\lambda x.f(f(x)) : !(X \multimap X) \multimap X \multimap X$$

since there are two references to the function bound to f. At first sight it seems plausible that this type system can fit into the framework of this thesis. For each type U we would have two properties U and $!(U)$ corresponding to the properties *"definitely unshared"* and *"possibly shared"*. The implication ordering would then be $U \leq !(U)$. However, there is a problem with two essential rules from the linear type system. The rules

$$\text{Promotion } \frac{!(A) \vdash v : V}{!(A) \vdash v : !(V)} \qquad \text{Dereliction } \frac{A, x : U \vdash v : V}{A, x : !(U) \vdash v : V}$$

are conflicting with our rule for weakening. They allow to weaken a formula on both sides of the turnstile whereas we only allow weakening on the conclusion side. Reversing the order to $!(U) \leq U$ does not help since now they allow us to strengthen formulae on both sides. In short there does not appear to be a view of linear types that will allow a straightforward application of the framework here to obtain a corresponding abstract interpretation. It is conceivable that a corresponding abstract interpretation can be designed based on Girard's coherence spaces [GLT89] which have been suggested as models of linear logic but to our knowledge coherence spaces have not yet been used in abstract interpretation.

Another type system for which an abstract interpretation seems to require new techniques is the system of Boolean Strictness Types devised by Wright [Wri91b,

[1] As is pointed out in the paper this intuition does not quite hold but it will suffice for the present discussion.

Wri91a]. The types here characterise functions according to how much of their argument will be needed during evaluation. Wright operates with a set of function type constructors among which are $\Rightarrow$, meaning that the function will reduce its argument to head normal form, $\not\rightarrow$, meaning that a function will not need its argument, and arrow variables $\rightarrow_1, \rightarrow_2, \ldots$. Wright imposes a boolean algebra structure on this set of arrows. In this way he expresses the fact that in the expression $\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ only one of e_1 and e_2 will be evaluated by giving if the type

$$\mathsf{if}\ : \mathsf{Bool} \Rightarrow \alpha_1 \rightarrow_1 \alpha_2 \neg \rightarrow_1 \alpha_3 \qquad (\alpha_3 \leq \alpha_1, \alpha_3 \leq \alpha_2).$$

Recent work by Baker-Finch [BF92] suggests that this type system can be explained in terms of relevance logic (which is linear logic with the Contraction rule added). For the moment, however, the abstract interpretations corresponding to these "ressource-sensitive" logics have yet to be formulated.

7.3 Final remarks

The area of program analysis has for some time witnessed a feud between two camps: one arguing the case for using abstract interpretation, the other advocating non-standard type systems. Type systems for program logic are praised because they are easy to formulate and to understand. They tell you *what* an analysis does. Abstract interpretations may appear more complicated and less straightforward to use. This is not surprising since they also tell you *how* to do an analysis.

The feud has partly been kept alive by the absence of a means of relating these two techniques. With this thesis we have provided a framework for comparing the expressive power of abstract interpretation with that of type systems for program analysis. We hope that this framework will lead to a reconciliation that will allow both sides to learn from the other part.

Bibliography

[Abr86] S. Abramsky. Strictness analysis and polymorphic invariance. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects*, pages 1–23, Berlin, 1986. Springer Verlag. Lecture Notes in Computer Science Vol. 217.

[Abr90] S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *Journal of Logic and Computation*, 1, 1990.

[Abr91a] S. Abramsky. A domain equation for bisimulation. *Information and Computation*, 92(2), 1991.

[Abr91b] S. Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51, 1991.

[AH87] S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[AJ91] S. Abramsky and T. P. Jensen. A relational approach to strictness analysis of higher order polymorphic functions. In *Proc. 18th ACM Symposium on Principles of Programming Languages.* ACM Press, 1991.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers. Principles, Techniques and Tools.* Addison—Wesley, 1986.

[Ban80] H.-J. Bandelt. The tensor product of continuous lattices. *Mathematische Zeitschrift*, 172:89–96, 1980.

[Bar84] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, revised edition, 1984.

[Bar91] G. Baraki. A note on abstract interpretation of polymorphic functions. In J. Hughes, editor, *Proc. of 5th ACM Conference on Functional Programming Languages and Computer Architecture*, LNCS vol. 523. Springer Verlag, 1991.

[Bar92] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, chapter 2, pages 117–310. Oxford University Press, 1992.

[BCD83] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, Dec. 1983.

[BDC91] F. Barbanera and M. Dezani-Ciancaglini. Intersection and union types. In T. Ito and A. Meyer, editors, *Theoretical Aspects of Computer Software*. Springer LNCS 526, 1991.

[Ben92a] P. N. Benton. Strictness logic and polymorphic invariance. In A. Nerode and M. Taitslin, editors, *Proceedings of the Second International Symposium on Logical Foundations of Computer Science, Tver, Russia*, LNCS vol. 620. Springer, 1992.

[Ben92b] P.N. Benton. *Strictness Analysis and Optimising Transformations for Lazy Functional Programs*. PhD thesis, Computer Laboratory, University of Cambridge, 1992. To be submitted.

[BF92] C. Baker-Finch. Relevant logic and strictness analysis. In *Proc. of WSA'92*, Bordeaux, 1992. BIGRE 81–82.

[BH89] G. Baraki and J. Hughes. Abstract interpretation of polymorphic functions. In *2nd Glasgow Workshop on Functional Programming*, University of Glasgow, Department of Computing Science, August 1989. Research Report 89/R4.

[BHA86] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory and practice of strictness analysis for higher order functions. *Science of Computer Programming*, 7:249–278, 1986.

[Bur91] G.L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1991.

[Bur92] G. L. Burn. Properties of abstract interpretation techniques. Technical Report Doc 92/19, Imperial College, Univ. of London, 1992.

[BW90] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall, 1990.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Proc. of 4th ACM Symposium on Principles of Programming Languages*, 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. 6th ACM Symposium on Principles of Programming Languages*, 1979.

[CC92] P. Cousot and R. Cousot. Abstract interpretation frameworks. Technical Report LIX/RR/92/10, Ecole Polytechnique, 1992.

[CDV80] M. Coppo, M. Dezani–Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrifft f. Mathematische Logik*, 27:45–58, 1980.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[DP90] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[EM91] C. Ernoult and A. Mycroft. Uniform ideals and strictness analysis. In *Proc. of 18th Int. Colloq. on Automata, Languages and Programming*, LNCS vol. 510. Springer, 1991.

[FH89] A.B. Ferguson and J. Hughes. An iterative powerdomain construction. In K. Davis and J. Hughes, editors, *Functional Programming. Proc. of the 1989 Glasgow workshop*. Springer Verlag, Workshops in Computing Series, 1989.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–101, 1987.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science vol. 7. Cambridge University Press, 1989.

[Gom90] C. Gomard. Partial type inference for untyped functional programs. In *Lisp and Functional Programming '90*, pages 282–287. ACM, 1990.

[Gom91] C. Gomard. *Program analysis matters*. PhD thesis, DIKU, University of Copenhagen, 1991. Available as DIKU Report 91/17.

[GS90] C. Gunter and D. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. North-Holland, 1990.

[Hen91] F. Henglein. Efficient type inference for higher–order binding–time analysis. In J. Hughes, editor, *Proc. of 5th ACM Conference on Functional Programming Languages and Computer Architecture*, LNCS vol. 523. Springer Verlag, 1991.

[Hen92] F. Henglein. Dynamic typing. In B. Krieg–Brückner, editor, *Proc. of 4th European Symposium on Programming*, LNCS vol. 582. Springer Verlag, 1992.

[HS91] S. Hunt and D. Sands. Binding time analysis: A new PERspective. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics–Based Program Manipulation*, 1991.

[Hud87] P. Hudak. A semantic model of reference counting and its abstraction. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[Hug87] R.J.M. Hughes. Analysing strictness by abstract interpretation of continuations. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[Hug88] J. 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.

[Hun91] L.S. Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, Univ. of London, 1991.

[HY86] P. Hudak and J. Young. Higher-order strictness analysis in untyped lambda calculus. In *Proc. of 13th Conference on Principles of Programming Languages*, 1986.

[Jen91] T. P. Jensen. Strictness analysis in logical form. In J. Hughes, editor, *Proc. of 5th ACM Conference on Functional Programming Languages and Computer Architecture*, LNCS vol. 523. Springer Verlag, 1991.

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

[JM89] S. B. Jones and D. Le Metayer. Compile-time garbage collection by sharing analysis. In *Proc. 4th. Int. Conf. on Functional Programming and Computer Architecture*. ACM Press, 1989.

[JM90] T. P. Jensen and T. Mogensen. A backwards analysis for conpile time garbage collection. In N. Jones, editor, *Proc. of 3rd European Symposium on Programming*, LNCS vol. 432. Springer Verlag, 1990.

[Joh82] P. T. Johnstone. *Stone Spaces*. Cambridge Studies in Advanced Mathematics vol. 3. Cambridge University Press, Cambridge, 1982.

[JSS89] N. Jones, P. Sestoft, and H. Søndergaard. Mix: A self—applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[KM87] T-M Kuo and P. Mishra. On strictness and its analysis. In *Proc. of the 14th ACM Conf. on Principles of Programming Languges*, 1987.

[KM89] T-M Kuo and P. Mishra. Strictness analysis : A new perspective based on type inference. In *Proc. 4th. Int. Conf. on Functional Programming and Computer Architecture*. ACM Press, 1989.

[Lau89] J. Launchbury. *Projection Factorisations in Partial Evaluation*. PhD thesis, Department of Computing, University of Glasgow, 1989.

[LM91] A. Leung and P. Mishra. Reasoning about simple and exhaustive demand in higher–order lazy languages. In J. Hughes, editor, *Proc. of 5th ACM Conference on Functional Programming Languages and Computer Architecture*, LNCS vol. 523. Springer Verlag, 1991.

[LW84] K. G. Larsen and G. Winskel. Using information systems to solve recursive domain equations effectively. In D. B. MacQueen G. Kahn and G. Plotkin, editors, *Semantics of Data Types*. Springer-Verlag, 1984. LNCS vol. 173.

[Mac71] S. MacLane. *Categories for the Working Mathematician*. Springer Verlag, 1971.

[Men79] E. Mendelson. *Introduction to Mathematical Logic*. Van Nostrand, 2nd edition, 1979.

[Mit91] J. Mitchell. Type inference with simple sub—types. *Journal of Functional Programming*, 1(3), 1991.

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

[Nie84] F. Nielson. *Abstract Interpretation Using Domain Theory.* PhD thesis, University of Edinburgh, 1984.

[Nie85] F. Nielson. Tensor products generalize the relational data flow analysis method. In *Proc. of the 4th Hungarian Computer Science Conference*, pages 211–225, 1985.

[NN92] H. R. Nielson and F. Nielson. The tensor product in Wadler's analysis of lists. In B. Krieg–Brückner, editor, *Proc. of 4th European Symposium on Programming*, LNCS vol. 582. Springer Verlag, 1992.

[Pie91] B. Pierce. *Programming with intersection types and bounded quantification.* PhD thesis, Carnegie—Mellon University, 1991. Available as CMU report CMU–CS–91–205.

[Plo76] G. D. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5(3):452–487, 1976.

[Plo81] G. D. Plotkin. Post-graduate lecture notes in advanced domain theory. Dept. of Computer Science, Univ. of Edinburgh, 1981.

[Rey83] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83.* North Holland, 1983.

[Roc88] S. Ronchi Della Rocca. Principal type scheme and unification for intersection type discipline. *Theoretical Computer Science*, 59, 1988.

[San90] D. Sands. *Calculi for Time Analysis of Functional Programs.* PhD thesis, Imperial College, September 1990.

[Sch78] J. Schwarz. Verifying the safe use of destructive operators in applicative programs. In *Proc. 3rd Int Symp. on Programming.* Dunod Informatique, 1978.

[Sch86] D. A. Schmidt. *Denotational Semantics.* Allyn and Bacon, 1986.

[Sco82] D. S. Scott. Domains for denotational semantics. In M. Nielson and E. M. Schmidt, editors, *Int. Conf. on Automata, Languages and Programming*, LNCS vol. 140. Springer-Verlag, 1982.

[Ses91] P. Sestoft. *Analysis and Efficient Implementation of Functional Programs.* PhD thesis, DIKU, University of Copenhagen, 1991. Available as DIKU Report 92/6.

[SP82] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Computing*, 11:761–783, 1982.

[Tay86] P. Taylor. *Recursive Domains, Indexed Category Theory and Polymorphism*. PhD thesis, Cambridge University, 1986.

[Ten91] R.D. Tennent. *Semantics of Programming Languages*. International Series in Computer Science. Prentice–Hall, 1991.

[vB92] S. v. Bakel. Complete restrictions of the intersection type discipline. *Theoretical Computer Science*, 102(1):135–163, 1992.

[Wad87] P. Wadler. Strictness analysis in non–flat domains (by abstract interpretation over finite domains). In S. Abramsky and C. Hankin, editors, *Abstract interpretation of declarative languages*. Ellis Horwood, 1987.

[Wad91] P. Wadler. Is there a use for linear logic? In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics–Based Program Manipulation*, 1991.

[WH87] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *1987 Conference on Functional Programming and Computer Architecture*, LNCS vol. 274. Springer, 1987.

[Wri91a] D. Wright. An intensional type discipline. Technical Report R91–3, Department of Computer Science, University of Tasmania, 1991.

[Wri91b] D. Wright. A new technique for strictness analysis. In *Proceedings of TAPSOFT91*, LNCS vol. 494. Springer, 1991.