UNIVERSITY
of
GLASGOW

Department of
Computing Science

A Refinement Calculus for Expressions

Sharon Flynn

Submitted for the degree of Doctor of Philosophy

September 1996

Thesis 10944
Copy 2

Abstract

This thesis describes a calculus intended for the refinement of expressions, in particular the calculus provides a framework for the formal derivation of executable expressions from initial specifications. The approach taken follows and extends the work of Back, Morris and Morgan on the refinement calculus for imperative style programs. We contribute to the area by providing a refinement calculus of expressions with a simple semantics and support for the formulation and development of specifications in parts.

We take the view that a refinement calculus consists of: a specification language, which usually includes constructs which are non-executable, but is a "super-language" of a programming language; a refinement relation between specifications, which possesses particular properties necessary for the refinement of specifications in a stepwise and piecewise manner; and a set of laws determining how such refinements may proceed.

We describe a simple functional language of expressions which includes features for undefinedness, non-determinism and partiality. The added constructs allow the easy formulation of expressive and abstract specifications, giving maximum freedom to the implementor.

The issue of methods to structure large specifications is addressed through the concept of partiality. We provide support for the construction of specifications in parts, together with operations to compose partial specifications to form the whole. We also consider how the state and exception monads, used to hide imperative features in pure functional programs, might be used similarly to structure specifications.

A refinement relation between specifications is defined. A set of laws suitable for the manipulation and refinement of expressions is proposed.

The expression language is given a simple denotational semantics, using powerdomain structures to capture non-determinism. This semantics allows the easy and intuitive formal definition of refinement, using the Smyth ordering for powerdomains, and facilitates the construction of the proofs of the proposed laws for the calculus.

Contents

1 Introduction and Background **1**

1.1 Specification 1

1.1.1 Approaches to Specification 2

1.1.2 Undefined terms 3

1.1.3 Non-Determinism 3

1.2 Program Development and Refinement 4

1.2.1 Refinement Calculi 5

1.2.2 Refinement of Expressions 7

1.3 Structuring Large Specifications 10

1.4 Thesis Proposal and Plan 11

1.4.1 Plan of Thesis 12

2 The Expression Language **14**

2.1 General Overview 15

2.1.1 Scope of the Language 15

2.1.2 Undefinedness 16

2.1.3 Non-Determinism and Partiality 17

2.1.4 Equivalence and Refinement 19

2.1.5 The Type System 20

2.1.6 Treatment of the Language 21

2.2 Undefinedness and Non-Determinism Formally 22

2.3 The Logic 24

2.3.1 Predicate Logic 26

2.3.2 Theorems 29

2.3.3 Sufficient Axiomatisation 29

2.4 The Type System 33

2.4.1 Integers 33

2.4.2 Characters 34

2.4.3 Products 35

2.4.4 Functions 36

2.4.5 Sets 37

2.4.6 Bags 40

2.4.7 Sequences 43

2.4.8 Partial Mappings 46

2.4.9 Simple Types 47

2.5 Language Constructs 47

2.5.1 Conditional Expressions 47

2.5.2 Local Definitions 48

2.5.3 Recursive Functions 49

2.5.4 Specification Expressions 49

2.5.5 Assumptions and Partially Defined Functions 50

2.5.6 Inverse Functions 52

2.5.7 Generic Functions 52

2.6 Partiality 53

2.6.1 Potentially Partial Expressions 54

2.6.2 Managing Miracles . . . 55

2.7 Conclusions . . . 58

3 Making Specifications 60

3.1 Useful Functions . . . 60

3.2 The Form of a Specification . . . 62

3.2.1 Types in Specification Modules . . . 63

3.2.2 Syntax of Specifications . . . 65

3.3 Examples . . . 66

3.4 Conclusions . . . 71

4 Structuring Specifications 72

4.1 Partial Specifications . . . 73

4.1.1 Using Partial Functions . . . 74

4.1.2 Combining Partial Functions . . . 75

4.1.3 Manipulating Partial Functions . . . 76

4.2 A Printing Control Example . . . 76

4.2.1 Notation . . . 77

4.2.2 Problem Description . . . 80

4.2.3 Building the Specification . . . 80

4.3 Using Monads . . . 86

4.3.1 The State Monad . . . 87

4.3.2 The Exception Monad . . . 89

4.3.3 Combining State and Exceptions . . . 90

4.3.4 Monads in the Specification Language . . . 91

4.4 Conclusions . . . 93

5 Proofs and Refinement **95**

5.1 The Proof System 96

5.2 Transformation Laws 97

5.3 Refinement 100

5.3.1 Proving Refinements 103

5.3.2 Refinement Laws 104

5.4 Examples of Formal Reasoning 107

5.4.1 Simple Proofs 108

5.4.2 A Larger Example 109

5.4.3 Reasoning with Monads 113

5.4.4 Reasoning about Δ 114

5.4.5 Simple Refinements 115

5.4.6 Refinement with Recursion 116

5.4.7 The N-Queens Revisited 118

5.5 Towards Imperative Programming 125

5.5.1 Background to the Derivation Style 126

5.5.2 The Specification 127

5.5.3 Refinements 128

5.6 Conclusions 130

6 Semantics **133**

6.1 Methodology 134

6.2 Semantics of Expressions 136

6.3 Semantic Domains and Recursion 143

6.3.1 Cpo's and Fixpoints 144

6.3.2 Domain Constructors 145

6.3.3 Semantic Domains . . . 146
6.3.4 The Egli-Milner Powerdomain . . . 147
6.3.5 Recursive Function Definitions . . . 149
6.3.6 Semantics of Recursive Function Definitions . . . 151
6.4 Refinement . . . 152
6.4.1 The Smyth Ordering . . . 152
6.4.2 Semantics of Refinement . . . 154
6.5 Soundness . . . 154
6.5.1 Undefinedness and Non-Determinism . . . 154
6.5.2 The Equivalence Axioms . . . 156
6.5.3 Strictness Proofs . . . 157
6.5.4 Distribution . . . 159
6.5.5 Products and Functions . . . 160
6.5.6 Assumptions and Guards . . . 162
6.5.7 Generalised Choice and Biased Choice . . . 163
6.5.8 Recursion . . . 164
6.5.9 Refinement . . . 165
6.5.10 Inference Rules . . . 169
6.6 Semantics of Specification Modules . . . 169
6.7 Conclusions . . . 171

7 Discussion and Conclusions 173

7.1 A Refinement Calculus for Expressions . . . 173
7.2 Discussion . . . 174
7.2.1 Evaluation . . . 175
7.2.2 Applications . . . 176

7.3 Comparison to Other Work 177

7.3.1 Approaches to Formal Reasoning 178

7.3.2 Formal Program Development 180

7.3.3 Refinement of Expressions 181

7.4 Future Work 185

7.5 Final Remarks 191

A Theorems of the Logic 192

A.1 Theorems of Propositional Logic 192

A.1.1 Laws Depending on Proper Values 195

A.2 Theorems of Predicate Logic 197

B The Printer Control Specification 200

Acknowledgements

To my supervisor, Joe Morris, I am indebted for his inspiration, support, encouragement, time and patience. I would like to thank the other members of my supervisory committee, Muffy Thomas, Kieran Cleneghan and David Watt, for their input.

I am very grateful to my parents, Elizabeth and Tony Flynn, and my sister, Joanne, for their constant support, financial and otherwise, over the last number of years. Thanks also to my husband John, for understanding, and being very very patient.

I would like to express my gratitude towards a number of people who gave of their time to advise me on certain issues, including: John Launchbury, Alex Ferguson and Samson Abramsky, for clarification on powerdomain theory; Michael Butler, Graham Hutton, Mary Sheeran and various anonymous referees for their feedback and direction; and the members of the Formal Methods and Theory Group for general advice. I also wish to thank Alex Bunkenburg, my true peer, for many useful discussions. On the technical side, *i.e.* LaTeX, Aran Lunzer, Will Partain and Micheál O'Foghlú were instrumental in getting things working.

In dcs, Glasgow, I have to thank: Aran, Alex and Stephane of the nightshift; the various inmates of F101; lots of ex-flatmates including Anne, Alison, Jackie, Steve, Mounia, Finbarr and Aran; Artur for proofreading services; Michelle and Sheila for getting it submitted; and Tom Melham for advice at a very low point in its completion. In UCG, I am very grateful to: Gerry Lyons for allowing me the time and space to finish up; Jim Duggan another proofreader; and Conn Mulvihil for suggesting that I do a Ph.D. in the first place!

I wish to acknowledge EPSRC for supporting this work, and the department of computing science, University of Glasgow, for the studentship which made living possible.

Finally, I wish to dedicate this thesis to the late Dr. John J. Kelly, of UCD, who was my mentor and M.Sc supervisor.

Declaration

The material presented in this thesis is the product of the author's own independent research carried out at Glasgow University, under the supervision of Joseph M. Morris of the Department of Computing Science.

Some of the basic ideas have been presented at the Glasgow Functional Programming Workshop, Ayr, 1994, as part of an example derivation, with Alexander Bunkenburg as co-author [19].

Chapter 1

Introduction and Background

Formal methods for the development of reliable software is an ambitious goal, but we take the view that it is ultimately worthwhile. Research into the area has resulted in many useful methods, for example: tools for the writing of unambiguous specifications of software systems; methods of verifying that a program meets its specification; semantics of programming languages which help us to understand the meaning of a program; laws which encapsulate the process of program development. That a program should be derived from its formal specification, so developing program and proof of correctness together, seems intuitively obvious. This is exactly the aim of research into formal program development, particularly refinement calculi. At the very least, it provides us with an understanding of the concepts and issues involved, and defines a common framework within which both specifications and programs can be discussed.

Our aim is to describe a refinement calculus of expressions, so extending the imperative refinement calculus and providing a theoretical basis for the formal development of functional programs. In this chapter we give a brief account of the background areas of specification, formal program development and refinement, and attempt to indicate how the refinement calculus of expressions fits into this context.

1.1 Specification

A specification of a software system is a description of the desired behaviour of that system. It can be thought of as a contract between a customer and a programmer. It must be written in such a way that it can be understood by the customer, but is rigorous enough to exclude ambiguities. Natural language is not suitable as a specification language because it allows too many ambiguities. However, a programming language is too restrictive as a specification language since it gives too much detail about *how* a task is to be accomplished.

We expect a specification to be more abstract and less machine-oriented than a program which implements it. Formal specifications are written using languages which are based on mathematical principles, and are therefore rigorous, but have a notation rich enough to express properties of a system in a way which is easily understood by the customer.

The existence of a formal specification also allows us to establish that a program implements that specification. A statement concerning the correctness of a program presupposes the existence of an external frame of reference. The formal specification may be used either to prove the correctness of a program, or in the development process to derive a program which satisfies the specification.

1.1.1 Approaches to Specification

Model-Oriented Specification

The specification languages Z [27, 75, 44] and VDM [10, 11] are both examples of a model-oriented approach to specification. This involves the construction of a model of the concept to be described, taking advantage of available mathematical tools. The associated operations of the concept are then specified with respect to the particular model which has been used.

The Z specification language follows an approach to specification which is state-based. It has as its mathematical basis familiar mathematical concepts and notations such as set theory and first order predicate logic. It uses the set operations such as union, intersection, set difference, set membership *etc.* , and operations on mappings between sets to build a conceptual model of the system to be specified. Operations from predicate logic are used to build sets and to make assertions concerning the components of the specifications.

The known properties of the underlying mathematical concepts used for specification in the model-oriented approach can be used to reason about specifications in a formal setting. The Logic of Partial Functions (LPF) provides a logical framework for proofs about VDM specifications [43].

Algebraic Specification Techniques

The algebraic style of specification, as found for example in [70], is theoretically based on the notion of algebraic types. In contrast to the model-oriented approach of Z or VDM, concepts are specified implicitly by describing their construction, modification and access operations using sets of axioms. Thus the internal structure of the concept is not explicitly revealed.

The advantage of an algebraic specification is that a more abstract description of the system is obtained. Although no explicit model of the concept is formulated, there may be many models which satisfy the specification. A programmer is not restricted to any particular model and may choose between possible models during the program development process.

However, the axiomatic equations to describe the system are difficult to construct. In addition, it is often the case that a particular implementation for a data type suggests itself and it is then easier to specify the data type in terms of that model.

1.1.2 Undefined terms

In the specification and development of software systems undefined expressions arise quite naturally, usually in the application of functions to arguments where the function is not defined, or termination is not guaranteed. Simple examples of this are integer division by zero, or the integer square root of a negative number. This necessitates a method for dealing with formulae which involve undefined terms. Many examples illustrating the need for such methods may be found in the literature, for example [9, 22, 40, 41]. It is clear that classical logic is unable to deal with such terms.

There are various ways of forming proofs about undefined expressions. Some of these attempt to keep to classical logic by making functions everywhere defined over a restricted domain, or by using relations to avoid function application, as in the Z specification language. Other methods use conditional forms of the familiar conjunction and disjunction operators, as in many programming languages, resulting in non-symmetric operators. Another method is to use a logic which has the ability to deal with terms which are not well-defined, a 3-valued logic such as LPF of VDM. An overview of various methods of dealing with the problem of undefined terms may be found in [22] and more recently in [42].

Our approach, as developed in chapter 2, is to admit to the existence of undefined terms and to use a logic, distinct from LPF, which accomodates them.

1.1.3 Non-Determinism

An expression is deterministic if separate evaluations of that expression, in the same environment, always give the same result. An expression is non-deterministic if separate evaluations may give different results. Constructs for non-determinism are used in specifications to increase abstractness, when there may be a number of design options which are equally suitable. During program development, this allows freedom for design decisions to be made. We take the view that, ultimately, programs must be deterministic.

Non-determinism may be used as a specification tool for *under-specification* of a problem. An often used example is that of searching. *Find the index of some occurrence of x in the list L.* This gives the implementor freedom to search for the first, last or any occurrence of the given x.

Non-determinism in specifications is usually obtained through the introduction of a choice operator $[\!]$, such that for expressions E and F, the expression $E \, [\!] \, F$ may evaluate to either the value of E or the value of F. We take the view that, from a specification $E \, [\!] \, F$, the customer will be happy with a program implementing E or a program implementing F or some combination of the two.

In [84] three sorts of non-deterministic choice operator for expressions are distinguished by the way choices are made in the presence of undefinedness. With *angelic* non-determinism, all choices are made in favour of termination, i.e. $E \, [\!] \, F$ is undefined only when both E and F are undefined. With *demonic* non-determinism, all choices are made in favour of non-termination, i.e. $E \, [\!] \, F$ is undefined if either of E or F is undefined. With *erratic* choice, nothing is done to favour or avoid non-termination. The terms *angelic* and *demonic* are attributed to C.A.R. Hoare, while the term *erratic* is due to M. Broy.

Although erratic non-determinism can be described operationally as being similar to the tossing of a coin, notice that it cannot be used to specify such a process. This is because, for example, the specification *heads* $[\!]$ *tails* may be implemented by the program *heads*, which always gives the same result.

In chapter 2, we introduce a specification language of expressions which includes a choice operator. In order not to limit the properties of the language un-necessarily, this choice operator is erratic. Our logic, which handles undefined terms, also accomodates non-deterministic values.

1.2 Program Development and Refinement

Given the formal specification of a program, the programmer's objective is to develop a program which satisfies the specification. The task of verifying a program after its construction is a laborious one, and it is well recognised that a program and the proof of its correctness should be developed together. This may be done in an informal manner, however in order to build programs which are correct with respect to their specifications, it is necessary to validate rigorously each step of the process.

In [24] Dijkstra describes a simple imperative programming language, the language of *guarded commands*. A methodology is presented in [24, 31, 38, 45] which allows the program and proof of its correctness to be developed together, from a specification consisting of a pre- and post-condition. A program development methodology for Z specifications is described in [75]. This uses a notion of refinement of both data and operations. The weakness of these, and other programming methodologies, is that while both the specification and the program are formal objects, in refining from specification to program, the intermediate objects are not necessarily formal, since they may be considered as hybrids, a mixture between specification and program.

The problem of having informal aspects in the development process is addressed by using a specification language which is a "superlanguage" of a programming language. The advantage of this is that both program and specification may be reasoned about using the same semantic framework. This is the case with the Extended ML specification language, which has, as its executable sublanguage, the Standard ML programming language [78, 79]. In [79] a formal program development methodology is presented which describes how a specification may be developed in stages by replacing non-algorithmic elements by executable code. Each step of the development is associated with certain proof obligations. The development process effectively describes the refinement of a specification such that the final specification is executable, *i.e.* a program.

Expressions are much easier to manipulate than statements, because we are no longer concerned with possible side-effects or changes to the state. This can be seen very clearly in reasoning about pure functional programs [12] and in the work of Bird and Meertens [13, 5] on manipulating lists. More recently, Bird has used notations from category theory [8] to specify concisely and very elegantly certain classes of problems [14, 15, 16]. Using mathematics of category theory these specifications can be transformed to equivalent but more efficient expressions of a functional programming language. Some work is involved in formulating the initial specifications and the notation could not be considered suitable for a naive user to read. The approach is also limited to a certain class of optimisation problems.

1.2.1 Refinement Calculi

The main aim of a refinement calculus is to allow the stepwise development of programs from specifications in a formal manner, ensuring a correct transformation. One approach to such a calculus is achieved by describing a specification language which contains as sublanguage a programming language. The specification language will, in general, contain some constructs which are very expressive but are non-executable or expensive to implement. This is usually obtained by extending a programming language with additional expressive, but possibly non-algorithmic, constructs. An example of this is the Extended ML specification language, mentioned above.

The calculus will also include a refinement relation between specifications, usually written $X \sqsubseteq Y$ for specifications X and Y. This expresses the fact that whenever specification X is acceptable (to a customer) so also is specification Y, but Y is generally more algorithmic than X. We use the term *algorithmic* loosely here, to mean that Y is closer to being a program than X.

The purpose of a refinement calculus is to allow the stepwise calculation of a program from an initial specification, S_0. This means the development of a sequence of specifications, $S_0 \sqsubseteq S_1 \sqsubseteq \ldots \sqsubseteq S_n$, where each S_i, for $0 \leqslant i < n$, is refined by S_{i+1}, and S_n is a program. In order to conclude that S_n is a correct implementation of initial specification S_0, it is necessary that refinement is transitive. In fact, the refinement relation should be a preorder, so that if any of the $\sqsubseteq$ is replaced by $\equiv$ (equivalence) in the above sequence, we can still conclude that $S_0 \sqsubseteq S_n$.

It is also important that refinement can progress in a piecewise manner, so that refinement of part of a specification results in refinement of the whole specification. To facilitate piecewise refinement, it should be the case that the constructs of the specification language are monotonic with respect to refinement of subterms. So, if $S[X]$ is a specification containing X as subspecification, and it can be shown that $X \sqsubseteq Y$, then it should be the case that $S[X] \sqsubseteq S[Y]$.

The final part of the calculus is a set of refinement laws. In deriving a program from its specification, it is not necessary to use the definition of refinement directly. Instead, the definition is used to form a set of refinement laws, which can be used to justify each step of the derivation.

The Refinement Calculus for Imperative Programs

A refinement calculus for imperative programs was first inspired by Back [3, 4], and further developed, independently, by Morris [59, 62] and Morgan [55, 56]. Dijkstra's *guarded command* language [24], whose semantics is given in terms of predicate transformers, is extended by adding expressive but non-executable constructs, including a specification statement consisting of a pre- and a postcondition. The added constructs are also given a formal semantics in terms of predicate transformers. The refinement relation between specifications is formalised, and intuitive notions of program development are described formally, resulting in a set of refinement laws.

Non-determinism, which is an important aspect of specification, is permitted in the imperative refinement calculus at the level of statements only. Non-deterministic expressions are not permitted. Morris [63] argues that expressions which are undefined or non-deterministic can fit into the refinement calculus for imperative programs by defining a suitable semantics. His approach results in an elegant form of assignment, but does not accommodate expressions which are of function type.

Data Refinement

In extending the guarded command language of Dijkstra to form a specification language, a richer set of data types is added along with richer operations on data. This facilitates specification using the model-oriented approach. During the refinement process, these richer types must be replaced with simpler and more easily implementable types. This process is known as data refinement.

Replacement of abstract data types by more concrete types using coordinate transformations was suggested by Dijkstra in [24]. A formal notion of data refinement with laws governing its application has been developed by Morris [60, 61] and Morgan [58] to compliment the imperative refinement calculus.

1.2.2 Refinement of Expressions

It is recognised that expressions are easier to manipulate than statements, and we have already mentioned the use of functional programming languages, and the work of Bird and Meertens. Refinement of expressions was excluded from the work on the imperative refinement calculus, although Morris [64, 65] has since done some research in the area. The ability to write non-deterministic, more abstract expressions at the specification stage, and to allow these be refined along with the refinement of statements would greatly extend the power of the imperative calculus.

It is also possible to consider writing an initial specification as an expression and, by refinement, calculate an imperative program to implement it. This would involve a special form of expression refinement since it would mean transforming from one type, the type of the specification expression, to the type of statements.

In pure functional programming [12] a program is essentially an expression which is evaluated by the computer. The task of a programmer is to build a function to solve a particular problem. A notion of refinement of expressions therefore could be used not only to increase the power of the imperative refinement calculus, but also as the basis of a refinement calculus for functional programs.

Logical Specifications for Functional Programs

In [68] Norvell and Hehner present an approach to expression refinement, with the aim of deriving functional programs. As with the approach used for the imperative refinement calculus, they take a simple programming language of expressions, and extend it by adding non-executable constructs. Non-determinism is achieved through the use of bunches [38, 39], resulting in an erratic form of choice. Bunches are similar to sets, but without the bracket notation, without nesting, and with distribution of operations over the elements.

Function abstraction, in the specification language of [68], distributes over bunch union, resulting in functions which are *under-determined* rather than non-deterministic. Essentially, what this means is that a function with a non-deterministic body is exactly equivalent to a choice between functions with deterministic bodies. Therefore it can be assumed that every function has a deterministic body.

The identity of bunch union is the *null* specification which refines all specifications, but cannot be implemented. The zero of bunch union is the *all* specification which is refined by all specifications. There is no explicit treatment of undefinedness, although *all* may be used to represent errors. The notion of refinement is based on the superbunch operator.

The semantics for the language is axiomatic, but there is no satisfactory treatment of recursion. In particular, examples of refinements are given which introduce recursive functions without any theoretical basis for doing so.

The approach of Norvell and Hehner results in a simple treatment of expression refinement at a syntactic level, but it does not address the problems which exist at a deeper level. The specification language is concise, but the notation is somewhat difficult to read, and the examples given are all small examples, of the searching and sorting variety. It is not clear how the language would be used to describe bigger problems, or how refinement in parts would be achieved.

A Refinement Calculus for Nondeterministic Expressions

In his PhD thesis [90], Ward gives a fuller account of a refinement calculus of expressions with a view to deriving functional programs. As in the work of Norvell and Hehner, he takes a simple functional programming language and extends it with non-executable constructs. Interesting additions include constructs for both demonic and angelic non-determinism.

The inclusion of angelic non-determinism means that backtracking problems can be expressed quite elegantly. This is because the evaluation of an expression involving angelic non-determinism in some sense *looks ahead* and chooses the correct value to give the desired result.

Ward gives a semantics to the specification language based on a notion of weakest preconditions for expressions. While in the imperative refinement calculus statements are regarded as functions from output states to input states, Ward treats expressions as functions from sets of values (evaluations) to sets of environments. We consider that the resulting semantics is unnecessarily complicated. The weakest precondition semantics is very suitable for a state-based language, but is not required to give a meaning to expressions.

Based on the semantics of the specification language, Ward gives a definition of the refinement relation between expressions and proposes a set of refinement laws, most of which are intuitively reasonable. However, because of the overcomplicated semantics, the proofs of these laws seem more involved than expected.

Although this work results in an expressive specification language, and a formal notion of refinement with associated laws, it is not clear how it would be used to tackle large problems. Ward does not address the issues of structuring large specifications, which is essential for any specification language.

Refinement of Imperative Expressions

In his Ph.D. thesis [18], Bunkenburg describes a calculus of expressions which has as target language an expression language with imperative threads. Although the aim of the calculus, to derive imperative style programs from functional specifications, is different from that of Ward or Norvell and Hehner, some of the approaches and techniques are similar.

Bunkenburg begins by laying out a language of expressions which includes a choice operator $\sqcap$ for demonic non-determinism. Non-terminating, or undefined, expressions are also considered, with lazy function application. Bunkenburg claims that a lazy language is more expressive.

Imperative programming techniques are permitted in the language through the inclusion of the state monad (see chapter 4). The algebraic laws asociated with the monad are included with the laws governing the expression language.

Informally, Bunkenburg treats non-deterministic expressions as sets of outcomes which are upward closed (with respect to definedness). An upward closed set is such that, if the set contains an outcome v, then it also contains all outcomes better (more defined) than v. The refinement relation is then treated as superset between upward closed sets of outcomes. Bunkenburg provides many axioms describing the behaviour of the refinement relation.

A denotational semantics is given to the language, again using upward closed sets, but this time in a formal manner. Bunkenburg states that a programmer needs the semantics to write the initial specificaiton but not for the derivation of a program. The semantics are needed to decide what to prove, but not in order to complete the proof.

The resulting semantics (for the non-imperative features of the language) is reasonably straightforward, using notations and theory from powerdomain theory. It is also possible, by extending the notation and imposing some restrictions, to give a denotational semantics to the state monad within the same framework.

Bunkenburg demonstrates the use of his calculus in a number of interesting examples from various problem domains. These are all concerned with the use of state threads in imperative-style expressions, rather than with basic expressions themselves. Consequently, it is difficult to compare the use of the calculus with that of the pure expression refinement approach of Ward.

1.3 Structuring Large Specifications

For large, or even medium sized, specifications and programs it becomes essential to have some method of structuring the specification into individual units. One of the most important features of Z is that it supports the decomposition of large specifications into manageable units, called *schemas*. Each schema should model a conceptual unit of the specification so that it is relatively self-contained, and can be reasoned about individually. This process may be described as "separation of concerns". A number of operators, such as conjunction and disjunction, are defined for combining schemas, in a sensible manner, to form the complete specification.

In the algebraic approach, type definitions may be structured so that each type declaration represents a conceptual unit of the specification. Specifications are built in an hierarchical fashion, allowing object classes to be defined in a structured way.

In [30], Frappier, Mili and Desharnais present a method to promote program construction by parts. Given a number of user requirements in the form of partial specifications, a partial program is derived for each one. These are combined to form a program which satisfies all the requirements simultaneously. Specifications are represented by binary relations, and the derivation process is a stepwise transformation of relations.

Back and Butler, in [2], examine various summation and product operators in a higher order logic approach to the imperative refinement calculus, using category theory. At a more abstract level than [30], the summation and product operators can be applied to the composition of partial specifications.

1.4 Thesis Proposal and Plan

The aim of this thesis is to provide a refinement calculus suitable for the refinement of expressions. The calculus could be used in a number of ways: to extend the imperative refinement calculus by allowing specification and refinement using more abstract expressions; to provide the basis for a calculus to allow the development of imperative programs from specification expressions; or to provide the basis of a framework for the formal development of functional programs from specifications. The approach will parallel the work of Back, Morris and Morgan on the refinement calculus for imperative programs.

The first stage is to describe formally a simple specification language of expressions. This is based upon familiar expressions of well-understood types, such as booleans, integers, functions *etc.* Additional, less familiar constructs will allow the easy formulation of expressive and abstract specifications, giving maximum freedom to the implementor. In order to achieve more abstract specifications we allow non-determinism in expressions by providing a choice operator. We also aim to enable formal reasoning about and with expressions which may contain undefined terms.

So that the extended language can be used to specify real problems it is vital that we provide support for the construction of specifications in parts, together with operations to compose partial specifications to form the whole. We will show that it is possible to reason about and refine these partial expressions individually.

A refinement relation between expressions will be defined. As described in section 1.2, this is a preorder, allowing the refinement process to progress in a stepwise manner. We will show that constructs of the expression language are, with a few exceptions, monotonic with respect to refinement, allowing piecewise refinement to occur.

The last part of defining a refinement calculus involves the compilation of a set of laws which may be used in the derivation of an executable expression, without requiring the use of the definition of the refinement relation at each step. We aim to provide both equivalence laws, used in the manipulation of specifications, and refinement laws, which describe how expressions may be refined.

The expression language will be given a denotational semantics, with powerdomain structures to capture non-determinism. The aim of the semantics is to provide a model of the language which can be used to justify the axioms and rules of inference, and so demonstrating that the theory is consistent.

In general, we expect our specification language to look similar to that of Norvell and Hehner and that of Ward, although there will be some different constructs which we have found useful and more expressive in formulating specifications. In particular, the support of partial specifications extends both of these approaches. We feel that the denotational approach to the semantics of the language is more suitable than the weakest precondition approach of Ward. Although our semantics is similar to that of Bunkenburg, we discuss powerdomains only at the semantic level, and so the user is not required to have any knowledge of a model of upward closed sets. The simple semantics and ease with which refinement laws are proved will support the claim that the denotational approach using powerdomains is most suitable for a language of this form.

We hope to contribute to the area of formal program development by providing a refinement calculus of expressions with a simple semantics and support for the formulation and development of specifications in parts.

1.4.1 Plan of Thesis

In this chapter we have given some background to the area of formal methods for specification and development of software. We assume that the reader is familiar with the various approaches to formal specification, formal programming in the style of Dijkstra [24, 31, 38, 45], and the refinement calculus for imperative programs, as developed by Back, Morris and Morgan [3, 4, 59, 62, 55, 56].

In chapter 2 we will introduce the specification language of expressions, based on familiar mathematical expressions, but including constructs to handle undefinedness, and a choice operator to provide for nondeterministic expressions. We will also describe the logic which forms part of the language, and give an argument that it is sufficiently axiomatised. In addition we describe what it means for an expression to be partial and introduce operators for forming and totalising such expressions, so excluding miraculous specifications.

Chapter 3 describes how expressions are used to form specifications. The syntax of a specification, as a collection of expressions, is described informally; and a number of small examples is given to illustrate this.

In chapter 4 we address the issue of how to structure large specifications. In particular, the formation, use and combination of partial functions as the units of partial specifications is examined. This is accompanied by a larger example to illustrate these new ideas. We also look at how certain monads, already used generally to structure functional programs, might be used to structure specifications.

Chapter 5 examines how to reason about expressions, including how to prove properties of, how to transform, and how to refine specifications. A proof system, based on the logic system of the language, is described. In order to support more high level manipulations than those suggested by the axioms of chapter 2, a collection of transformation laws is provided. The refinement operator $\sqsubseteq$ is introduced into the language, with a set of axioms and a collection of refinement laws to support the process of stepwise refinement. Examples are used to illustrate the various concepts introduced, including an example showing the derivation of an imperative-style expression from a simple specification.

The formal semantics of the language is described in chapter 6. This is a denotational semantics using powerdomains to capture non-determinism. In particular, we tackle the problem of giving a meaning to recursive function definitions which might contain non-deterministic terms. The refinement relation is given a meaning based upon the Smyth ordering for powerdomains. We show how the semantic definitions support the axioms and laws provided in chapters 2 and 5. We also consider how a semantics might be give to the informal concept of specification modules introduced in chapter 3.

Chapter 7 concludes the thesis. A summary of the main points is given, along with some discussion of the contributions made. We compare the approach taken to other work in the area of refinement calculi for expressions. Finally, some suggestions for future directions of research are given.

Chapter 2

The Expression Language

In this chapter we aim to define a specification language of expressions. This language is to form one of the components of the refinement calculus.

A programming language is not, in general, useful for specification, since specifications are usually more abstract than programs. This is because a specification should be concerned with expressing *what* is to be achieved, while the program implementing it will dictate *how* the goal will be achieved.

As in the approach taken by Morris and Morgan in the imperative refinement calculus [62, 59, 57, 56], we extend a simple language of expressions with operations and facilities for constructing expressions which are more expressive and less algorithmic in nature. In particular, we add operations for the manipulation of undefined terms, and introduce non-deterministic constructs. Both of these add abstractness to specifications while allowing an implementor to make certain decisions regarding the implementation of a specification.

Various concepts such as undefinedness, non-determinism, equivalence and refinement are explored informally in section 2.1, as well as an overview of the methodology to be employed in the description of the expression language. Section 2.2 gives a formal treatment of undefinedness and non-determinism. The logic of the expression language is set out formally in section 2.3, including an argument for sufficient axiomatisation. The types of expressions are set out in section 2.4 using type rules and axioms. Additional language constructs for specification are described in section 2.5.

Finally, section 2.6 treats the topic of partiality which, in this context, has a different meaning to the usual mathematical interpretation. In fact, as we shall see in chapter 4, partial expressions, and partial functions in particular, are necessary for the construction of specifications in parts. The introduction of partial expressions, however, also means the introduction of possibly miraculous specifications. We show how this may be dealt with syntactically.

2.1 General Overview

In this section we give an informal overview of the various important aspects of the specification language.

2.1.1 Scope of the Language

The language of expressions we use in this thesis has a very broad scope. It is a specification language, with a programming sub-language as well as other non-algorithmic constructs; it contains a logic, both for specification and also forming a reasoning mechanism for the language; it has a module system which is suitable for the construction of large specifications; it has relations for equivalence and refinement, used for comparing expressions; and it is also a calculus, a framework for the rigorous construction of programs from specifications. All of this will become clear in this and the next three chapters.

The basic specification language, which is treated in this chapter, is made up of expressions. Each expression has a unique type, according to the type system described in section 2.4. We do not say exactly which expressions form the programming sub-language. In fact, this will depend on a given problem. For some applications of the calculus, the aim may be to find a deterministic, well-defined specification. For other applications a more low-level expression might be the goal. Indeed, it might be the aim simply to refine an initial specification to a particular form which can be easily tranformed into e.g. an imperative expression. Elements which are certainly not present in the programming language are the non-monotonic elements, such as the equivalence and refinement relations.

2.1.2 Undefinedness

Undefined values necessarily occur in any mathematical language of expressions. Simple examples, with explanations, include

$4/0$	division by zero
$0/0 = 1$	division by zero
$\sqrt{-5}$	when complex numbers are not considered
$hd\langle\rangle$	trying to return the first element of the empty sequence

Although it is clear that such simple expressions do not result in a well-defined value, it is not so clear what should be the outcome of such expressions as

$$(\forall\, n : \mathbb{Z} \mid \bullet n = 0 \lor n/n = 1)$$

$$(\forall\, S : Seq\ T \mid \bullet S = \langle\rangle \lor S = hd\ S \frown tl\ S)$$

where, if the first disjunct is true, the second must be undefined. The first expression states the property that for any integer n, either n is zero, or $n/n = 1$. The second states a property of sequences, that either a sequence is empty, or it is composed of its head and its tail. Undefined expressions are unavoidable, the problem lies in how to handle them.

We make the decision to handle undefinedness explicitly. In order to allow reasoning about such expressions, we augment each type T with a special value '$\bot_T$', usually pronounced "bottom", which represents the undefined value of type T. For example, we say that the result of the evaluation of the expression $4/0$ is $\bot_{\mathbb{Z}}$. We shall drop the subscript in '$\bot_T$' if the type T is clear from the context, or is irrelevant. The undefined expression $\bot_T$ will also be used to represent a "don't care" value, where the specifier doesn't care about the result. This is in keeping with the treatments of [68, 90].

We now need to consider how expressions behave when their constituents are possibly undefined. In most cases it is appropriate to enforce strictness, *i.e.* an operator will yield $\bot$ when applied to $\bot$. So, for example, the expression $(4/0 + 3)$ is undefined, as is the expression $(0/0 = 1)$. As we introduce each operator of the language in turn, we will state whether or not that operator is strict.

However, we do want to have the ability to reason about undefined expressions. For example, it is desirable that the two quantified expressions above should hold. Enforcing strictness of the boolean operators would result in these being undefined. This leads us to new versions of the disjunction and conjunction operators which are symmetric and which satisfy the equivalences:

$$X \wedge \mathit{False} \equiv \mathit{False}$$

$$X \vee \mathit{True} \equiv \mathit{True}$$

for arbitrary (possibly undefined) logical expression X. Formal rules defining these operators will appear in section 2.3. As well as these boolean operators, we will also introduce other non-strict operators, including equivalence $\equiv$ and refinement $\sqsubseteq$. As each such operator is introduced we will describe its behaviour in the presence of undefined terms.

One issue which arises when considering possibly undefined expressions is that of *monotonicity*. An operation *op* is monotonic with respect to an ordering $\sqsubseteq$ if, for any expressions E and F with $E \sqsubseteq F$, we have $E' \sqsubseteq F'$, where E' and F' are the results of applying *op* to E and F respectively. The new versions of conjunction and disjunction retain monotonicity (with respect to the definedness ordering) and are equivalent to their 2-valued counterparts when terms are well-defined. Other non-strict operators may be non-monotonic, including equivalence, essential for reasoning within the language. This operator allows us to assert such equivalences as $(4/0 \equiv \bot_{\mathbb{Z}})$.

In order to distinguish undefined terms in specifications, a non-strict, non-monotonic operator δ will be introduced. For any expression E of any type, $\delta\, E$ is *True* if E is well-defined, and *False* otherwise. Clearly $\neg\delta \bot_T$ holds for any type T. Formal rules for δ will be provided in section 2.2 and as each type of the language is introduced.

2.1.3 Non-Determinism and Partiality

To allow greater flexibility and to increase abstractness in specifications, we introduce the possibility of non-determinism in expressions. In a non-deterministic expression, any one of a number of possible outcomes is acceptable. For example, a familiar non-deterministic specification is to search a sequence for the index of a particular value. If the value occurs more than once in the sequence, it doesn't matter whether the first, the last, or any other occurrence of that value is found.

We admit non-determinism by introducing the choice operator '$[\!]$'. For E and F expressions of the same type T, the expression $E \;[\!]\; F$, also of type T, denotes the non-deterministic choice between the two expressions. Evaluation of $E \;[\!]\; F$ could result in the evaluation of

E or the evaluation of F, but we don't know or care which. Choice enjoys the properties of commutativity, associativity and idempotency.

Non-determinism is often modelled in terms of sets of possible outcomes. For example, the expression 3 has one possible outcome, namely the value 3. The expression $\sqrt{4}$, on the other hand, has two possible outcomes, the elements of the set $\{-2, 2\}$. The set of possible outcomes of an expression $E \mathbin{[]} F$, then, contains the possible outcomes of expression E and the possible outcomes of expression F.

Facilitating non-determinism in the expression language is not a simple matter of just introducing the choice operator $[]$. We also need to consider how other operators of the language behave in the presence of non-deterministic operands. Most operators, such as integer addition, distribute over choice. So, for example, $(3 [] 4) + 7 \equiv 10 [] 11$. A few operators, such as equivalence, refinement and some of the boolean operators, do not distribute. As each operator is formally introduced in sections 2.3 and 2.4, we will state if that operator distributes over choice. If it does not, we must show how that operator is used with choice.

We must also consider the definedness properties of a possibly non-deterministic expression $E \mathbin{[]} F$. In terms of sets of possible outcomes, the undefined integer $\perp_{\mathbb{Z}}$ has $\{\perp_{\mathbb{Z}}\}$ as its set of possible outcomes, while the expression $3 \mathbin{[]} \perp_{\mathbb{Z}}$ is modelled by $\{3, \perp_{\mathbb{Z}}\}$. However, we say that both expressions are undefined. We make the decision that $\delta(E \mathbin{[]} F)$ should hold only when both E and F are well-defined, $\delta\, E \wedge \delta\, F$. This means that $\neg\, \delta(E \mathbin{[]} F)$ holds if either E or F has $\perp$ as a possible outcome. So, $\delta \perp$ is *False*, as is $\delta(3 \mathbin{[]} \perp)$. In contrast, $\delta(3 \mathbin{[]} 4)$ is *True*, as is $\delta\, 3$.

If an expression E yields a single, well-defined outcome, then we say that E is *proper* and we write $\Delta\, E$. For example, $\Delta\, 3$ is *True*, while $\Delta \perp$, $\Delta(3 \mathbin{[]} \perp)$ and $\Delta(3 \mathbin{[]} 4)$ are all *False*. When all expressions are proper, the specification language reduces to the normal, everyday expressions involving familiar types such as integers, booleans, tuples, functions etc. Formal rules for the Δ operator will be given in section 2.2 and also as each type of the language is introduced. Intuitively, it should be clear that if an expression is proper $\Delta\, E$, then it is well-defined $\delta\, E$.

An expression which has a non-empty set of possible outcomes is called *total*. Otherwise, if it has no possible outcomes, not even the undefined outcome, we say that it is *partial*. The partial value, which will be introduced in section 2.6, is written $\top$ (top) and is miraculous. This means that there is no program which implements it. We would like all our specifications to be total, so that we can find (or calculate) programs to implement them. Therefore, we make the decision that our language is to contain only total expressions, although we allow partial sub-expressions. We will show how to accomplish this by restricting the language, in section 2.6.2.

2.1.4 Equivalence and Refinement

We have already mentioned the existence of a non-strict equivalence operator $\equiv$ which does not distribute over choice. It is distinct from the usual equality operator $=$, which is strict and does distribute. Equality will usually be part of the programming language and behaves as expected when its operands are proper. Equivalence, on the other hand, is not part of the algorithmic portion of the language. Its main role in the specification language is for reasoning about expressions. In terms of our model, it compares sets of possible outcomes – if two expressions have the same set of possible outcomes, then they are equivalent.

In sections 2.2, 2.3 and 2.4, the equivalence operator is used to give axioms defining the expression language. These axioms are generally of the form $E \equiv F$, for E and F arbitrary expressions of the same type, which says that the set of possible outcomes of E is exactly the set of possible outcomes of F.

While equivalence $\equiv$ is an equivalence relation over expressions of the language, refinement is an ordering relation. In fact, it is a pre-order. Intuitively, if $E \equiv F$, then a customer asking for E will be happy with F, and vice versa. If $E \sqsubseteq F$, then a customer asking for E will be happy with F, but not the other way round. Again, the refinement operator is not part of the programming language, and is used for reasoning about (refining) specifications.

We have that an undefined expression can be refined by anything, so $\perp_T \sqsubseteq E$ for arbitrary expression E of type T. This supports the decision to allow $\perp$ to be a "don't care" specification, since it can be replaced (refined) by anything. Thus refinement increases definedness.

In terms of possible outcomes, certainly if the set of possible outcomes of E is a superset of the possible outcomes of F, then we must have $E \sqsubseteq F$. So, refinement decreases non-determinism.

Since the set of possible outcomes of the miraculous expression $\top$ is empty, and so a subset of every set, it follows that $\top$ refines every expression, *i.e.* $E \sqsubseteq \top$ for arbitrary expression E. Of course, $\top$ cannot be implemented; if it could, the programmer would have a very simple job.

Formal axioms describing the refinement relation will be given in chapter 5.

2.1.5 The Type System

Every expression of the specification language has a unique type. This is achieved using notation from type theory [6, 7, 20, 76, 87] to introduce various expression formers for each type. The basic types of the language are booleans, integers, characters, products, functions, sets, bags and sequences.

The type theoretic approach to defining the syntax of the language serves two purposes. First, it shows how legal expressions of each type are formed, and so we say that valid expressions of the language are those which are well-typed. For example, $(3 \mathbin{[\!]} 4) + 7$ is well-typed and so a valid expression; while $3 \Rightarrow (4 = 2)$ is not well-typed and so not part of our language.

Secondly, the type theoretic approach also assigns to each expression a unique type. Thus the language has the property of type unicity.

We use the symbols T and T_i, for i any subscript, to represent an arbitrary type. A type judgement, written $a : T$, asserts that value a has type T, and $E : T$ asserts that expression E has type T. A type rule, consisting of zero or more judgements or conditions over a single judgement and separated by a horizontal line, should be interpreted as meaning that, if the conditions above the line are satisfied, then the judgement below the line may be asserted. A condition may be of the form $x : T \vdash E : T'$, where x may occur free in E, meaning that, under the assumption that x has type T, then we can infer that E has type T'.

As well as providing type rules for each expression former, we also give axioms describing the behaviour of such expressions. The expressions introduced here are, essentially, familiar, and their behaviour is well understood and documented, for example in [32, 39]. Our main concern is to describe how the expression may be manipulated in the presence of undefinedness and non-determinacy. Many of the familiar axioms may hold only when constituent terms are proper, or may require some subtle changes to allow for improper terms.

In general, there are not many changes to the standard axioms since most expression constructors are strict and distribute over choice, thereby only making it necessary to describe their behaviour for proper sub-terms. When all terms are proper, the expressions behave exactly as described in any standard treatment.

We will use the identifiers a, b for constant values; x, y for variables; E, F, G for arbitrary expressions; P for Boolean expressions; f, g, h for function expressions; A for sets; B for bags; S for sequences. For any expression E which may contain subexpression x, $E[F/x]$ is the same expression, but with F substituted for each free occurrence of x.

2.1.6 Treatment of the Language

In the next section we begin the formal treatment of the expression language. The language is described using type rules, to give the formal syntax, and axioms to give the behaviour of the various expressions. Since the axioms must necessarily be presented in a linear fashion, some operators are used before their axioms appear. For example, within the axioms for δ and Δ, the implication operator $\Rightarrow$ is used before implication has been introduced. Therefore, we assume that all axioms are asserted at once.

We start, in section 2.2 with an initial description of the operators $[\!]$, δ and Δ, since these are probably new to the reader. The description is initial because more axioms concerning these operators will appear in sections 2.3 and 2.4.

Section 2.3 describes the logical system of the language. This treatment is unusual in that seven logical values are accomodated. Since most of the logical operators are non-strict and do not distribute over choice, some attention must be given to the collection of axioms describing them. We also show that seven distinct values do exist and outline an argument that every logical operator is fully defined with respect to these seven values.

Section 2.4 then describes the remaining types of the language – integers, characters, products, functions, sets, bags and sequences. These types are well-known and understood and so it may be surprising that they are treated here in such detail. The answer is that, while the types may be familiar when all terms are proper (well-defined and deterministic), we need to explicity treat the expressions in the event of improper terms. In many cases it is not so straightforward what is meant by, e.g. applying a function to a non-deterministic argument or adding an undefined value to a set. What we intend to achieve is to provide a set of axioms which describes exactly this form of behaviour, allowing us to reason about and manipulate formally such improper expressions.

For each of the basic types (booleans, integers and characters) we will introduce the proper values. These correspond to the usual values of each type, e.g. *True* and *False* for the booleans. The terms E, F, G, P, Q, R, f, g, h, A, B and S all denote total expressions unless otherwise stated.

Finally, section 2.6 will treat partial expressions.

2.2 Undefinedness and Non-Determinism Formally

We first give the type rules for statements about equivalence and equality of expressions. For any type T, non-strict equivalence and strict equality exist

$$\frac{E : T \quad F : T}{(E \equiv F) : Bool} \qquad \frac{E : T \quad F : T}{(E = F) : Bool}$$

The type *Bool* will be described in the next section.

Now, we introduce undefinedness into the expression language using the type rule:

$$\frac{}{\perp_T : T}$$

This rule states that for any type T, $\perp_T$ has type T. This is the $\perp$-introduction rule.

Non-determinism is introduced into the language using the choice operator:

$$\frac{E : T \quad F : T}{E \mathbin{[\!]} F : T}$$

So, if E and F are both expressions of type T, then the expression $E \mathbin{[\!]} F$ also has type T. This is the $[\!]$-introduction rule.

We introduce the operators δ, which determines the definedness of an expression, and Δ, which determines proper expressions.

$$\frac{E : T}{\delta\, E : Bool} \qquad \frac{E : T}{\Delta\, E : Bool}$$

Now the following axioms describe some of the properties of the above operators. Other axioms will follow in sections 2.3 and 2.4. We assume that E, F and G are arbitrary expressions of an appropriate type and v is any proper value of appropriate type.

Axioms for δ and Δ

$$\Delta\, v$$

$$\Delta\, x$$

$$\neg\delta \perp_T$$

$$\Delta E \Rightarrow \delta E$$
$$\Delta(\delta E)$$
$$\Delta(\Delta E)$$

These axioms state that: all proper values and all variable expressions are proper (and hence well-defined); for every type T, $\perp_T$ is not defined; every expression that is proper is necessarily well-defined; and it is always determined whether an expression is proper or well-defined.

Axioms for $[\!]$

$$E \mathbin{[\!]} E \equiv E$$
$$E \mathbin{[\!]} F \equiv F \mathbin{[\!]} E$$
$$E \mathbin{[\!]} (F \mathbin{[\!]} G) \equiv (E \mathbin{[\!]} F) \mathbin{[\!]} G$$
$$\Delta(E \mathbin{[\!]} F) \equiv \Delta E \wedge \Delta F \wedge (E \equiv F)$$
$$\delta(E \mathbin{[\!]} F) \equiv \delta E \wedge \delta F$$

These axioms state that: choice is idempotent, symmetric and associative; the expression $E \mathbin{[\!]} F$ is proper whenever E and F are proper and equivalent expressions; the expression $E \mathbin{[\!]} F$ is well-defined exactly when both E and F are well-defined.

Equivalence

$$E \equiv E$$
$$(E \equiv F) \equiv (F \equiv E)$$
$$((E \equiv F) \equiv \mathit{True}) \equiv (E \equiv F)$$
$$(E \equiv F) \wedge (F \equiv G) \Rightarrow (E \equiv G)$$
$$(E \equiv F) \Rightarrow (G[E/x] \equiv G[F/x])$$
$$(E \not\equiv F) \equiv \neg(E \equiv F)$$

The first four axioms give the usual properties of equivalence. The fifth axiom is the axiom of Liebniz, which enables substitution of equivalent subterms in an expression G. Clearly, x must have the same type as E and F. The last axiom defines non-equivalence.

Equality We let u, v and w range over proper values of type T.

$$
\begin{aligned}
&\delta(E = F) \equiv \delta\, E \wedge \delta\, F\\
&\Delta(E = F) \equiv \Delta\, E \wedge \Delta\, F\\
&\Delta(E = F) \equiv ((E = F) \equiv (E \equiv F))\\
&(E = F) \equiv (F = E)\\
&(u = u)\\
&(u = v) \wedge (v = w) \Rightarrow (u = w)\\
&(u = v) \equiv (u \equiv v)\\
&((E \mathbin{[\!]} F) = G) \equiv (E = G) \mathbin{[\!]} (F = G)\\
&(E \neq F) \equiv \neg(E = F)
\end{aligned}
$$

The first three axioms state definedness and determinedness properties of equality. The next five axioms state the usual properties of equality for proper values. The eighth axiom shows how equality distributes over choice. The last axiom defines non-equality.

2.3 The Logic

The type of Booleans is represented by *Bool* and has two proper values, *True* and *False*.

$$\frac{}{\mathit{True} : \mathit{Bool}} \qquad \frac{}{\mathit{False} : \mathit{Bool}}$$

From these type rules, and the $\bot$-introduction and $[\!]$-introduction rules, it follows that we can form seven values of type *Bool*: True, False, $\bot_{\mathit{Bool}}$, $\mathit{True} \mathbin{[\!]} \mathit{False}$, $\mathit{True} \mathbin{[\!]} \bot_{\mathit{Bool}}$, $\mathit{False} \mathbin{[\!]} \bot_{\mathit{Bool}}$, $\mathit{True} \mathbin{[\!]} \mathit{False} \mathbin{[\!]} \bot_{\mathit{Bool}}$. We will show, after the presentation of the axioms for logical expressions, that these values are distinct.

The usual disjunction and negation operators exist

$$\frac{P : \mathit{Bool} \quad Q : \mathit{Bool}}{P \vee Q : \mathit{Bool}} \qquad \frac{P : \mathit{Bool}}{\neg P : \mathit{Bool}}$$

The negation operator is strict and distributes over choice. Disjunction is non-strict, but does distribute over choice. The axioms for the propositional logic follow. We assume that the symbols P, Q and R represent arbitrary expressions of type *Bool*.

Disjunction

$$
\begin{aligned}
&P \vee Q \equiv Q \vee P\\
&P \vee (Q \vee R) \equiv (P \vee Q) \vee R\\
&P \vee P \equiv P\\
&P \vee \mathit{True} \equiv \mathit{True}\\
&(P \mathbin{[\!]} Q) \vee R \equiv (P \vee R) \mathbin{[\!]} (Q \vee R)\\
&((P \vee Q) \equiv \mathit{True}) \equiv (P \equiv \mathit{True}) \vee (Q \equiv \mathit{True})
\end{aligned}
$$

The first four axioms give the usual properties of symmetry, associativity, idempotency and *True* as a zero of disjunction. The next axiom treats the behaviour of disjunction with non-deterministic operands. The last axiom shows distributive properties of $\equiv$ over $\vee$.

Negation

$$
\begin{aligned}
&\mathit{False} \equiv \neg \mathit{True}\\
&(\neg P \equiv Q) \equiv (P \equiv \neg Q)\\
&\neg \bot_{Bool} \equiv \bot_{Bool}\\
&\neg (P \mathbin{[\!]} Q) \equiv \neg P \mathbin{[\!]} \neg Q
\end{aligned}
$$

The first two axioms define negation for proper values. The third axioms describes the strictness property of negation. The last axiom treats the behaviour of negation with a non-deterministic operand.

We now define conjunction and implication in terms of disjunction and negation. The definition of conjunction is standard, but the definition of implication is a little unusual.

Conjunction

$$
\begin{aligned}
&P \wedge Q \equiv \neg(\neg P \vee \neg Q)\\
&(P \wedge Q \equiv P) \equiv (P \vee Q \equiv Q)\\
&P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)\\
&((P \wedge Q) \equiv \mathit{True}) \equiv (P \equiv \mathit{True}) \wedge (Q \equiv \mathit{True})
\end{aligned}
$$

The first axiom defines conjunction. The second axiom is the consistency axiom. The last two axioms show how conjunction distributes over disjunction, and a distribution property of equivalence over conjunction.

Implication

$$P \Rightarrow Q \equiv \neg P \vee \neg\Delta\, P \vee Q$$
$$P \Rightarrow (Q \equiv R) \equiv (P \Rightarrow Q \equiv P \Rightarrow R)$$
$$(P \equiv Q) \Rightarrow (P \Rightarrow Q)$$

The first axiom defines implication. This is different from the usual definition, and is based on a definition by Avron given in [1]. When P is proper, this definition reduces to the usual definition of implication. The next two axioms show distribution of implication over equivalence to the right, and the weakening of $\equiv$ to $\Rightarrow$.

Finally, we give an axiom concerning Δ for logical expressions.

Δ-Definition

$$\Delta\, P \equiv ((P \equiv \mathit{True}) \equiv P)$$

This defines Δ for logical values.

2.3.1 Predicate Logic

We now treat quantification in our logical system. Prediate calculus introduces universal and existential quantification over variables in a logical expression. In the current context we need to consider what values the variables can range over; and what happens when the logical expression may be improper.

We make the decision that the quantified variables range only over proper values of the appropriate type. This means that, for example, the expression

$$(\forall\, x : \mathit{Bool} \mid \bullet x = x)$$

is *True*, since x can take only the values *True* and *False*. This decision is further supported by the axiom already given in section 2.2 which stated that any variable identifier x is proper, $\Delta\, x$.

The second consideration concerns the interpretation of quantification with expressions which may be improper. We make the decision that universal quantification is to be treated as generalised conjunction and existential quantification as generalised disjunction. This has the advantage that de Morgan's laws for the quantifiers are retained and that the classical logic holds when all terms are proper.

Other possible treatments might make the quantifiers strict and distribute over non-deterministic expressions. We find that our version is better in that the relationship with the disjunction and conjunction operators is retained, which means that most of the familiar axioms for predicate logic can be asserted in our system.

We now introduce quantified expressions and list the axioms which describe them. The reader will be familiar with most of these axioms. Further theorems are listed in appendix A. The most noticeable difference from classical theory is the **Trading** law for existential quantification. The difference arises because of the new definition of implication. This will be discussed further in the section.

For $\oplus$ one of $\forall, \exists$, we have the type rule for introduction of quantified expressions

$$\frac{x : T \vdash P : Bool \quad x : T \vdash Q : Bool}{(\oplus x : T \mid P \bullet Q) : Bool}$$

We also allow quantified expressions of the form $(\oplus x : T \mid \bullet Q)$ which is simply a shorthand:

$$(\oplus x : T \mid \bullet Q) \equiv (\oplus x : T \mid True \bullet Q)$$

The meaning of quantified expressions is given by the following set of axioms. The symbol $\oplus$ represents either $\forall$ or $\exists$ throughout the axiom in which it occurs. In the following P, Q, Q' and R represent arbitrary expressions of type *Bool* which may contain free variable identifiers x or y; and E is an arbitrary expression of appropriate type.

One-Point Provided x is not free in E and $\Delta\, E$,

$$(\oplus x : T \mid x = E \bullet Q) \equiv Q[E/x]$$

Distribution Provided x is not free in R

$$(\forall x : T \mid P \bullet Q) \wedge (\forall x : T \mid P \bullet Q') \equiv (\forall x : T \mid P \bullet Q \wedge Q')$$
$$R \vee (\forall x : T \mid P \bullet Q) \equiv (\forall x : T \mid P \bullet R \vee Q)$$

Interchange of Dummies Provided y is not free in P and x is not free in Q

$$(\oplus x : T \mid P \bullet (\oplus y : T' \mid Q \bullet R)) \equiv (\oplus y : T' \mid Q \bullet (\oplus x : T \mid P \bullet R))$$

Nesting Provided y is not free in P

$$(\oplus x, y : T, T' \mid P \wedge Q \bullet R) \equiv (\oplus x : T \mid P \bullet (\oplus y : T' \mid Q \bullet R))$$

Dummy Renaming Provided y is not free in P or Q

$$(\oplus x : T \mid P \bullet Q) \equiv (\oplus y : T \mid P[y/x] \bullet Q[y/x])$$

Trading

$$(\forall x : T \mid P \bullet Q) \equiv (\forall x : T \mid \bullet P \Rightarrow Q)$$

Generalised DeMorgan

$$(\exists x : T \mid P \bullet Q) \equiv \neg(\forall x : T \mid P \bullet \neg Q)$$
$$(\exists x : T \mid \bullet Q) \equiv \neg(\forall x : T \mid \bullet \neg Q)$$

Distribution of $\equiv$

$$((\forall x : T \mid P \bullet Q) \equiv \mathit{True}) \equiv (\forall x : T \mid P \bullet Q \equiv \mathit{True})$$
$$(\forall x : T \mid P \bullet Q \equiv Q') \Rightarrow ((\forall x : T \mid P \bullet Q) \equiv (\forall x : T \mid P \bullet Q'))$$
$$((\exists x : T \mid P \bullet Q) \equiv \mathit{True}) \equiv (\exists x : T \mid P \bullet Q \equiv \mathit{True})$$

Further theorems derived from the axioms appear in appendix A. One noticeable theorem is that of **Trading** for existential quantification. From the **Generalised DeMorgan**, **Trading** for universal quantification and the **$\Rightarrow$-Definition**, we get

$$(\exists x : T \mid P \bullet Q) \equiv (\exists x : T \mid \bullet \neg(P \Rightarrow \neg Q))$$

This is equivalent to the usual 2-valued version when all terms are proper.

2.3.2 Theorems

The set of theorems of the specification language is the smallest set of expressions of type *Bool* such that: every axiom is a theorem; a theorem follows from other theorems by an application of one of the inference rules

$$\frac{P \quad P \Rightarrow Q}{Q} \qquad \text{Modus Ponens}$$

$$\frac{P}{(\forall x : T \mid \bullet P)} \qquad \text{Generalisation}$$

A proof of theorem P proceeds as expected, by supplying a sequence of theorems ending with P, where each member is an axiom, a known theorem, or follows from previous elements by an application of an inference rule. Chapter 5 shows how we may reason about expressions of the language using equational reasoning, similar to the style employed by Gries and Schneider in [32].

2.3.3 Sufficient Axiomatisation

The purpose of this section is two-fold. First, we attempt to show that the seven values of type *Bool* are distinct and that this is fixed by the axioms presented. Secondly we will outline an argument that every operator introduced is fully defined with respect to these seven values.

Distinct Values

We have seen that the type *Bool* contains the two proper values, *True* and *False*, the bottom value, $\bot_{Bool}$, and the various combinations of these with the choice operator, giving $True \,[]\, False$, $True \,[]\, \bot_{Bool}$, $False \,[]\, \bot_{Bool}$ and $True \,[]\, False \,[]\, \bot_{Bool}$. In many cases, the distinctness of any two values is shown using the operators Δ and δ.

We first show that *True* and *False* are distinct values, with the following short proof. Notice that we are employing equational reasoning, to be justified in chapter 5.

$$
\begin{array}{ll}
 & False \equiv True \\
\equiv & \quad \text{“}True \text{ an identity for } \wedge \text{ (See appendix A)”}
\end{array}
$$

$$\begin{array}{ll} & (\mathit{False} \equiv \mathit{True}) \wedge \mathit{True} \\ \equiv & \quad \text{"Substitution rule for } \wedge \text{ (See appendix A)"} \\ & (\mathit{False} \equiv \mathit{True}) \wedge \mathit{False} \\ \equiv & \quad \text{"}\mathit{False} \text{ a zero for } \wedge \text{ (See appendix A)"} \\ & \mathit{False} \end{array}$$

and so we conclude that $\mathit{False} \not\equiv \mathit{True}$.

The distinctness of any two values X and Y can be shown by finding a function f such that $f\ X \not\equiv f\ Y$. It follows that $X \not\equiv Y$.

Consider the function Δ. From the axioms we note that $\Delta\ \mathit{True}$ and $\Delta\ \mathit{False}$, but $\neg\Delta\ \bot_{Bool}$. And so we now have three distinct values.

Now consider the value $\mathit{True} \mathbin{[]} \mathit{False}$. From the axiom

$$\Delta(E \mathbin{[]} F) \equiv \Delta\ E \wedge \Delta\ F \wedge (E \equiv F) \tag{2.1}$$

it follows that

$$\Delta(\mathit{True} \mathbin{[]} \mathit{False}) \equiv \mathit{False}$$

since $(\mathit{True} \equiv \mathit{False}) \equiv \mathit{False}$. So $\mathit{True} \mathbin{[]} \mathit{False}$ is distinct from both True and False. It is also distinct from $\bot_{Bool}$ since, from the axiom

$$\delta(E \mathbin{[]} F) \equiv \delta\ E \wedge \delta\ F \tag{2.2}$$

it follows that

$$\delta(\mathit{True} \mathbin{[]} \mathit{False}) \equiv \mathit{True}$$

since both $\delta\ \mathit{True}$ and $\delta\ \mathit{False}$ hold, but $\neg\delta\ \bot_{Bool}$. We now have four distinct values.

Now we consider the three values $X \mathbin{[]} \bot_{Bool}$ for X one of True, False or $\mathit{True} \mathbin{[]} \mathit{False}$. Using the axiom for Δ, (2.1) above, we conclude that

$$\Delta(X \mathbin{[]} \bot_{Bool}) \equiv \mathit{False}$$

and so $X \mathbin{[]} \bot_{Bool}$ is distinct from True and False. Now, using the axiom for δ, (2.2) above, we conclude that

$$\delta(X \mathbin{[]} \bot_{Bool}) \equiv \mathit{False}$$

and so $X \mathbin{[\!]} \bot_{Bool}$ is distinct from $\mathit{True} \mathbin{[\!]} \mathit{False}$.

We now need to distinguish between the undefined values $\bot_{Bool}$, $\mathit{True} \mathbin{[\!]} \bot_{Bool}$, $\mathit{False} \mathbin{[\!]} \bot_{Bool}$ and $\mathit{True} \mathbin{[\!]} \mathit{False} \mathbin{[\!]} \bot_{Bool}$. Unfortunately this is not possible from the axioms as they stand. It would be necessary to introduce a new operator which would distinguish the value $\bot_{Bool}$ from the other undefined values. Although this is possible, it would mean providing a large number of axioms for the new operator to describe its behaviour with each form of expression.

So, we cannot distinguish the undefined values $\bot_{Bool}$, $\mathit{True} \mathbin{[\!]} \bot_{Bool}$, $\mathit{False} \mathbin{[\!]} \bot_{Bool}$ and $\mathit{True} \mathbin{[\!]} \mathit{False} \mathbin{[\!]} \bot_{Bool}$ from each other. Equally, we cannot prove that they are the same value. This means a certain incompleteness in our axioms. It also demonstrates how easily the choice operator could be made demonic by simply asserting that all undefined values are equivalent.

Note that, if we could distinguish $\bot_{Bool}$ from $X \mathbin{[\!]} \bot_{Bool}$, for arbitrary defined X, then it would be a simple matter to show seven distinct values. Using the disjunction operator we would have

$$\begin{aligned} \bot_{Bool} \vee (\mathit{True} \mathbin{[\!]} \bot_{Bool}) &\equiv \mathit{True} \mathbin{[\!]} \bot_{Bool} \\ \bot_{Bool} \vee (\mathit{False} \mathbin{[\!]} \bot_{Bool}) &\equiv \bot_{Bool} \end{aligned}$$

Since we could show that $\bot_{Bool}$ is distinct from $\mathit{True} \mathbin{[\!]} \bot_{Bool}$, we would conclude that $\mathit{True} \mathbin{[\!]} \bot_{Bool}$ is distinct from $\mathit{False} \mathbin{[\!]} \bot_{Bool}$. Now, using the expression template $(X \equiv \neg X)$, we would have that the expression is *True* when X is $\mathit{True} \mathbin{[\!]} \mathit{False} \mathbin{[\!]} \bot_{Bool}$, and *False* when X is either of $\mathit{True} \mathbin{[\!]} \bot_{Bool}$ or $\mathit{False} \mathbin{[\!]} \bot_{Bool}$.

We conclude from all of the above that seven possible logical values exist and that at least four are distinct. Figure 2.1 shows how the operators Δ and δ distinguish logical values.

Sufficient Axioms

The second objective of this section is to outline an argument that every logical operator is fully defined with respect to the axioms. In the above argument we illustrated sufficient axiomatisation for the operators Δ and δ. We have also seen that $\equiv$ is not sufficiently axiomatised since we cannot find a value for e.g.

$$(\mathit{True} \mathbin{[\!]} \bot_{Bool}) \equiv \bot_{Bool}$$

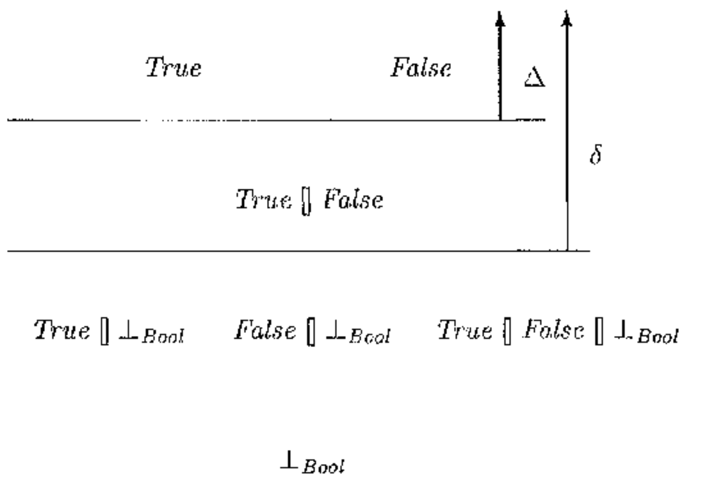

Figure 2.1 Using Δ and δ to distinguish logical values

Other logical operators are $\neg$, $\vee$, $\wedge$ and $\Rightarrow$. Conjunction and Implication are defined in terms of negation, disjunction and Δ, so our task now is to show sufficient axiomatisation $\neg$ and $\vee$.

In the case of negation, the following facts are immediate from the axioms:

$$\begin{aligned}\neg True &\equiv False\\ \neg False &\equiv True\\ \neg\perp_{Bool} &\equiv \perp_{Bool}\end{aligned}$$

For the other four logical values, each of which is of the form $(P \; [] \; Q)$, the axiom concerning **Distribution of $\neg$ over $[]$** is sufficient to yield a value.

In the case of disjunction, there is an axiom describing *True* **as a zero of** $\vee$, a theorem describing *False* **as an identity of** $\vee$ (see appendix A), and an axiom describing the **idempotency of** $\vee$. These laws, together with the axiom for **distribution of** $\vee$ **over** $[]$, are sufficient to yield a value for $P \vee Q$, for logical values P and Q. To illustrate:

$$(True \; [] \perp_{Bool}) \vee (False \; [] \perp_{Bool})$$

$\equiv$ "Distribute $\vee$ over $[]$, Associativity of $[]$"

$$(True \vee False) \; [] \; (True \vee \perp_{Bool}) \; [] \; (\perp_{Bool} \vee False) \; [] \; (\perp_{Bool} \vee \perp_{Bool})$$

$\equiv$ "*True* a zero for $\vee$, Idempotency of $[]$"

$$\begin{array}{ll} & \mathit{True} \;[]\; (\bot_{Bool} \vee \mathit{False}) \;[]\; (\bot_{Bool} \vee \bot_{Bool}) \\ \equiv & \quad \text{“}\mathit{False}\text{ an identity for }\vee\text{”} \\ & \mathit{True} \;[]\; \mathit{False} \;[]\; (\bot_{Bool} \vee \bot_{Bool}) \\ \equiv & \quad \text{“Idempotency of }\vee\text{”} \\ & \mathit{True} \;[]\; \mathit{False} \;[]\; \bot_{Bool} \end{array}$$

Now, conjunction and implication are defined in terms of disjunction, negation and Δ. It follows that, for logical values P and Q, it is possible to find the values of $P \wedge Q$ and $P \Rightarrow Q$ from the definitions of $\wedge$ and $\Rightarrow$, and from the sufficient axiomatisation of $\neg$, $\vee$ and Δ.

2.4 The Type System

In this section we describe the types of the language and how to form expressions of each type.

The basic types are Booleans (as already described), Integers, Characters, as well as other user-defined types to be described in chapter 3. Type constructors include products, functions, sets, bags and sequences. We treat each of these in turn. We also give the axioms governing the behaviour of expressions of each type. It is not claimed that this set of axioms is minimal.

2.4.1 Integers

The type of integers is represented by $\mathbb{Z}$, and we assume the usual proper values.

$$\ldots, -2, -1, 0, 1, 2, \ldots : \mathbb{Z}$$

From the axioms for Δ in section 2.3, each of these is proper, and thus well-defined.

The usual operators over integers are included. For $\oplus$ one of $+, -, *, /, \mathit{mod}, \sqcap$ (min), $\sqcup$ (max), we have the type rule

$$\frac{E, F : \mathbb{Z}}{E \oplus F : \mathbb{Z}}$$

We assume the usual conventions for precedence of operators and the use of bracketing.

The integers are ordered by the '<' operator

$$\frac{E, F : \mathbb{Z}}{E < F : \mathit{Bool}}$$

which has the usual interpretation, and similarly for other comparison operators $>, \leqslant, \geqslant$.

We assume the usual axioms of arithmetic for proper terms, e.g. [32], or a different approach is given in [39]. In particular, we have induction over the natural numbers $\mathbb{N}$, the subset of the integers containing the non-negative elements of $\mathbb{Z}$.

$$(\forall\, n : \mathbb{Z} \mid n \geqslant 0 \bullet (\forall\, i : \mathbb{Z} \mid 0 \leqslant i < n \bullet P) \Rightarrow P[n/i]) \Rightarrow (\forall\, n : \mathbb{Z} \mid n \geqslant 0 \bullet P[n/i])$$

For improper terms, all of the operators over integers are strict and distribute over choice. For $\oplus$ one of $+, -, *, /, \mathit{mod}, \sqcap, \sqcup, <$

$$E \oplus (F \,[\!]\, G) \equiv (E \oplus F) \,[\!]\, (E \oplus G)$$
$$(E \,[\!]\, F) \oplus G \equiv (E \oplus G) \,[\!]\, (F \oplus G)$$
$$\delta(E \oplus F) \Rightarrow (\delta\, E \wedge \delta\, F)$$

The last axiom is an equivalence when $\oplus$ is one of $+, -, *, \sqcap, \sqcup, <$.

Attempts to divide by zero result in undefined terms. For $\oslash$ one of $/, \mathit{mod}$, and with $\Delta\, F$,

$$\delta(E \oslash F) \equiv \delta\, E \wedge \delta\, F \wedge (F \neq 0)$$

These axioms, together with the usual axiomatisation for proper integers, describe the integers of our expression language.

2.4.2 Characters

The type of characters is represented by *Char*. We assume the proper values of the type *Char* to include letters, 'a',...,'z' and 'A',...,'Z', digits, '0',...,'9', punctuation characters and other symbols, e.g. '\$','%','#', as well as the space character, ' ' and the end of line character ' $\hookleftarrow$ '. As with the integers, these also are proper, and hence well-defined.

Apart from comparison of characters, using equality, there are no other operations over characters. The main use for characters is to form strings, which are sequences of characters.

2.4.3 Products

For T_1 and T_2 types, so also is $T_1 \times T_2$ a type.

A member of type $T_1 \times T_2$ consists of the pairing of an element of T_1 and an element of T_2. We have the type rule

$$\frac{E : T_1 \quad F : T_2}{(E, F) : T_1 \times T_2}$$

Components of a pair can be retrieved using the (family of) functions **fst** and **snd**. The type rules are

$$\frac{E : T_1 \times T_2}{\mathbf{fst}\, E : T_1} \qquad \frac{E : T_1 \times T_2}{\mathbf{snd}\, E : T_2}$$

The axioms concerning Δ are that a pair is proper iff its components are proper; and if a pair is proper then retrieving its first or second component will result in a proper expression.

$$\Delta(E, F) \equiv \Delta\, E \wedge \Delta\, F$$
$$\Delta(\mathbf{fst}\, E) \wedge \Delta(\mathbf{snd}\, E) \equiv \Delta\, E$$

Product formation and the functions **fst** and **snd** are strict,

$$\delta(E, F) \equiv \delta\, E \wedge \delta\, F$$
$$\delta(\mathbf{fst}\, E) \equiv \delta\, E$$
$$\delta(\mathbf{snd}\, E) \equiv \delta\, E$$

and distribute over choice

$$(E \mathbin{[]} F, G) \equiv (E, G) \mathbin{[]} (F, G)$$
$$(E, F \mathbin{[]} G) \equiv (E, F) \mathbin{[]} (E, G)$$
$$\mathbf{fst}\, (E \mathbin{[]} F) \equiv \mathbf{fst}\, E \mathbin{[]} \mathbf{fst}\, F$$
$$\mathbf{snd}\, (E \mathbin{[]} F) \equiv \mathbf{snd}\, E \mathbin{[]} \mathbf{snd}\, F$$

This deals with non-deterministic product expressions and expressions with subterms which are not well-defined. For proper expressions we have the usual axioms, where $\Delta\, E$ and $\Delta\, F$

$$\mathbf{fst}(E, F) \equiv E$$

$$\textbf{snd}(E,F) \equiv F$$
$$(E = F) \equiv (\textbf{fst}\, E \equiv \textbf{fst}\, F) \wedge (\textbf{snd}\, E \equiv \textbf{snd}\, F)$$

An example of where these axioms would fail with improper terms is the following:

$$(3,4) \mathbin{[]} (5,6) \not\equiv (3 \mathbin{[]} 5, 4 \mathbin{[]} 6)$$

Both pairs have $3 \mathbin{[]} 5$ as the first component, and $4 \mathbin{[]} 6$ as the second, but the pairs are not equivalent.

In general we allow product types of the form $T_1 \times T_2 \times \ldots \times T_n$, for $n \geqslant 2$. Values of this type look like $(E_1, E_2, \ldots, E_n)$ for $E_i : T_i$. Associated projection functions are written π_i^n of type $T_1 \times T_2 \times \ldots \times T_n \to T_i$, for each $1 \leqslant i \leqslant n$.

2.4.4 Functions

For T_1 and T_2 types, so also is $T_1 \to T_2$ a type.

Elements of a function type are formed using the type rule

$$\frac{x : T_1 \vdash E : T_2}{(\textbf{fun}\ x \in T_1 : E) : T_1 \to T_2}$$

Function application, written using juxtaposition, has the following type rule.

$$\frac{f : T_1 \to T_2 \quad E : T_1}{f\, E : T_2}$$

We take function composition as a basic operation over functions, with the type rule

$$\frac{f : T_2 \to T_3 \quad g : T_1 \to T_2}{f \circ g : T_1 \to T_3}$$

The following axioms hold for Δ

$$\Delta(\textbf{fun}\ x \in T_1 : E)$$
$$\Delta(f \circ g) \equiv \Delta f \wedge \Delta g$$

So, a function abstraction is always proper, *i.e.* well-defined and deterministic, even though its body E might not be. It follows that function abstraction is not strict and does not distribute over choice.

Function application and composition are strict, giving the axioms

$$\delta(f\ E) \Rightarrow \delta\, f \wedge \delta\, E$$
$$\delta(f \circ g) \equiv \delta\, f \wedge \delta\, g$$

and distribute over choice

$$f(E \,[\!]\, F) \equiv f\ E \,[\!]\, f\ F$$
$$(f \,[\!]\, g)E \equiv f\ E \,[\!]\, g\ E$$
$$f \circ (g \,[\!]\, h) \equiv (f \circ g) \,[\!]\, (f \circ h)$$
$$(f \,[\!]\, g) \circ h \equiv (f \circ h) \,[\!]\, (g \circ h)$$

This deals with function expressions which are improper. For proper expressions, with $\Delta\, F$, $\Delta\, f$ and $\Delta\, g$, we have the usual axioms for functions

$$(\textbf{fun}\ x \in T_1 : E)\, F \equiv E[F/x]$$
$$(f \circ g)\, E \equiv f(g\, E)$$
$$(f \equiv g) \equiv (\forall\, x : T_1 \mid \bullet f\ x \equiv g\ x)$$

The last axiom does not hold when either f or g is improper. Examples are

$$\bot_{T_1 \to T_2} \not\equiv (\textbf{fun}\ x \in T_1 : \bot_{T_2})$$

$$(\textbf{fun}\ x \in T : 3) \,[\!]\, (\textbf{fun}\ x \in T : 4) \not\equiv (\textbf{fun}\ x \in T : 3 \,[\!]\, 4)$$

In both cases the left function expression is improper while the right function expression is proper. The functions may also be distinguished when higher-order functions are applied to them.

2.4.5 Sets

For T a type, so also is $\mathbb{P}\, T$ a type.

A set of type $\mathbb{P}\, T$ is an unordered, possibly infinite collection of elements of type T. Each type T is itself a set of type $\mathbb{P}\,T$.

$$\frac{}{T : \mathbb{P}\,T}$$

Sets can also be formed using a predicate.

$$\frac{x : T \vdash P : \mathit{Bool}}{\{x \in T \;:\; P\} : \mathbb{P}\,T}$$

A set of type $\mathbb{P}\,T$ can be obtained by taking the generalised union of a set of sets, of type $\mathbb{P}\,\mathbb{P}\,T$.

$$\frac{A : \mathbb{P}\,\mathbb{P}\,T}{\cup/A : \mathbb{P}\,T}$$

Set membership is denoted by the '$\in$' operator.

$$\frac{E : T \quad A : \mathbb{P}\,T}{E \in A : \mathit{Bool}}$$

Δ for set expressions has the axioms, with T any type

$$\begin{aligned}
&\Delta\, T \\
&\Delta\{x \in T \;:\; P\} \\
&\Delta(\cup/A) \Leftarrow \Delta\, A \\
&\Delta(E \in A) \Leftarrow \Delta\, E \wedge \Delta\, A
\end{aligned}$$

Generalised union $\cup/$ is strict and distributes over choice

$$\begin{aligned}
&\delta(\cup/A) \equiv \delta\, A \\
&\cup/(A_1 \mathbin{[\,]} A_2) \equiv (\cup/A_1) \mathbin{[\,]} (\cup/A_2)
\end{aligned}$$

Membership $\in$ is strict and distributes over choice to its right.

$$\begin{aligned}
&\delta(E \in A) \equiv \delta\, A \wedge \delta\, E \\
&E \in (A_1 \mathbin{[\,]} A_2) \equiv (E \in A_1) \mathbin{[\,]} (E \in A_2)
\end{aligned}$$

This deals with improper sets. For the case where $A, A_1 : \mathbb{P}\,T$, $A' : \mathbb{P}\,\mathbb{P}\,T$, $x : T \vdash P : Bool$ and $E : T$, we have the axioms for proper set expressions $\Delta\,A$, $\Delta\,A_1$, $\Delta\,A'$

$$
\begin{aligned}
&E \in \{x \in T \,:\, P\} \equiv (\mathbf{fun}\; x \in T : P)E \\
&A \in \mathbb{P}\,A_1 \equiv (\forall\, x : T \mid \bullet x \in A \Rightarrow x \in A_1) \\
&E \in \cup/A' \equiv (\exists\, A : \mathbb{P}\,T \mid \bullet E \in A \wedge A \in A') \\
&(A \equiv A') \equiv (\forall\, x : T \mid \bullet x \in A \equiv x \in A')
\end{aligned}
$$

A result of the axioms is that an expression $E \,[\!]\, F$ is in a set A only if both E and F are in A. For example, we have

$$(2 \,[\!]\, 3) \in \{x \in \mathbb{Z} \,:\, x = x\} \equiv \mathit{True}$$

We define the empty set, and the usual operations for sets, where $A, A' : \mathbb{P}\,T$, $a : T$, $x : T \vdash P : Bool$, $x : T \vdash E : T'$, $i, j : \mathbb{Z}$, $f : T \rightarrow T'$, $p : T \rightarrow Bool$,

$$
\begin{array}{lcl}
\emptyset_T & \hat{=} & \{x \in T \,:\, \mathit{False}\} \\
A \cup A' & \hat{=} & \{x \in T \,:\, x \in A \vee x \in A'\} \\
A \cap A' & \hat{=} & \{x \in T \,:\, x \in A \wedge x \in A'\} \\
A \backslash A' & \hat{=} & \{x \in T \,:\, x \in A \wedge x \notin A'\} \\
A \subseteq A' & \hat{=} & A \in \mathbb{P}\,A' \\
A \subset A' & \hat{=} & A \subseteq A' \wedge A \not\equiv A' \\
\{a\} & \hat{=} & \{x \in T \,:\, x \equiv a\} \\
\{x \in A \,:\, P\} & \hat{=} & \{x \in T \,:\, x \in A \wedge P\} \\
\{x \in T \,:\, P : E\} & \hat{=} & \{y \in T' \,:\, (\exists\, x : T \bullet P \wedge E \equiv y)\} \\
\{x \in T \,::\, E\} & \hat{=} & \{x \in T \,:\, \mathit{True} : E\} \\
f * A & \hat{=} & \{x \in A \,::\, f\,x\} \\
p \lhd A & \hat{=} & \{x \in T \,:\, p\,x\} \\
\{i..\} & \hat{=} & \{x \in Z \,:\, i \leqslant x\} \\
\{..j\} & \hat{=} & \{x \in Z \,:\, x \leqslant j\} \\
\{i..j\} & \hat{=} & \{i..\} \cap \{..j\} \\
\mathbb{N} & \hat{=} & \{0..\}
\end{array}
$$

A set A is *finite* if there exists a one-to-one, onto mapping f from $\{0..n-1\}$ to A for some natural number n; in that case its cardinality $\#A$ is defined to be equal to n. Otherwise A is *infinite*. Finite sets may be described by listing the elements of the set, which is just a notational shorthand. For example

$$\{2,4,8\} \quad \equiv \quad \{x \in \mathbb{Z} : x = 2 \vee x = 4 \vee x = 8\}$$

We also introduce reduce over sets, where $\oplus$ which is associative, commutative and idempotent, *i.e.* one of $\cup$, $\cap$, $\sqcup$ or $\sqcap$.

$$\frac{\oplus : T \times T \to T}{\oplus/A : \mathbb{P}\, T \to T}$$

For any such $\oplus$, $\oplus/$ is a function which is strict and distributes over choice in its arguments. So it is sufficient to give axioms for the behaviour of $\oplus/$ when applied to proper arguments. When A_1 and A_2 are proper set expressions, with $\Delta\, E$,

$$\oplus/\{E\} \equiv E$$
$$\oplus/(A_1 \cup A_2) \equiv \oplus/A_1 \cup \oplus/A_2$$

And when $\oplus$ has an identity $1_\oplus$,

$$\oplus/\emptyset \equiv 1_\oplus$$

These axioms fix $\oplus/$ for finite sets only.

We say that reduce distributes to the left over non-deterministic operators.

$$(\oplus \,[\!]\, \oslash)/ \equiv \oplus/ \,\|\, \oslash/$$

It is sometimes useful to consider only finite sets in a specification. For any type T, we use $\mathbb{F}\ T$ to denote the set of finite sets of elements from T, with the expected operators inherited from the type $\mathbb{P}\ T$. In addition, we use $\mathbb{F}_1\ T$ to denote the set of finite, non-empty sets of elements from T. Both $\mathbb{F}\ T$ and $\mathbb{F}_1\ T$ can be defined within the expression language:

$$\mathbb{F}\ T \equiv \{A \in \mathbb{P}\, T\ : (\exists\, n : \mathbb{N} \mid \bullet \#\, S \equiv n)\}$$
$$\mathbb{F}_1\ T \equiv \mathbb{F}\ T \backslash \emptyset_T$$

2.4.6 Bags

If T is a type, then so also is $\mathbb{B}\, T$ a type.

Elements of the type $\mathbb{B}\, T$ are unordered, possibly infinite collections of elements of type T.

A bag is described using a function giving the number of occurrences of each element in the bag. We have the type rule

$$\frac{x : T \vdash E : \mathbb{Z}}{[\![x : T \ggg E]\!] : \mathbb{B}T}$$

For a bag B of elements from T and $E : T$, the expression $B.E$ denotes the number of occurrences of E in B.

$$\frac{B : \mathbb{B}T \quad E : T}{B.E : \mathbb{Z}}$$

A bag expression using bag formation is always proper.

$$\Delta[\![x : T \ggg E]\!]$$

Bag application is strict and distributes over choice to its left.

$$\delta(B.E) \Rightarrow \delta\, B \wedge \delta\, E$$
$$(B_1 \,[\!]\, B_2).E \equiv B_1.E \,[\!]\, B_2.E$$

This accounts for bags which are improper. For proper bags we have the axioms

$$]x : T \ggg E]\!].F \doteq E[F/x] \sqcup 0$$
$$(B \equiv B') \equiv (\forall\, x : T \mid \bullet B.x \equiv B'.x)$$

If E is undefined or non-deterministic at F, then this is reflected in the result of applying the bag to F. So, although the bag $[\![x : T \ggg E]\!]$ and F may be proper, the result of the bag application might not be.

The empty bag, bag membership, bag union, bag subtraction, the subbag relation and filter for bags are defined, for $B, B' : \mathbb{B}\,T$, $a : T$, $p : T \to Bool$, $x : T \vdash P : Bool$,

$$\begin{array}{lcl}
[\![\,]\!]_T & \hat{=} & [\![x : T \ggg 0]\!] \\
a \in B & \hat{=} & (B.a > 0 \equiv True) \\
B \uplus B' & \hat{=} & [\![x : T \ggg B.x + B'.x]\!] \\
B - B' & \hat{=} & [\![x : T \ggg B.x - B'.x]\!] \\
B \subseteq B' & \hat{=} & (\forall x : T \mid \bullet B.x \leqslant B'.x \equiv True) \\
p \triangleleft B & \hat{=} & [\![x : T \ggg \textbf{if } p\,x \textbf{ then } B.x \textbf{ else } 0]\!] \\
[\![x \in B\ :\ P]\!] & \hat{=} & (\textbf{fun } x \in T : P) \triangleleft B
\end{array}$$

Finite bags may be described by listing the elements of the bag, but this is just a shorthand notation. So, for example

$$\begin{array}{lcll}
[\![1, -2, -2, 0]\!] & \equiv & [\![x : \mathbb{Z} & \ggg \textbf{if } x = 1 \textbf{ then } 1 \textbf{ else} \\
 & & & \quad \textbf{if } x = -2 \textbf{ then } 2 \textbf{ else} \\
 & & & \quad \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } 0]\!] \\
[\![\text{‘}a\text{’}]\!] & \hat{=} & [\![x : Char & \ggg \textbf{if } x = \text{‘}a\text{’} \textbf{ then } 1 \textbf{ else } 0]\!]
\end{array}$$

We have map and reduce for bags. With $\oplus$ an associative and commutative operator, we have the type rules

$$\frac{f : T_1 \to T_2}{f* : \mathbb{B}\,T_1 \to \mathbb{B}\,T_2} \qquad \frac{\oplus : T \times T \to T}{\oplus/ : \mathbb{B}\,T \to T}$$

With the axioms

$$\begin{aligned}
&\Delta(f*) \equiv \Delta\, f \\
&\Delta(\oplus/) \equiv \Delta\, \oplus \\
&\delta(f*) \equiv \delta\, f \\
&\delta(\oplus/) \equiv \delta\, \oplus \\
&(f_1 \,[\!]\, f_2)* \equiv f_1 * [\!]\, f_2 * \\
&(\oplus \,[\!]\, \oslash)/ \equiv \oplus/ \,[\!]\, \oslash/
\end{aligned}$$

It is now sufficient to describe the properties of $f*$ and $\oplus/$ over proper bags.

$$\begin{aligned}
&f * [\![\,]\!] \equiv [\![\,]\!] \\
&f * [\![E]\!] \equiv [\![f\ E]\!] \\
&f * (B_1 \uplus B_2) \equiv (f * B_1) \uplus (f * B_2)
\end{aligned}$$

$$\oplus/[\![E]\!] \equiv E$$
$$\oplus/(B_1 \uplus B_2) \equiv (\oplus/B_1) \oplus (\oplus/B_2)$$

When $\oplus$ has an identity, $1_\oplus$

$$\oplus/[\![\,]\!] \equiv 1_\oplus$$

These axioms fix $f*$ and $\oplus/$ for finite bags only.

The set of non-empty bags of elements from type T is denoted by $\mathbb{B}_1 T$.

2.4.7 Sequences

For T a type, then so also is *Seq* T a type.

Elements of *Seq* T are ordered, possibly infinite collections of elements of type T. A sequence is described using a function mapping the natural numbers $\mathbb{N}$ or an initial subset of the natural numbers $\{0..n\}$ to elements of T.

$$\frac{n : \mathbb{Z} \qquad i : \mathbb{Z} \vdash E : T}{\langle i : \{0..n\} \ggg E\rangle : Seq\ T} \qquad \frac{i : \mathbb{Z} \vdash E : T}{\langle i : \mathbb{N} \ggg E\rangle : Seq\ T}$$

The domain of the sequence is the set over which the sequence is defined.

$$\frac{S : Seq\ T}{dom\ S : \mathbb{P}\,\mathbb{Z}}$$

The expression $\#S$, where $S : Seq\ T$, denotes the size (or length) of the sequence S.

$$\frac{S : Seq\ T}{\#S : \mathbb{Z}}$$

For $j : \mathbb{Z}$, the element of type T at position j in S is denoted by $S[j]$.

$$\frac{S : Seq\ T \quad j : \mathbb{Z}}{S[j] : T}$$

Axioms for Δ are

$$\Delta\langle i : \{0..n\} \ggg E\rangle \equiv \Delta\, n \wedge \Delta\, E$$

$$\Delta\langle i : \mathbb{N} \ggcurly E\rangle \equiv \Delta E$$
$$\Delta(dom\ S) \Leftarrow \Delta S$$
$$\Delta(\#\ S) \Leftarrow \Delta S \wedge (dom\ S \not\equiv \mathbb{N})$$
$$\Delta(S[j]) \Leftarrow \Delta S \wedge \Delta j \wedge j \in dom\ S$$

The functions *dom*, $\#$ and sequence application distribute over choice

$$dom(S_1 \mathbin{[\!]} S_2) \equiv dom\ S_1 \mathbin{[\!]} dom\ S_2$$
$$\#(S_1 \mathbin{[\!]} S_2) \equiv \#\ S_1 \mathbin{[\!]} \#\ S_2$$
$$(S_1 \mathbin{[\!]} S_2)[j] \equiv S_1[j] \mathbin{[\!]} S_2[j]$$
$$S[j_1 \mathbin{[\!]} j_2] \equiv S[j_1] \mathbin{[\!]} S[j_2]$$

For proper S and j, i.e. ΔS and Δj

$$\delta(\#\ S) \equiv \delta S \wedge (dom\ S \not\equiv \mathbb{N})$$
$$\delta(S[j]) \rightarrow j \in dom\ S$$

Now, the axioms for proper sequence expressions, with Δj and $I : \mathbb{P}\mathbb{Z}$ such that $I = \mathbb{N}$ or there is some $n : \mathbb{Z}$ such that $I = \{0..n\}$

$$\#\langle i : I \ggcurly E\rangle \equiv \#I$$
$$dom\langle i : I \ggcurly E\rangle \equiv I$$
$$\langle i : I \ggcurly E\rangle[j] \equiv E[j/i] \text{ if } j \in I$$
$$(S = S') \equiv ((dom\ S = dom\ S') \wedge (\forall j : \mathbb{Z} \mid j \in dom\ S \bullet S[j] \equiv S[j']))$$

We define the empty sequence, sequence membership and map for sequences, with $i : \mathbb{Z} \vdash E : T$, $x : T$, $S : Seq\ T$, $f : T \rightarrow T'$,

$$\begin{aligned}
\langle\rangle_T &\mathrel{\hat{=}} \langle i : \{0 \ldots -1\} \ggcurly E\rangle \\
x \in S &\mathrel{\hat{=}} (\exists i : \mathbb{Z} \mid i \in dom\ S \bullet S[i] = x) \\
f * S &\mathrel{\hat{=}} \langle i : dom\ S \ggcurly f(S[i])\rangle
\end{aligned}$$

Concatenation of sequences, $S \frown S'$, for $S, S' : Seq\ T$ is defined as follows

$$\begin{array}{rcll} S \frown S' & \hat{=} & \langle i : \{0, \ldots, (\#S + \#S' + 1)\} \ggg \textbf{if } i < \#S \textbf{ then } S[i] \textbf{ else } S'[i - \#S]\rangle & \\ & & & \text{if S finite} \\ & & S & \text{otherwise} \end{array}$$

Finite sequences, as for bags, may be described by listing the elements of the sequence. Again, this is a notational shorthand. For example

$$\begin{array}{lcll} \langle 1, -2, -2, 0\rangle & \equiv & \langle i : \{0, \ldots, 3\} & \ggg \textbf{ if } i = 0 \textbf{ then } 1 \textbf{ else} \\ & & & \quad\ \textbf{if } i = 1 \textbf{ then } -2 \textbf{ else} \\ & & & \quad\ \textbf{if } i = 2 \textbf{ then } -2 \textbf{ else} \\ & & & \quad\ \textbf{if } i = 3 \textbf{ then } 0 \textbf{ else } n\rangle \\ \langle `a'\rangle & \equiv & \langle i : \{0, \ldots, 0\} & \ggg \textbf{ if } i = 0 \textbf{ then } `a' \textbf{ else } c\rangle \end{array}$$

where n may be any integer, and c is any character.

We introduce reduce for sequences. With $\oplus$ an associative operator, we have the type rule

$$\frac{\oplus : T \times T \to T}{\oplus/ : SeqT \to T}$$

With the axioms

$$\Delta(\oplus/) = \Delta\ \oplus$$

$$\delta(\oplus/) \equiv \delta\ \oplus$$

$$(\oplus \,[\!]\, \oslash)/ \equiv \oplus/ \,[\!]\, \oslash/$$

It is now sufficient to describe the properties of $\oplus/$ over proper sequences.

$$\oplus/\langle E\rangle \equiv E$$

$$\oplus/(S_1 \frown S_2) \equiv (\oplus/S_1) \oplus (\oplus/S_2)$$

When $\oplus$ has an identity, $1_\oplus$

$$\oplus/\langle\rangle \equiv 1_\oplus$$

Now, filter for sequences and sequence comprehensions are defined, with $S : SeqT$, $p : T \to$

$Bool$, $x : T \vdash E : T'$, $x : T \vdash P : Bool$,

$$\begin{aligned} p \lhd S &\mathrel{\hat{=}} {}^\frown/\langle i : dom\ S \ggg \textbf{if}\ p\ S[i]\ \textbf{then}\ \langle S[i]\rangle\ \textbf{else}\ \langle\rangle\rangle \\ \langle x \in S : P : E\rangle &\mathrel{\hat{=}} (\textbf{fun}\ x \in T : E) * ((\textbf{fun}\ x \in T : P) \lhd S) \end{aligned}$$

We identify the set of strings, $String$, with sequences of characters.

$$String \mathrel{\hat{=}} Seq\ Char$$

Instead of writing strings using the sequence notation, as in

⟨'T','h','i','s',' ','i','s',' ','a',' ','s','t','r','i','n','g','.'⟩

they can be written using double quotes, as in "This is a string.".

We define the set of injective sequences of elements from a type T. $ISeq\,T$ contains sequences of elements in which any a in T occurs at most once.

$$ISeq\ T \mathrel{\hat{=}} \{S \in Seq\ T : (\forall\, i, j : \mathbb{Z} \mid 0 \leqslant i, j < \#S \bullet S[i] = S[j] \Rightarrow i = j)\}$$

The set of non-empty sequences of elements from type T is denoted by $Seq_1\,T$.

2.4.8 Partial Mappings

We could also include the set of partial mappings from a domain type T_1 to a range type T_2, written $T_1 \nrightarrow T_2$. This uses the Z notation and operations for partial functions, as given in [75, 44], and can be defined in terms of sets of pairs of type $T_1 \times T_2$.

For example, we can define the set of partial mappings $T_1 \nrightarrow T_2$ as

$$\begin{aligned} T_1 \nrightarrow T_2 \ \equiv\ & \{f \in \mathbb{P}(T_1 \times T_2) : \\ & (\forall\, x \in T_1 \mid \bullet(\forall\, y_1, y_2 \in T_2 \mid (x, y_1) \in f \wedge (x, y_2) \in f \bullet y_1 = y_2))\} \end{aligned}$$

For f a partial mapping in $T_1 \nrightarrow T_2$, instead of writing elements of f using product notation (x, y), we may use the standard maplet notation $x \mapsto y$. Override and application can be defined as in [44]. The set of total mappings $T_1 \nrightarrow_t T_2$ can be defined as

$$T_1 \nrightarrow_t T_2 \equiv \{f \in T_1 \nrightarrow T_2 : \{x \in T_1 : (\exists\, y \in T_2 \mid \bullet x \mapsto y \in f)\} = T_1\}$$

Since the notation for partial and total mappings is defined in terms of products, which have already been treated for undefinedness and partiality, there is no need to give an axiomatisation for them. They may be considered as useful syntactic definitions only.

2.4.9 Simple Types

We define the collection of *simple types* to be the smallest such which includes

- the types $Bool$, $\mathbb{Z}$ and $Char$;
- the types $T_1 \times T_2$, $\mathbb{P}\, T$, $\mathbb{B}\, T$ and $Seq\ T$ for T, T_1 and T_2 simple.

2.5 Language Constructs

In this section we describe the expression formers of the language. Again we use type theory to introduce the new concepts.

2.5.1 Conditional Expressions

We introduce the constructor for conditional expressions, **if** P **then** E **else** F. We have the type rule

$$\frac{P : Bool \qquad E, F : T}{\textbf{if } P \textbf{ then } E \textbf{ else } F : T}$$

In fact, we take the view that there is an **if** constructor for each type T, and that these form a family of such constructors. The conditional expression is strict in its first argument. Axioms for conditional expressions are

$$\begin{array}{l}
\textbf{if } True \textbf{ then } E \textbf{ else } F \equiv E \\
\textbf{if } False \textbf{ then } E \textbf{ else } F \equiv F \\
\neg \Delta\, P \Rightarrow (\textbf{if } P \textbf{ then } E \textbf{ else } F \equiv \bot)
\end{array}$$

The last axiom may seem a little odd, particularly for the case where P is $True \mathbin{[\!]} False$. This derives from the fact that a conditional expression is considered to be part of the programming language, rather than a specification constructor. As such, its first argument is expected to be deterministic. If it is not deterministic, then the expression is treated as undefined. We note that the **if** constructor described by these axioms is monotonic in each argument.

2.5.2 Local Definitions

We introduce the **let** expression for local definitions. If $E : T_1$ and $x : T_1 \vdash F : T_2$, then the expression

$$\textbf{let } x = E \textbf{ in } F$$

has type T_2 and is defined by

$$\textbf{let } x = E \textbf{ in } F \quad \hat{=} \quad (\textbf{fun } x \in T_1 : F)\, E$$

There is a **let** constructor for each pair of types (T_1, T_2).

More generally, several local definitions can be introduced in parallel using a single **let** construct, successive definitions separated by '$\|$'. If $E_i : T_i$ and we have the judgement $x_1 : T_1, \ldots, x_n : T_n \vdash F : T'$, then the expression

$$\textbf{let } x_1 = E_1 \| \ldots \| x_n = E_n \textbf{ in } F$$

has type T' and may be defined as

$$\textbf{let } x_1 = E_1 \| \ldots \| x_n = E_n \textbf{ in } F \quad \hat{=} \quad (\textbf{fun } x_1 \in T_1, \ldots, x_n \in T_n : F)\, (E_1, \ldots, E_n)$$

Clearly the order of writing the definitions of the x_i's in the **let** expression makes no difference to the expression.

To avoid having expressions with lots of nested **let** definitions we introduce a syntactically nicer form

$$\textbf{let } x_1 = E_1 \,\&\, \ldots \,\&\, x_n = E_n \textbf{ in } F$$

where x_i may occur in E_j provided $i < j$. This form is equivalent to

$$\textbf{let } x_1 = E_1 \textbf{ in } (\textbf{let } x_2 = E_2 \textbf{ in } (\ldots (\textbf{let } x_n = E_n \textbf{ in } F) \ldots))$$

which, in turn, denotes

$$(\textbf{fun } x_1 \in T_1 : (\textbf{fun } x_2 \in T_2 : \ldots (\textbf{fun } x_n \in T_n : F) E_n \ldots) E_2)) E_1$$

2.5.3 Recursive Functions

Recursive function definitions are included in the specification language using a **let** expression where the free variable f may occur free in its defining expression $E[f]$.

$$\textbf{let } f = (\textbf{fun } x \in T : E[f]) \textbf{ in } F[f]$$

The notation $E[f]$ means that f is a free variable of expression E. We limit recursive definitions to function types only. For example, we could have the expression

$$\textbf{let } \mathit{fac} = (\textbf{fun } x \in \mathbb{Z} : \textbf{if } x \leqslant 1 \textbf{ then } 1 \textbf{ else } x * \mathit{fac}(x-1)) \textbf{ in } \mathit{fac}\, 3$$

which we expect should result in the value 6.

The behaviour of such a recursive definition may be described by unfolding its definition, so we assert the axiom

$$\textbf{let } f = E[f] \textbf{ in } F[f] \quad \doteq \quad F[E[(\textbf{let } f = E[f] \textbf{ in } f)]]$$

Applying this a number of times to the above example gives the desired result. This axiom states that f is a fixpoint of some functional. In fact, as will be seen in the semantics presented in chapter 6, f is a least fixpoint of the functional, with respect to a definedness ordering.

2.5.4 Specification Expressions

We introduce a new operation on sets called *generalised choice* and write this $[]/$. Clearly, it is based on using the choice operator $[]$ with reduce for sets. If S is a non-empty, possibly infinite set of type $\mathbb{P}\, T$, then the expression $[]/S$ has type T and can be interpreted as 'choose any element of S'. For example

$$[]/\{3,4,5,6\} \equiv 3 \,[]\, 4 \,[]\, 5 \,[]\, 6$$

The type rule is

$$\frac{S : \mathbb{P}\, T}{[]/S : T}$$

Expressions of the form $[]/S$ are termed *specification expressions* [90].

We have the following axioms for $[]/$

$$\begin{aligned}
&\Delta([]/S) \equiv (\# S \equiv 1)\\
&\delta([]/S) \equiv \delta\, S\\
&[]/(S_1 \; [] \; S_2) \equiv ([]/S_1) \; [] \; ([]/S_2)
\end{aligned}$$

and for $\Delta\, S$, $\Delta\, S_1$ and $\Delta\, S_2$

$$\begin{aligned}
&[]/\{v\} \equiv v\\
&[]/(S_1 \cup S_2) \equiv ([]/S_1) \; [] \; ([]/S_2)\\
&([]/S_1 \equiv []/S_2) \equiv (S_1 \equiv S_2)
\end{aligned}$$

The expressive power of the generalised choice operator is realised when it is used with set comprehensions. We have the axiom

$$(\exists\, x \in T \bullet P) \Rightarrow P([]/\{x \in T \,:\, P\, x\})$$

An initial specification can be given by defining the properties required of a solution using a predicate P say, forming the set of all elements which satisfy that property $\{x \in T \,:\, P\, x\}$, and then using $[]/$ to choose any one of those elements. Provided it can be proven that there is a solution, *i.e.* $(\exists\, x \in T \bullet P)$, then the set $\{x \in T \,:\, P\, x\}$ is non-empty, and the specification is given as

$$[]/\{x \in T \,:\, P\, x\}$$

which may, of course, be a non-deterministic expression. For example

$[]/\{x \in \mathbb{Z} \,:\, 0 \leqslant x : 2 * x\}$	Any even natural
$[]/\{s \in \mathbb{P}\mathbb{Z} \,:\, \#s = 10\}$	Any integer set with exactly 10 elements

More interesting examples using this form of specification can be found in the following chapter.

2.5.5 Assumptions and Partially Defined Functions

We introduce a new expression constructor, '$\succ\!\!-$', with the type rule

$$\frac{P : \mathit{Bool} \quad E : T}{P \succ\!\!- E : T}$$

The boolean expression P is called the assumption. The intuitive meaning of $P \succ\!\!- E$ is such that, if $P \equiv \mathit{True}$ then $P \succ\!\!- E \equiv E$, and otherwise $P \succ\!\!- E \equiv \bot_T$.

The assumption constructor $\succ\!\!-$ is strict in its left argument and distributes over choice to the right. Axioms for assumptions are, with $E : T$,

$$\begin{aligned}
&P \succ\!\!- E \mathbin{[\!]} F \equiv (P \succ\!\!- E) \mathbin{[\!]} (P \succ\!\!- F)\\
&\mathit{True} \succ\!\!- E \equiv E\\
&\mathit{False} \succ\!\!- E \equiv \bot_T\\
&\neg\Delta\, P \Rightarrow (P \succ\!\!- E \equiv \bot_T)
\end{aligned}$$

This last axiom may appear unusual for the case when P is $\mathit{True} \mathbin{[\!]} \mathit{False}$, although we notice that $\succ\!\!-$ is monotonic in both arguments. The above axiomatisation is useful for case based reasoning about expressions of the form $P \succ\!\!- E$. There are three cases to consider, $P \equiv \mathit{True}$, $P \equiv \mathit{False}$ and $\neg\Delta\, P$.

We sometimes want to specify a function which will only ever be applied to elements of a restricted set, and we don't care what happens if it is applied to something outside that set. For example, the integer square root function should only ever be applied to the natural numbers, $\mathbb{N}$. Having assumptions gives us an easy way to write such functions which are only partially defined. For A a set of type $\mathbb{P}\, T$, we define

$$(\textbf{fun}\ x \in A : E) \quad \hat{=} \quad (\textbf{fun}\ x \in T : (x \in A) \succ\!\!- E)$$

Now the function $(\textbf{fun}\ x \in A : E)$ acts like the function $(\textbf{fun}\ x \in T : E)$ whenever it is applied to something in A. For any $a \notin A$, the result of the application will be equivalent to $\bot$.

For example, the square root function can be specified as

$$\mathrm{Sqrt} \quad \hat{=} \quad (\textbf{fun}\ n \in \mathbb{N} : [\!]/\{x \in \mathbb{Z} : x^2 \leqslant n < (x+1)^2\})$$

It can be proven that the set comprehension will not be empty, and so $[\!]/$ will pick one of the elements which satisfy the predicate used to describe the set.

2.5.6 Inverse Functions

For any function $f : T_1 \to T_2$, we define the inverse of f, called f^{-1} as follows. For externally nondeterministic functions, we assert that inverse distributes over choice,

$$(f \mathbin{[\!]} g)^{-1} \equiv f^{-1} \mathbin{[\!]} g^{-1}$$

For f proper we define

$$f^{-1} \;\hat{=}\; (\textbf{fun } y \in T_2 : (y \in f * T_1) \succ\!\!- \mathbin{[\!]}/\{x \in T_1 : f\,x \equiv y\})$$

So, for any $y \in T_2$, $f^{-1}\,y$ is defined if there is some $x \in T_1$, not necessarily unique, with $f\,x \equiv y$, i.e. y is in the range of f. If there is more than one such x, in the case where f is internally nondeterministic, the result of $f^{-1}\,y$ is a choice between them.

2.5.7 Generic Functions

A generic function definition actually defines a family of functions. The notation we use is *function_name*$[T]$, which represents a family of functions, one for each type T. In actual use, the index T can usually be inferred from the context, and so the index will be dropped.

A generic function is defined using a type parameter, as in

$$\textit{function\_name}[T] \;\hat{=}\; f_T$$

where f is a function expression containing the type index T. For example, we could define a generic search function as follows

$$\textit{search}[T] \;\hat{=}\; (\textbf{fun } x \in T, A \in \textit{Seq } T : (\exists\, i : \mathbb{N} \mid \bullet A[i] = x) \succ\!\!- \mathbin{[\!]}/\{i \in \mathbb{N} : A[i] = x\})$$

This actually specifies a family of search functions, one for each possible type T.

More generic functions will be described in chapter 3.

A polymorphic function is one whose actual parameters can have more than one type. Literature in the area of type theory, e.g. [20, 21, 29, 76], identifies at least two forms of polymorphism: parametric polymorphism, where a function works uniformly on a range of types; and ad-hoc polymorphism, where a function works on several different types and may behave differently for each type.

Our generic functions, defined using a type parameter, are similar to parameterised templates. They must be instantiated with actual types before use. But each instantiated function behaves in the same way, independent of the type instantiation. Thus we claim that our generic functions provide a weak form of parametric polymorphism.

In most cases, this weak form of polymorphism is sufficient. What is missing is the possibility of having higher-order functions that accept polymorphic functions as arguments. For example, although we can define

$$id[T] \mathrel{\hat{=}} (\mathbf{fun}\ x \in T : x)$$

which, for a given type T, has type $T \to T$; we are not allowed to define the function

$$illegal[T] \mathrel{\hat{=}} (\mathbf{fun}\ f \in T \to T : (f\ 3, f\ True))$$

because it cannot be typed for a given T.

The reason we are using the weaker form of polymorphism for our expression language is because of the simplicity of the type system. In order to allow higher-order functions accepting polymorphic functions as arguments, we would require a second-order type system. Although we have not fully investigated such an approach, Reynolds [76] suggests that type deduction in such a system might be problematic, and that the language could present semantic difficulties. On the other hand, he also presents some examples illustrating the possible benefits arising from the more expressive langauge.

2.6 Partiality

Experience with the Z specification language has shown that it is a useful feature to allow a specification to be constructed in parts. Such partiality is distinct from undefinedness as described in section 2.1. Partial specifications mean that a single aspect of the problem can be focussed upon in isolation, and the complete specification obtained by assembling the parts.

We obtain partiality by introducing an identity for choice, which we give the fictitious value $\top$, pronounced "top". So, we have that $\top \mathbin{[]} E \equiv E$ for any expression E. We assert that $\top$ is distinct from $\bot$, and so it must be well-defined $\delta\,\top$. But $\top$ is not a proper value, so we assert $\neg\Delta\,\top$.

Now that $[]$ has an identity, it follows that the generalised choice operator $[]/$ is also defined for empty sets. From the properties of reduce we must have that $[]/\emptyset \equiv \top$.

As defined in section 2.1.3, we say that an expression E is *total* if E cannot evaluate to $\top$. Otherwise E is *partial*. All the expressions we have seen so far have been total.

2.6.1 Potentially Partial Expressions

We now introduce the concept of a guarded expression. We have the type rule

$$\frac{P : \mathit{Bool} \quad E : T}{P \to E : T}$$

where the boolean expression P is called the guard. The intuitive meaning of a guarded expression $P \to E$ is such that: if P is *True* then $P \to E \equiv E$; if P is *False* then $P \to E \equiv \top$; and otherwise $P \to E \equiv \bot$.

The expression constructor $\to$ is strict in its left argument and distributes over choice to the right. The axioms are, with $E : T$,

$$\mathit{True} \to E \equiv E$$
$$\mathit{False} \to E \equiv \top$$
$$\neg\Delta\, P \Rightarrow (P \to E \equiv \bot_T)$$

As for assumptions, these axioms have been formed to facilitate case-based reasoning. To prove something about an expression $P \to E$ it is convenient to consider three cases, $P \equiv \mathit{True}$, $P \equiv \mathit{False}$ and $\neg\Delta\, P$.

Since an expression of the form $P \to E$ may 'evaluate' to $\top$, we say that guarded expressions are *potentially partial*. This means that expressions of the form $[]/S$ are also potentially partial, in the case where S might be empty. We note the following law, for any set S with $\Delta\, S$,

$$[]/S \equiv (S \not\equiv \emptyset) \to []/S$$

An *alternation expression* is of the form $P_1 \to E_1\ [] \ldots []\ P_n \to E_n$. Any guard P_i which evaluates to *False* has the result that the guarded expression $P_i \to E_i$ effectively disappears from the alternation. If all the guards are proper, then the alternation is such that some expression E_j for which the corresponding guard P_j evaluates to *True* will be chosen and evaluated. For example, the alternation

$$x \geqslant 0 \to \text{`+'}\ []\ x \leqslant 0 \to \text{`}-\text{'}$$

will evaluate to '+' if the integer x is positive, to '−' if x is negative, and to either '+' or '−' if x is 0. An alternation expression is potentially partial, since all guards may be *False*.

The conditional expression, introduced in section 2.3, is a special form of the alternation expression. We have

$$\textbf{if } P \textbf{ then } E \textbf{ else } F \quad \equiv \quad P \rightarrow E \,[]\, \neg P \rightarrow F$$

It should be clear that a conditional expression is total, provided E and F are total.

Partial expressions, on their own, are not useful as specifications, since no program can satisfy such a specification. The intention in introducing potentially partial expressions is that they may be combined, using choice, to form total specifications. In order to control occurrences of potentially partial expressions in specifications, we restrict the syntax of the language, as described in the next section.

2.6.2 Managing Miracles

Although the introduction of $\top$ brings great expressive power to the language and, as we will see in chapter 5, greatly facilitates the piecewise refinement of expressions, it is nonetheless a very dangerous expression.

No program can satisfy the specification $\top$. It is the miraculous specification which solves all our problems, but cannot be implemented. We will see, in chapter 5, that it is the most refined specification, since it refines every expression. Therefore, we have a problem. Given an initial specification expression E, there is nothing to stop the developer from over-refining E, perhaps in a sequence of steps, to the miraculous specification, thereby resulting in something which is unimplementable. Although this is not desirable on the part of the developer, it is possible that he may inadvertantly introduce partial, and therefore problematic, subexpressions during the refinement.

We intend to control occurrences of potentially partial expressions so that every specification of the language, whether an initial specification or one calculated by refinement from a previous specification, is total. We find that it is possible to impose simple syntactic restrictions which will ensure that every specification is a total expression.

Recognising Potentially Partial Expressions

From the language description in this chapter, and from earlier comments in this section, we see that potentially partial expressions can occur in exactly 2 possible ways:

- from a generalised choice, $[]/S$
- from a guarded expression, $P \to E$

In the first case, the expression $[]/S$ is partial when S is the empty set; in the second case, the expression $P \to E$ is partial when P is *False*. There are no other constructs where partiality might be created. All other language constructs are total. So, it is only in the cases of generalised choice and guarding where we need to be concerned about the possible introduction of the miraculous expression $\top$. Both of these cases are recognisable syntactically.

Potentially partial expressions are defined as the smallest subset of expressions satisfying

- Expressions of the form $[]/S$ are potentially partial.
- Expressions of the form $P \to E$ are potentially partial.
- If E is potentially partial then so is $E \; [] \; F$, for arbitrary F.

Restricting the Syntax

We don't want to eliminate potentially partial expressions completely. We've seen that guarded expressions are very useful when used with choice to form alternation expressions. Generalised choice expressions are also extremely useful specification tools. We do, however, intend to ensure that potentially partial expressions are never used directly with operators (other than choice), constructors or function application. None of these can create partiality, but they would propogate it.

What is required is a way of 'totalising' potentially partial expressions, *i.e.* transform them into total expressions, so that they can be used freely in specifications. We introduce a new operator, biased choice $\overleftarrow{[]}$, which always chooses its left operand if possible. The type rule is

$$\frac{E, F : T}{E \; \overleftarrow{[]} \; F : T}$$

Intuitively, $E \; \overleftarrow{[]} \; F$ is equivalent to E if E is total, otherwise $E \; \overleftarrow{[]} \; F$ is equivalent to F. Biased choice is associative and idempotent, but clearly not symmetric. It is strict in its left argument and distributes over choice to the right. We have the axioms

$$(E \equiv \top) \Rightarrow (E \overset{\leftarrow}{[]} F \equiv F)$$
$$(E \not\equiv \top) \Rightarrow (E \overset{\leftarrow}{[]} F \equiv E)$$

Most importantly, the expression $E \overset{\leftarrow}{[]} F$ is guaranteed to be total if F is. This means that given a potentially partial expression, such as $P \to E$, it can be 'totalised' by combining it with a total 'alternative' F, giving an expresson of the form $P \to E \overset{\leftarrow}{[]} F$.

We now give the extra restrictions placed on expressions of the specification language. The use of potentially partial expressions is such that they may only be:

- operands of $[]$ – thus forming a new potentially partial expression;
- the left operand of $\overset{\leftarrow}{[]}$ – thus forming a total expression;
- operands of $\equiv$, $\sqsubseteq$, Δ and δ – thus forming total expressions.

Biased Choice and Conditionals

The specification form $E \overset{\leftarrow}{[]} \perp$ is used frequently in specifications. Intuitively it means that if E is total then choose an outcome of E and otherwise we don't care about the value of the expression. We define the shorthand

$$\textbf{if } E \textbf{ fi} \quad \equiv \quad E \overset{\leftarrow}{[]} \perp$$

which allows us to write nicer alternation expressions, for example

$$(\textbf{fun } x \in \mathbb{Z} : \textbf{if } x \geqslant 0 \to \text{'+'} \; [] \; x \leqslant 0 \to \text{'−'} \textbf{ fi})$$

instead of

$$(\textbf{fun } x \in \mathbb{Z} : (x \geqslant 0 \to \text{'+'} \; [] \; x \leqslant 0 \to \text{'−'}) \overset{\leftarrow}{[]} \perp)$$

There is a connection between expressions based on biased choice and the conditional expression which we met at the end of section 2.5.1. We have that

$$\textbf{if } P \textbf{ then } E \textbf{ else } \perp \equiv (P \to E) \overset{\leftarrow}{[]} \perp$$

Further, if ΔP

$$\textbf{if } P \textbf{ then } E \textbf{ else } F \equiv (P \to E) \overset{\leftarrow}{[\!]} F$$

A Relaxation of the Rules

There is one case where we would like to relax the special syntax rules given above. In general, we are not permitted to write $P \rightarrowtail [\!]/S$, since $[\!]/S$ is potentially partial and so cannot be an operand of the assumption operator $\rightarrowtail$. However, if P guarantees that S is not empty, and ΔS, then we allow such expressions. In particular, we allow

$$\begin{array}{rcl} S \neq \emptyset & \rightarrowtail & [\!]/S \\ (\exists x \in T \bullet P\,x) & \rightarrowtail & [\!]/\{x \in T : P\,x\} \end{array}$$

We claim that such a form is very useful for specifications, and we have in fact already used this style of specification in the definition of function inverse in section 2.5.6.

The justification for this relaxation is based on the theorem, which will be given in chapter 5,

$$(S \neq \emptyset \rightarrowtail [\!]/S) = [\!]/S \overset{\leftarrow}{[\!]} \perp$$

when ΔS. Since $\perp$ is total, the expression on the right is total, and so the expression on the left must also be total.

2.7 Conclusions

In this chapter we have defined a specification language of expressions, based on ordinary mathematical expressions, but including facilities for the formation and manipulation of expressions which are undefined or nondeterministic.

The language has been described using type rules and axioms. The type rules ensure that every expression has a unique type. The axioms describe how the various constructs behave with non-proper terms, which is usually based on strictness and distribution over choice. Axioms are also provided for proper terms.

The syntax of specification modules will be described in the next chapter, where we give a number of small example specifications, illustrating the use of the various concepts of the expression language.

A proof system governing the manipulation and refinement of expressions using these axioms will be discussed in chapter 5.

Section 2.6 introduced the concept of a partial expression. Such potentially partial expressions cannot be implemented and so can be dangerous in a specification. However, they are useful in the process of constructing specifications by parts. This method of constructing specifications will be further developed in chapter 4 when we describe how the language can be used for large specifications. On a bigger scale, considering specifications in parts is vital.

Luckily, potentially partial expressions may be recognised syntactically. They may arise in only a limited number of ways. This means that it is possible to control their use and, by always totalising such expressions, to ensure that complete specifications are always total.

Chapter 3

Making Specifications

In this chapter we show how to use the specification language of chapter 2 to make specifications.

First we define some generic functions which, though not part of the language definition itself, are used frequently in specifications. Rather than replicating their definitions at each point of use, they are defined in section 3.1, with the understanding that the function names are replaced by their definitions wherever the names occur. The act of replacing a name by its definition is sometimes referred to as *unfolding the definition*.

The concept of a specification module is described in section 3.2. Although each expression of the language is a specification, it is generally the case that a specification will require a number of expressions, together with user defined types, collected together to form a module. We describe methods by which user defined types, e.g. *Book*, *Person*, *Colour etc.* , can be introduced into a specification, and give an informal syntax for specification modules.

Finally, to illustrate the expression language and how it is used in specification, we give four substantial examples. A larger example illustrating the problem of structuring specifications will be developed in chapter 4.

3.1 Useful Functions

In this section we define some generic functions which are useful in specifications.

Ran

The range of a function is the set of its possible outcomes. The function $ran[T_1, T_2]$ is applied to a function f of type $T_1 \to T_2$ and returns its range, formally

$$ran[T_1, T_2] \mathrel{\hat{=}} (\textbf{fun } f \in T_1 \to T_2 : f * T_1)$$

The range of a sequence is simply the set of values that appear in it. In this case, the function $ran[T]$ is applied to a sequence S of type $Seq\ T$ and returns its range, formally

$$ran[T] \mathrel{\hat{=}} (\textbf{fun } S \in Seq\ T : \{i \in \{0 \ldots \#S - 1\} :: S[i]\})$$

Conversions to Sets

It is sometimes necessary to convert a bag or a sequence to a set. For a bag, this means losing frequency information, and for a sequence, both duplication and order are lost. The function $BagToSet[T]$ converts a bag B of type $\mathbb{B}\,T$ to a set of type $\mathbb{P}\,T$, and is defined by

$$BagToSet[T] \mathrel{\hat{=}} (\textbf{fun } B \in \mathbb{B}\,T : \{x \in T \ : \ B.x > 0\})$$

Similarly, the function $SeqToSet[T]$ converts a sequence S of type $Seq\ T$ to a set of type $\mathbb{P}\,T$, defined by

$$SeqToSet[T] \mathrel{\hat{=}} ran[T]$$

This is the same as just using $ran[T]$ but, in a specification, it may be desirable to make explicit the intention of converting a sequence to a set.

Maximising/Minimising Functions

A very useful generic function is $minWRT[T]$ which, when applied to a function f of type $T \to \mathbb{Z}$ and a set S of type $\mathbb{P}\,T$, results in the set of elements of S which minimise f. For example

$$\begin{array}{lcl} minWRT\, f_{\#}\, \{\langle 1,2\rangle, \langle 1\rangle, \langle 2\rangle\} & = & \{\langle 1\rangle, \langle 2\rangle\} \\ minWRT\, f_{\sqcup}\, \{(2,4), (8,8), (4,7), (8,1)\} & = & \{(2,4)\} \end{array}$$

where

$$\begin{array}{lcl} f_{\#} & \hat{=} & (\textbf{fun}\ S \in Seq\,\mathbb{Z} : \#S) \\ f_{\sqcup} & \hat{=} & (\textbf{fun}\ p \in \mathbb{Z} \times \mathbb{Z} : fst\ p \sqcup snd\ p) \end{array}$$

and $\sqcup$ is the max operator introduced for the base type of integers. The definition for $minWRT[T]$ is given as

$$minWRT[T] \mathrel{\hat{=}} (\textbf{fun}\ f \in T \to \mathbb{Z} : (\textbf{fun}\ S \in \mathbb{P}\,T : \{x \in S\ :\ (\forall\, y \in S \bullet f\ x \leqslant f\ y\}))$$

Similarly, the $maxWRT[T]$ function is defined as

$$maxWRT[T] \mathrel{\hat{=}} (\textbf{fun}\ f \in T \to \mathbb{Z} : (\textbf{fun}\ S \in \mathbb{P}\,T : \{x \in S\ :\ (\forall\, y \in S \bullet f\ x \geqslant f\ y\}))$$

and results in the set of elements of S which maximise f. From the above examples,

$$\begin{array}{lcl} maxWRT\ f_{\#}\ \{\langle 1,2\rangle, \langle 1\rangle, \langle 2\rangle\} & \equiv & \{\langle 1,2\rangle\} \\ maxWRT\ f_{\sqcup}\ \{(2,4),(8,8),(4,7),(8,1)\} & \equiv & \{(8,8),(8,1)\} \\ maxWRT\ f_{\sqcap}\ \{(2,4),(8,8),(4,7),(8,1)\} & \equiv & \{(8,8)\} \end{array}$$

with $\sqcup$ (max) and $\sqcap$ (min) as before.

We also allow $minWRT[T]$ to be applied to bags and sequences, with implicit use of the $BagToSet[T]$ or $SeqToSet[T]$ functions. Thus, for B a bag

$$minWRT\ f\ B \equiv minWRT\ f\ (BagToSet\ B)$$

and similarly for sequences. Notice that the result is still a set and not a bag or sequence. This implicit conversion is merely a shorthand in the case of maximising/minimising functions, and is not a general rule.

3.2 The Form of a Specification

In this section we consider what is a specification. In its simplest form, a specification is just an expression with no free variables, with the special property that it is total. So, many of the expressions we've already seen are specifications.

In general, an expression which is a useful specification will probably be large in size, containing a number of local definitions. In such cases a clearer presentation would be to list the local definitions as named specifications, intervened with explanatory text. So, we may write a long specification, of the form

$$\textbf{let } S_1 = E_1 \,\&\, S_2 = E_2 \,\&\, \ldots \textbf{ in } E_n$$

where the E_i are typically long expressions, as

$$\begin{aligned} S_1 &\mathrel{\hat{=}} E_1 \\ S_2 &\mathrel{\hat{=}} E_2 \\ &\vdots \\ S_n &\mathrel{\hat{=}} E_n \end{aligned}$$

The convention is that S_i can appear in the specification named S_j provided $i < j$. In the above example the final specification has been given the name S_n, but the name of the final specification can be omitted.

Writing a specification in this way, as a list of sub-specifications, is simply a convenience for clear presentation. We still have a specification as a single expression. However, we frequently need to specify more than one operation in a specification document. For example, a library system will require specifications for adding a book, borrowing a book, adding a new member *etc.* Each one of these is a separate specification or expression.

In this case, we say that a specification is a collection of named specifications, and we may refer to the collection as a *specification module.* The collection is not ordered since, for example, it is not possible to say whether the operation to add a book to the library should come before the operation to add a new member. However, a named expression may be used by name within the definition of another. In this case, the defining occurence of the named expression should be presented before the expression in which it occurs, and it should be treated as a local definition for the later expression.

Within a specification expression we may need to introduce new types. For example, it would be impossible to give a library specification without referring to books, members, people *etc.* We now describe how such types may be introduced and used.

3.2.1 Types in Specification Modules

As well as the known types, and those which can be constructed using the type constructors described previously, it is also possible to introduce new types in specifications. Since we give type rules for these types, they can, in turn, be used with type constructors to form more complex types.

Given Types

These are the types which can be assumed in a specification. For example, in the library specification, we would like to use the given types *Book* and *Person* without having to explicitly say what those types are. A given type is introduced into a specification by the expression

$[typename]$

We do not know what the members of such a type are.

Although the declaration of a given type, such as

$[Person]$

means that we can now use that type in a specification, we cannot conclude any information about the elements of that type. We can ensure that the type is not empty, by using global constants (see below), but we cannot make any assumptions as to the size of the given type (as a set), or whether it contains an infinite number of values. Since, from section 2.4, each expression of the language must have a unique type, it follows that elements of the type *Person* are distinct from elements of any other type.

Global Constants

These are values of a type which are constant within a specification module. A global constant could also be handled as a parameter to each expression in the module. A global constant g is introduced into a specification module by the expression:

$| \; g : T$

where T is a type. For each expression of this form there is a corresponding introduction rule

$$\frac{}{g : T}$$

Thus we can introduce values of given types, described above. The introduction of two global constants, of the same type, does not guarantee that they are distinct values.

Datatype Definitions

These are new types with enumerated elements. For example, the type of rainbow colours

$$\mathit{Rainbow} ::= \mathit{red} \mid \mathit{orange} \mid \mathit{yellow} \mid \mathit{green} \mid \mathit{blue} \mid \mathit{indigo} \mid \mathit{violet}$$

A data type definition of the form

$$\mathit{typename} ::= v_1 \mid v_2 \mid \ldots \mid v_n$$

makes *typename* a type, and gives the introduction rules

$$\frac{}{v_1 : \mathit{typename}} \;\ldots\; \frac{}{v_n : \mathit{typename}}$$

Such a type is finite and contains exactly n elements, v_1, v_2, $\ldots v_n$. It follows that each v_i is distinct.

3.2.2 Syntax of Specifications

We give an informal syntax for specifications.

A specification may be a single expression as described previously. This may involve writing the specification as a list of subspecifications, which is purely for clarity in presentation.

A specification module begins with any number of user defined type declarations and global constants, as discussed above. This is followed by a list of expressions, separated by blank lines. The list must contain at least one expression, and the elements of the list are named, as in

$$\mathit{name} \mathrel{\hat{=}} \mathit{expression}$$

We use the convention that, within a specification module, a named expression may be subsequently used by name in a later expression. In this case the defining occurrence of the expression should be treated as a local definition for the later expression.

The notion of specification modules and named expressions is very informal. Our interest lies mainly in the use of expressions for specification, and in how such expressions may be refined. An informal treatment of specification modules allows us to group together such expressions and we shall see, in chapter 4, further notation allowing us to structure large specifications. However, if we were to provide a theory of modules and refinement of modules, it would be necessary to treat such specification structures in a more formal manner (see chapter 7).

In chapter 6 we will indicate how it might be possible to provide a semantics for specification modules. Since the syntax of specification modules is informal, it follows that the semantics will also be informal.

3.3 Examples

In this section we use the specification language to make some more interesting specifications than have already been given. A larger specification will be described in chapter 4.

We define the set *FSeq T* for any type T, to be the set of finite sequences of elements from T. Then $FSeq_1\ T$ is the set of non-empty, finite sequences of elements from T.

The multiplication problem is suggested by an example from [12].

Example : The Multiplication Problem Given two positive integers x and y each represented as a list of digits, multiply them together to form another list of digits.

We first define Digit, the set of all valid digits

$$\text{Digit} \mathrel{\hat{=}} \{x \in \mathbb{Z} : 0 \leqslant x \wedge x \leqslant 9\}$$

Then a valid number is a finite, non-empty sequence of digits not starting with '0'

$$\text{Number} \mathrel{\hat{=}} \{s \in \mathit{FSeq}_1\ \text{Digit} : s[0] \neq 0\}$$

The conversion from a Number to a positive integer is made in a standard fashion

$$\text{Convert} \mathrel{\hat{=}} (\textbf{fun}\ s \in \text{Number} : (+)/\langle i : \mathit{dom}\ s \ggg 10^{\#s-(i+1)} * s[i]\rangle)$$

Then to find a Number z which is the result of multiplying Numbers x and y is easily specified

$$\begin{aligned}\text{Multiply} \mathrel{\hat{=}} (\textbf{fun}\ x, y \in \text{Number} : \| / \{z \in \text{Number} :& \\ &\text{Convert}\ z = \text{Convert}\ x * \text{Convert}\ y\})\end{aligned}$$

.

It should be clear that the set comprehension above will result in a singleton set. We will show how to prove such a property in chapter 5.4.

Using the same style, it is possible to define other functions over positive integers represented as lists of digits, such as division and remainder

$$\begin{aligned}\text{Divide} \mathrel{\hat{=}} (\mathbf{fun}\ x, y \in \text{Number} : []/\{\ (z, r) \in \text{Number} \times \text{Number} :\ & \text{Convert}\, z = \text{Convert}\, x \text{ div Convert}\, y \\ & \wedge \text{Convert}\, r = \text{Convert}\, x \text{ mod Convert}\, y\})\end{aligned}$$

□

A familiar example is that of the N-Queens. The specification expression is also used in this specification.

Example : The N-Queens Problem To place N queens on an $N \times N$ chess board such that no queen can take any of the others.

We assume that $N \geqslant 4$. The chess board can be represented by an $N \times N$ matrix, so any position on the board can be given by its co-ordinate.

$$\text{Position} \mathrel{\hat{=}} \{1..N\} \times \{1..N\}$$

A proposed placing of the N queens will be given by a set of N positions.

$$\text{Placing} \mathrel{\hat{=}} \{Pl \in \mathbb{P}\,\text{Position} : \#Pl = N\}$$

For queens in any two positions, $p_1, p_2 \in$ Position, one queen can take the other if

- p_1 and p_2 are in the same row, $\mathbf{fst}\, p_1 = \mathbf{fst}\, p_2$;
- p_1 and p_2 are in the same column, $\mathbf{snd}\, p_1 = \mathbf{snd}\, p_2$;
- p_1 and p_2 are on the same diagonal, $\mid \mathbf{fst}\, p_1 - \mathbf{fst}\, p_2 \mid = \mid \mathbf{snd}\, p_1 - \mathbf{snd}\, p_2 \mid$.

From this we describe the property that two queens cannot take each other,

$$\begin{aligned}\text{CantTake} \mathrel{\hat{=}} (\textbf{fun}\ p_1, p_2 \in \text{Position} :\ & \\ & (\textbf{fst}\, p_1 = \textbf{fst}\, p_2 \lor \textbf{snd}\, p_1 = \textbf{snd}\, p_2 \lor \mid \textbf{fst}\, p_1 - \textbf{fst}\, p_2 \mid = \mid \textbf{snd}\, p_1 - \textbf{snd}\, p_2 \mid) \\ & \Rightarrow p_1 = p_2)\end{aligned}$$

For any placing of N queens on the $N \times N$ board, the property that no queen can take any other is given by

$$\text{SafePlacing} \mathrel{\hat{=}} (\textbf{fun}\ Pl \in \text{Placing} : (\forall\, p_1, p_2 : Pl \mid \bullet \text{CantTake}\, p_1\, p_2))$$

Now a solution to the problem is given as any safe placing.

$$\text{Solution} \mathrel{\hat{=}} [\!]/\{Pl \in \text{Placing} : \text{SafePlacing}\, Pl\}$$

□

This specification will be refined in chapter 5.4.

Another example uses the specification expression, assumptions, the *minWRT* function and exploitation of the higher-order function $map[T_1, T_2]$. This example is based on one suggested by J. Morris.

Example : The Tiling Problem A tile is a shape that can be assembled from unit squares. A rectangular tiling is a placement of tiles, without any gaps or overlappings, on a flat surface so that they form a rectangle. Given a particular shape of tile and using as many tiles as necessary, can we form a rectangular tiling?

We have an infinite grid of cells upon which all tilings are constructed. A tile placed on the grid is represented by the (finite) set of cells it occupies. A paving is a set of tiles.

$$\begin{aligned}\text{Cell} &\mathrel{\hat{=}} \mathbb{Z} \times \mathbb{Z} \\ \text{Tile} &\mathrel{\hat{=}} \mathbb{F}_1\, \text{Cell} \\ \text{Paving} &\mathrel{\hat{=}} \mathbb{P}\, \text{Tile}\end{aligned}$$

We define a function to test if a given area of the grid is a rectangle:

$$\begin{aligned}\text{isrectangle} \mathrel{\hat{=}} (\textbf{fun}\ \text{area} \in \mathbb{F}_1\, \text{Cell} : (\exists\, x, y : \mathbb{Z}, m, n \in \mathbb{N} :\ & \\ & \text{area} = \{x..x+m\} \times \{y..y+n\}))\end{aligned}$$

Then a paving is rectangular if the area it covers is a rectangle.

$\text{rectangular} \mathrel{\hat{=}} \text{isrectangle} \circ \bigcup /$

Two tiles overlap if their intersection is non-empty.

$\text{overlap} \mathrel{\hat{=}} (\textbf{fun}\ t_1, t_2 \in \text{Tile} : t_1 \cap t_2 \neq \emptyset)$

The condition that a paving contains no overlapping tiles may now be expressed.

$\text{noOverlap} \mathrel{\hat{=}} (\textbf{fun}\ p \in \text{Paving} : (\forall t_1, t_2 : p \mid \bullet \text{overlap}\ t_1\ t_2 \Rightarrow t_1 = t_2))$

Now, a given tile may be oriented in any way in order to form a paving. Any position of that tile on the grid is obtained from a combination of reflection, rotation and translation. A translation is a combination of any number of movements up, down, left or right:

$$\begin{array}{ll}
\text{reflect} & \mathrel{\hat{=}} (\textbf{fun}\ (x, y) \in \mathit{Cell} : (x, -y))* \\
\text{rotate} & \mathrel{\hat{=}} (\textbf{fun}\ (x, y) \in \mathit{Cell} : (y, -x))* \\
\text{up} & \mathrel{\hat{=}} (\textbf{fun}\ (x, y) \in \mathit{Cell} : (x, y + \mathbf{1}))* \\
\text{down} & \mathrel{\hat{=}} (\textbf{fun}\ (x, y) \in \mathit{Cell} : (x, y - \mathbf{1}))* \\
\text{left} & \mathrel{\hat{=}} (\textbf{fun}\ (x, y) \in \mathit{Cell} : (x - 1, y))* \\
\text{right} & \mathrel{\hat{=}} (\textbf{fun}\ (x, y) \in \mathit{Cell} : (x + 1, y))*
\end{array}$$

Finally, given a particular shape of tile, we first form the set of all possible positions for that tile. The set of all pavings contains all the finite pavings for that shape. We then filter out all the pavings which are non-overlapping and rectangular, and test that the set is not empty:

$$\begin{array}{l}
(\textbf{fun}\ \ \mathit{shape} \in \text{Tile} : \\
\quad \textbf{let}\ \mathit{alltiles} \quad = \bigcap/\{S \in \mathbb{P}\,\text{Tile} : \\
\qquad\qquad\qquad S = \{\mathit{shape}\} \cup (\text{reflect} * S) \cup (\text{rotate} * S) \cup (\text{up} * S) \\
\qquad\qquad\qquad\qquad \cup\, (\text{down} * S) \cup (\text{left} * S) \cup (\text{right} * S)\} \\
\quad \&\ \ \mathit{allpavings} \ = \mathbb{F}_1\ \mathit{alltiles} \\
\quad \&\ \ \mathit{rectpavings} = \text{rectangular} \lhd (\text{noOverlap} \lhd \mathit{allpavings})\ \textbf{in} \\
\quad \mathit{rectpavings} \neq \emptyset)
\end{array}$$

To find a smallest rectangular paving we need to minimise with respect to the area of the paving. We first define a function to find the size of a rectangular paving:

$$\begin{array}{l}\text{size} \mathrel{\hat{=}} (\textbf{fun}\ p \in \text{Paving} : \text{rectangular}\, p \succ\!\!- \\ \qquad [\!]/\{m, n \in \mathbb{N} : (\exists\, x, y \in \mathbb{Z} : \bigcup/p = \{x..x+m\} \times \{y..y+n\}) : m * n\})\end{array}$$

Then, assuming that a rectangular paving exists, we can find a smallest one:

$$\begin{array}{l}(\textbf{fun}\ \ \mathit{shape} \in \text{Tile} : \\ \qquad \textbf{let}\ \mathit{alltiles} \quad = [\!]/\{S \in \mathbb{P}\,\text{Tile} : \\ \qquad\qquad\qquad\qquad S = \{\mathit{shape}\} \cup (\text{reflect} * S) \cup (\text{rotate} * S) \cup (\text{up} * S) \\ \qquad\qquad\qquad\qquad\qquad\qquad \cup (\text{down} * S) \cup (\text{left} * S) \cup (\text{right} * S)\} \\ \qquad \&\ \ \mathit{allpavings} \ \ = \mathbb{F}_1\ \mathit{alltiles} \\ \qquad \&\ \ \mathit{rectpavings} = \text{rectangular} \lhd (\text{noOverlap} \lhd \mathit{allpavings})\ \textbf{in} \\ \qquad \mathit{rectpavings} \neq \emptyset \succ\!\!- [\!]/(\mathit{min\,WRT}\ \text{size}\ \mathit{rectpavings}))\end{array}$$

□

Finally, we have an example which uses the biased choice operator. This is based on an example from Dijkstra [25].

Example : Collinear Points Given a finite non-collinear set of integer-valued points in the Euclidean plane, find a line that passes through exactly two of them.

We say that a point is a pair of integers:

$$\text{Point} \mathrel{\hat{=}} \mathbb{Z} \times \mathbb{Z}$$

A line is given by two integer points.

$$\text{Line} \mathrel{\hat{=}} \text{Point} \times \text{Point}$$

Now given a line represented by the points (x_1, y_1) and (x_2, y_2), the point (x, y) is on that line if $(y - y_1) * (x - x_2) = (y - y_2) * (x - x_1)$, though we must treat separately the case where any of these terms evaluates to zero.

$$\begin{array}{l}\text{online} \mathrel{\hat{=}} (\textbf{fun}\ p \in \text{Point}, l \in \text{Line} : \\ \qquad\qquad \textbf{let}\ ((x_1, y_1), (x_2, y_2)) = l \parallel (x, y) = p\ \textbf{in} \\ \qquad\qquad\quad\ \ x = x_1 \vee x = x_2 \rightarrow x_1 = x_2 \\ \qquad\qquad\quad \underset{\leftarrow}{[\!]}\ y = y_1 \vee y = y_2 \rightarrow y_1 = y_2 \\ \qquad\qquad\quad [\!]\ (y - y_1) * (x - x_2) = (y - y_2) * (x - x_1))\end{array}$$

A given set of points is collinear if there is some line on which every point of the set occurs:

$$\text{collinear} \mathrel{\hat{=}} (\textbf{fun}\ S \in \mathbb{P}\,\text{Point} : (\exists\, l \in \text{Line} : (\forall\, p \in S : \text{online}(p, l))))$$

For the specification we need to consider only those sets which have more than one element, and whose elements are non-collinear.

$$\begin{array}{l} (\textbf{fun}\ S \in \mathbb{F}\ \text{Point} : \\ \quad \#\, S \geqslant 2 \wedge \neg\text{collinear}\, S \succ\!\!- \\ \quad [\!]/\{l \in \text{Line} : \#\{p \in S : \text{online}\, p\, l\} = 2\}) \end{array}$$

From Sylvester's theorem, stated in [25] as

> Consider a finite number of distinct points in the Real Euclidean plane; these points are collinear or there exists a straight line through exactly 2 of them.

the assumption in the above specification, $\#\, S \geqslant 2 \wedge \neg\text{collinear}\, S$, is sufficient to ensure that the set $\{l \in \text{Line} : \#\{p \in S : \text{online}\, p\, l\} = 2\}$ is non-empty.

3.4 Conclusions

In this chapter we have demonstrated the use of the expression language for specifications of a functional style. Some functions which appear often in specifications were identified and defined so that they can be used without definition in larger specifications. The concept of a specification module was introduced and this style of specification, as a collection of expressions with user-given types, was used in a number of examples. A possible semantics for specification modules will be suggested in section 6.6. A formal treatment of modules is discussed in chapter 7.

The examples illustrate the power of the specificaton language and, in particular the use of the specification expression, where the solution to a problem is expressed using a predicate. Assumptions and partial expressions were also used to formulate the example specifications, along with some of the functions from section 3.1. However, the examples given in this chapter are small examples. We need to address the problem of using the language to build larger, more useful specifications. In particular, the issue of using partiality to build specifications piecewise, on a larger scale than in section 2.6, should be examined. This issue is examined in chapter 4.

Chapter 4

Structuring Specifications

The language introduced in chapter 2 is sufficient to describe small problems, as demonstrated in chapter 3, but when attention is turned to bigger problems, the specification quickly becomes out of hand. In this chapter we examine the important, but often overlooked, issue of methods to structure large specifications.

In section 2.6 it was described how partial expressions, describing particular aspects of a specification, could be combined using choice to form a total specification. We will build on this notion and examine how partial functions, which are usually more substantial than partial expressions, can be used to construct bigger specifications in parts, and then combined using new *union* operators to form large specifications. In section 4.1 we examine the formation of partial functions, where and how they may be used and definitions of union operators. Similar to the situation for partial expressions, occurrences of partial functions are syntactically controlled. Section 4.1.3 suggests ways of manipulating partial functions using a special class of higher-order functions.

To illustrate the use of partial functions in larger specifications, in section 4.2 we describe a printing control system using the specification language of chapter 2. Some notation is first introduced which is used as a shorthand to make the specification more readable. We then show how the specification is built up, explaining why certain decisions were made, and ending with a full specification of the system in a pure functional style.

Finally, in section 4.3, we look at how the state and exception monads, used to structure functional programs, might be used to structure specifications. We describe the various monads and show how the printing control example of section 4.2 can be rewritten to take advantage of these. The resulting specification, in which details of state and error handling are hidden, is neater and more readable. In section 4.3.4 we give suggestions as to how the monads could be expressed in the specification language.

4.1 Partial Specifications

In section 2.6 we looked at potentially partial expressions and how they can be used to specify a problem in parts which are then combined to form the complete specification. Because partial expressions are potentially miraculous, the syntax of the language has been restricted so that potentially partial expressions may be direct arguments of choice $[]$ and biased choice $\overset{\leftarrow}{[]}$ only. Such a restriction is possible because potentially partial expressions can arise in exactly two ways, from a guarded expression or from a specification expression. Such expressions may be 'totalised', as discussed in section 2.6.2, using the biased choice operator $\overset{\leftarrow}{[]}$.

In this section we examine how partial functions can be used to structure large specifications. During the construction of a specification we claim that it is useful to allow an abstraction over a non-total expression, *i.e.* the formation of a partial function, with the intention that it be combined with other, possibly partial, functions at a later stage. In the same way that partial expressions are used for small specifications, partial functions are a useful concept in the language because they permit large specifications to be constructed in parts, with separation of concerns a major issue.

The intention is that a specification is written describing a result in a certain, perhaps error-free, case, generally of the form $(\mathbf{fun}\ x \in T : B \to E)$ where E is typically a large expression. The "error" case is described separately, perhaps of the form $(\mathbf{fun}\ x \in T : \neg B \to F)$. These two partial functions should be combined to form a new specification given by $(\mathbf{fun}\ x \in T : B \to E \,\|\, \neg B \to F)$. For example, the searching function for sequences of type $SeqT$ could be written as

$$(\mathbf{fun}\ S \in SeqT, x \in T : []/\{i \in \{0..\#S-1\} \,:\, S[i] = x\}) \tag{4.1}$$

This is a partial function since it yields $\top$ if the given x does not occur in the sequence. It could be made into a total function by combining it, for example, with a function which returns a default error value if the given value x does not occur in the sequence.

The Z specification language [75] permits the construction of specifications by combining schemas, which can be compared to partial functions. In a Z specification it is usual to combine schemas for partial specifications using schema disjunction. We will propose a similar method for combining partial functions.

Note the distinction between a total function and a total expression. A function expression can be total, while still being a partial function, *i.e.* its body is potentially partial. An example of this phenomenon is the search function (4.1).

4.1.1 Using Partial Functions

With the syntax rules given so far, we cannot construct partial functions, since all possible occurrences of possibly non-total expressions must be totalised before being used with the language constructors such as pairing, function application and, in particular, abstraction. We consider what happens when this rule is relaxed to allow abstraction over non-total expressions to form partial functions, as described above.

These functions are total expressions and, as such, there is no restriction on where they may occur, subject to typing conditions. This causes some problems, particularly with function application.

We consider the application of a partial function to some argument for which a result has not been specified in the function body. According to the axioms the result of the application is the value $\top$. So, for example, the result of the application

$$(\textbf{fun}\ x \in \mathbb{Z} : x \geqslant 0 \to `+\text{'})(-7)$$

is $\top$ and thus the expression is not total. From the example it is clear that, although in order to form the expression $(f\ e)$ both f and e must be total, it is possible that the new expression $(f\ e)$ is not total.

The result of allowing such applications is that a new form of potentially partial expression has been admitted, that of a function application. Rather than complicating specifications by requiring that all expressions of the form $(f\ e)$ are totalised, we instead insist that all functions occurring within an expression are total functions.

The admission of partial functions is intended only as a structuring agent for large specifications. This means that they should only be used in certain ways and otherwise must be totalised, just as partial expressions require to be totalised before being used.

Similar to the syntactic restrictions for partial expressions, we now require that potentially partial functions occur only as direct arguments of choice $[\!]$ and the syntactic union operators $\dot{\cup}$ and $\overleftarrow{\cup}$ which will be defined in section 4.1.2. Since the test for total functions is a syntactic one, this restriction can be imposed as a syntax rule.

4.1.2 Combining Partial Functions

Allowing the formation of partial functions results in the ability to build a specification in parts. This promotes the 'separation of concerns' approach to specification. Its intended use is in the specification of a result in a certain, perhaps error-free case, generally of the form:

$$(\mathbf{fun}\ x \in T : B \to E)$$

which we would like to make total by *combining* it with the specification describing the result in the "error" case:

$$(\mathbf{fun}\ x \in T : \neg B \to F)$$

Our aim in this section is to define an operator $\dot{\cup}$ which will take two partial functions and combine them such that

$$(\mathbf{fun}\ x \in T : E)\,\dot{\cup}\,(\mathbf{fun}\ x \in T : F) \quad \equiv \quad (\mathbf{fun}\ x \in T : E \mathbin{[\!]} F)$$

Since the formation and combination of partial functions appears to be a purely syntactic notion, it makes sense that the definition of $\dot{\cup}$ should also be syntactic. Restrictions to occurrences of $\dot{\cup}$ are that it is used only with function types. From the discussion in section 4.1.1, the functions must be of the form $(\mathbf{fun}\ x \in T : E)$, or a choice between functions of this form. The two defining rules for $\dot{\cup}$ are, therefore

$$\begin{aligned}(\mathbf{fun}\ x \in T : E)\,\dot{\cup}\,(\mathbf{fun}\ x \in T : F) &\ \hat{=}\ (\mathbf{fun}\ x \in T : E \mathbin{[\!]} F)\\ f\,\dot{\cup}\,(g_1 \mathbin{[\!]} g_2) &\ \hat{=}\ (f\,\dot{\cup}\,g_1) \mathbin{[\!]} (f\,\dot{\cup}\,g_2)\end{aligned}$$

Since choice is commutative, so also is $\dot{\cup}$.

Taking the union of two partial functions yields another partial function. We define another version of union, a biased union, which can be used to obtain a total function. A function $(f \overleftarrow{\cup} g)$ when applied to an argument e will result in $(f\ e)$ if it is total and otherwise $(g\ e)$. The definition is purely syntactic, with the defining rules given by

$$\begin{aligned}(\mathbf{fun}\ x \in T : E)\,\overleftarrow{\cup}\,(\mathbf{fun}\ x \in T : F) &\ \hat{=}\ (\mathbf{fun}\ x \in T : E \mathbin{\overleftarrow{[\!]}} F)\\ f\,\overleftarrow{\cup}\,(g_1 \mathbin{[\!]} g_2) &\ \hat{=}\ (f\,\overleftarrow{\cup}\,g_1) \mathbin{[\!]} (f\,\overleftarrow{\cup}\,g_2)\end{aligned}$$

Commutativity does not hold, in general, for $\overleftarrow{\cup}$. Moreover, $\overleftarrow{\cup}$ does not left-distribute over choice, which is why the left argument of $\overleftarrow{\cup}$ may not be a choice between functions. We see that a function $f \overleftarrow{\cup} g$ is guaranteed to be a total function if g is. Thus, the biased union can be used to form total functions.

4.1.3 Manipulating Partial Functions

We have suggested the use of partial functions as a means to construct a specification piecewise, so that the partial functions can be combined to form a complete specification. However, we may also want to manipulate partial functions. This means allowing certain higher-order functions to be applied to potentially partial functions.

In general, partial functions are not permitted as arguments to higher-order functions, for the reason that this might introduce partiality into a specification. For example, if f is a partial function, then it is not clear exactly what should be the meaning of $f*$ applied to a set, or whether such an expression is useful.

However, we propose a class of higher-order functions which may be applied to partial functions, and for which the resulting application is guaranteed to be total. Consider a higher-order function which takes two arguments, a possibly partial function f of type $\mathbb{Z} \to \mathbb{Z}$ and a string (sequence of characters) s. The result is a total function of type $\mathbb{Z} \to (\mathbb{Z} \times \mathit{String})$ which behaves in the following way: when applied to an argument x, if $(f\ x)$ is total then it returns the pair consisting of the value $(f\ x)$ and the string 'ok', otherwise it returns the pair $(0, s)$. Without the possibility of having partial functions, we could not specify this higher-order function. The specification can be expressed by

$$\begin{aligned} \text{totalise} \;\hat{=}\; & (\mathbf{fun}\ f \in \mathbb{Z} \to \mathbb{Z}, s \in \mathit{String} : \\ & \quad (\mathbf{fun}\ x \in \mathbb{Z} : ((f \overleftarrow{\cup} \text{zero})x, (x \in \mathit{dom}\, f \to \text{'}\mathit{ok}\text{'}) \overleftarrow{[]}\ s))) \end{aligned}$$

where zero $\hat{=}$ $(\mathbf{fun}\ x \in \mathbb{Z} : 0)$, and the function dom, when applied to a partial function f, returns the set of values for which f has been specified. Notice that the 'totalise' function, being a total function, can now be used to totalise a partial function.

There is no syntactic method to recognise higher-order functions that can be applied safely to partial functions. They will be used only to add clarity to specifications and when it is clear that the evaluation of their application would give a syntactically correct specification.

4.2 A Printing Control Example

In this section we use the example of a printing control system to show how we can use partial functions to help structure large specifications. We also introduce some notation which helps to make the specification more readable. The complete specification is reproduced in appendix B. In the next section (4.3) we will use the same example but with monads to help hide the details of state and error handling.

4.2.1 Notation

In the following example, of a printing control system, we use some notation which is introduced in this section.

In most specifications of any size a concept of state is required. In a functional world, the state can be passed as an argument from function to function, but this can make for unnecessarily cluttered specifications. We use a simple, though naive, notation to unclutter such specifications.

Record definitions are simply a syntactic shorthand for the specification of tuples with associated retrieval functions. They are used to make specifications shorter while increasing clarity and readability.

Detached Parameters

We may sometimes wish to detach, or make less explicit the parameters to a specification, in order to make the specification more readable. In the printer control system, we make the variable representing the state less explicit so that the main elements of the specification can be more evident.

A list of variables with their type information, $x_1 : T_1, \ldots, x_n : T_n$, which we write $x : T$ for convenience, is detached from an expression E using the notation

$$x : T \quad \vdash \quad name \mathrel{\hat{=}} E \tag{4.2}$$

where x may occur free in E. This specification is exactly the same as the definition

$$name \quad \hat{=} \quad (\mathbf{fun}\ x \in T : E) \tag{4.3}$$

It should be clear that, in specification (4.2), the argument x has simply been moved to a position where it may be less intrusive in the reading of expression E.

Having given a definition for *name* we expect that it will be used elsewhere in the specification. Since, from (4.3), *name* represents a function, we expect it to be applied to an argument. So, subsequent appearances of *name* are likely to be of the form

$$F[name\ e]$$

where e is an expression of type T. Unfolding the definition of *name*, this is the same as

$$F[(\mathbf{fun}\ x \in T : E)e]$$

as expected.

More generally, we can have a list of specifications of the form

$$\begin{array}{lll} x : T & \vdash & name_1 \mathrel{\hat{=}} E_1, \\ & & name_2 \mathrel{\hat{=}} E_2, \\ & & \vdots \\ & & name_n \mathrel{\hat{=}} E_n \end{array}$$

which is just shorthand for

$$\begin{array}{lll} x : T & \vdash & name_1 \mathrel{\hat{=}} E_1 \\ x : T & \vdash & name_2 \mathrel{\hat{=}} E_2 \\ & & \vdots \\ x : T & \vdash & name_n \mathrel{\hat{=}} E_n \end{array}$$

and so any E_i may contain $name_j$ provided $j < i$.

Record Definitions

In conjunction with the introduction of detached parameters, we have a shorthand notation for the specification of tuples with associated retrieval functions. For example, in the specification to follow we have the notion of a CurrentJob which is made up of the JobId and the number of pages printed so far. For every possible CurrentJob we want the ability to retrieve either of its components. We write the following specification

$$\begin{array}{l} \textsc{CurrentJob} \mathrel{\hat{=}} \textsc{JobId} \times \mathbb{N} \\ c : \textsc{CurrentJob} \vdash \text{CurrentId} \mathrel{\hat{=}} \pi_1\ c, \\ \qquad\qquad\qquad\qquad\ \text{PagesPrinted} \mathrel{\hat{=}} \pi_2\ c \end{array}$$

Instead of writing this specification out in full, we use the shorthand

$c : \textsc{CurrentJob} \mathrel{\hat{=}} [\mathrm{CurrentId} \in \textsc{JobId}, \mathrm{PagesPrinted} \in \mathbb{N}]$

In general, a specification of the form

$$r : \mathrm{R} \mathrel{\hat{=}} [X_1 \in T_1, \ldots, X_n \in T_n]$$

where the X_i are names and the T_i are sets (or types), is shorthand for the specification

$$\begin{array}{lll} \mathrm{R} \mathrel{\hat{=}} T_1 \times \cdots \times T_n & & \\ r : \mathrm{R} & \vdash & X_1 \mathrel{\hat{=}} \pi_1\, r, \\ & & \vdots \\ & & X_n \mathrel{\hat{=}} \pi_n\, r \end{array}$$

Often it is required that not all possible tuples are included in the set R, but rather just those which satisfy some requirement. In this case we add a predicate to the record definition. So, a specification of the form

$$r : \mathrm{R} \mathrel{\hat{=}} [X_1 \in T_1, \ldots, X_n \in T_n] : P(X_1, \ldots, X_n)$$

where P is a predicate over $X_1, \ldots, X_n$, is shorthand for the specification

$$\begin{array}{lll} \mathrm{R} \mathrel{\hat{=}} \{(X_1, \ldots, X_n) \in T_1 \times \cdots \times T_n : P(X_1, \ldots, X_n)\} & & \\ r : \mathrm{R} & \vdash & X_1 \mathrel{\hat{=}} \pi_1\, r, \\ & & \vdots \\ & & X_n \mathrel{\hat{=}} \pi_n\, r \end{array}$$

This form of record definition can be used, for example, to specify tuples consisting of a printer quota and the number of pages printed by a specific person.

$$r : \mathrm{R} \mathrel{\hat{=}} [\mathrm{Quota} \in \mathbb{N}, \mathrm{PagesPrinted} \in \mathbb{N}] : (\mathrm{Quota} \geqslant \mathrm{PagesPrinted})$$

In this case, the number of pages printed should be less than the quota.

The following specification, of a printing control system, demonstrates the use of both detached parameters and record definitions.

4.2.2 Problem Description

Example : Printing Control System A printing control system manages the allocation of page quotas to users, and provides such operations as:

- Allocate a page quota to a user.
- Add a print job to a print queue with a given priority.
- Give the print job that is active, the number of pages printed for this job so far, and the number of pages still to be printed.
- "Print" the next page of the active job, moving on to the next job (with the highest priority) if the active job is finished.
- Remove a print job from the print queue.

etc.

4.2.3 Building the Specification

We assume two sets, PERSON and PAGE

[PERSON] , [PAGE]

We define the following sets:

$$\begin{aligned}
\textsc{JobId} &\mathrel{\hat=} \mathbb{N} \\
\textsc{File} &\mathrel{\hat=} \mathit{Seq}\textsc{Page} \\
\textsc{Priority} &\mathrel{\hat=} \mathbb{N} \\
\textsc{Buffer} &\mathrel{\hat=} \textsc{Page}
\end{aligned}$$

We have a mapping for information about specific jobs, with corresponding retrieval functions

$$\begin{aligned}
\mathit{inf} : \textsc{Jobs} \mathrel{\hat=} [&\text{KnownJobs} \in \mathbb{P}\,\textsc{JobId} \\
&\text{FileOf} \in \text{KnownJobs} \twoheadrightarrow_t \textsc{File}, \\
&\text{OwnerOf} \in \text{KnownJobs} \twoheadrightarrow_t \textsc{Person}, \\
&\text{PriorityOf} \subset \text{KnownJobs} \twoheadrightarrow_t \textsc{Priority}]
\end{aligned}$$

$$\mathit{inf} : \textsc{Jobs} \vdash \text{SizeOf} \mathrel{\hat=} \# \circ \text{FileOf}$$

where $\twoheadrightarrow_t$ denotes a total mapping from the domain set, in this case the set KnownJobs.

The current job (being printed) is identified by its JOBID, but we also need to know how many pages have been printed so far

$$c : \textsc{CurrentJob} \mathrel{\hat{=}} [\text{CurrentId} \in \textsc{JobId}, \text{PagesPrinted} \in \mathbb{N}]$$

The jobs waiting to be printed go into the PRINTQUEUE. We use an injective sequence for the queue, to ensure that no two jobs in the Jobqueue can have the same JOBID.

$$\begin{aligned} &\textsc{PrintQueue} \mathrel{\hat{=}} \text{ISeq}(\textsc{JobId}\backslash\{0\}) \\ &q : \textsc{PrintQueue} \vdash \text{JobsWaiting} \mathrel{\hat{=}} \operatorname{ran} q, \\ &\qquad\qquad\qquad\qquad \text{RemQueue} \mathrel{\hat{=}} (\textbf{fun } id \in \textsc{JobId} : \text{Remove}(q, id)) \end{aligned}$$

where an operation to Remove some occurence of a given element from a sequence, or the occurence of an element from an injective sequence, can be added to the collection of operations over sequences. Its definition may be given as

$$\begin{aligned} \text{Remove}(x, S) \mathrel{\hat{=}} []/\{S' \in \text{Seq}\,T : (\exists i \in \{0 \ldots \#S\} : \\ S = S'[0 \ldots i] ^\frown \langle x \rangle ^\frown S'[i \ldots \#S])\} \\ \overleftarrow{[]}\, S \end{aligned}$$

for $x : T$ and $S : \text{Seq}\,T$ for some type T. The sequence is left unchanged if it does not contain the given element.

The current state of the printer queue is given by the PRINTQUEUE and the CURRENTJOB. The state queue is empty whenever the JOBID of the CURRENTJOB is zero.

$$\begin{aligned} q : \textsc{PrintQueue}, c : \textsc{CurrentJob} \vdash \text{JobsInQueue} &\mathrel{\hat{=}} \text{JobsWaiting} \cup \text{CurrentId}, \\ \text{EmptyQueue} &\mathrel{\hat{=}} (\text{CurrentId} = 0) \end{aligned}$$

We have a mapping for known users of the printing system to their quota and the number of pages used so far. Clearly, the quota should exceed the number of pages used.

$$\begin{aligned} u : \textsc{Users} \mathrel{\hat{=}} [&\text{KnownUsers} \in \mathbb{P}\,\textsc{Person}, \\ &\text{QuotaOf} \in \text{KnownUsers} \twoheadrightarrow_t \mathbb{N}, \\ &\text{PagesUsedBy} \in \text{KnownUsers} \twoheadrightarrow_t \mathbb{N}] : \\ &(\forall\, p \in \textsc{Person}.\text{QuotaOf}\, p \geqslant \text{PagesUsedBy}\, p) \end{aligned}$$

Now the state of the system is made up of five components, the PRINTQUEUE, the CURRENTJOB, a BUFFER for printing, information about the JOBS, and information about the USERS. Such a state must satisfy certain constraints, such as the number of pages printed of the current job cannot exceed the size of the job, the domain of the job information must be the same as the set of JOBIDs in the queue, and the owner of every job in the queue must be a known user.

$$
\begin{aligned}
\sigma : \Sigma \mathrel{\hat{=}} [&q \in \textsc{PrintQueue}, c \in \textsc{CurrentJob}, b \in \textsc{Buffer}, \mathit{inf} \in \textsc{Jobs}, u \in \textsc{Users}] : \\
&(\mathrm{PagesPrinted} \leqslant \mathrm{SizeOf} \circ \mathrm{CurrentId} \\
&\wedge\ \mathrm{KnownJobs} = \mathrm{JobsInQueue} \\
&\wedge\ \mathrm{KnownUsers} \supseteq \mathrm{OwnerOf} * \mathrm{JobsInQueue} \\
&\wedge\ \mathrm{CurrentId} \notin \mathrm{JobsWaiting} \\
&\wedge\ (\mathrm{CurrentId} = 0 \Rightarrow q = \langle\rangle)
\end{aligned}
$$

We now specify one of the operations described above, to add a print job to a print queue with a given priority. This is done in two stages, one where the owner of the file is known to the system, and the second in the error case where the owner is not known. If the job-owner is known, then we need to get a new job number and record the new job information. If the printer queue is empty, then the new job should become current immediately, otherwise it is added to the jobqueue

$$
\begin{aligned}
\sigma : \Sigma \vdash \mathrm{AddOk} \mathrel{\hat{=}} (&\mathbf{fun}\ p \in \textsc{Person}, f \in \textsc{File}, n \in \textsc{Priority} : \\
&p \in \mathrm{KnownUsers} \rightarrow \\
&\quad \mathbf{let}\ \mathit{newId} = [\,]/(\mathbb{N}\backslash(\{0\} \cup \mathrm{KnownJobs})) \\
&\quad \&\ \ \mathit{newq} = (\neg\mathrm{EmptyQueue} \rightarrow q \frown \langle \mathit{newId}\rangle \overleftarrow{[\,]}\ q) \\
&\quad \&\ \ \mathit{newc} = (\neg\mathrm{EmptyQueue} \rightarrow c\ \overleftarrow{[\,]}\ (\mathit{newId}, 0)) \\
&\quad \&\ \ \mathit{newInf} = (\mathrm{FileOf} \oplus \{\mathit{newId} \mapsto f\}, \\
&\qquad\qquad \mathrm{OwnerOf} \oplus \{\mathit{newId} \mapsto p\}, \\
&\qquad\qquad \mathrm{PriorityOf} \oplus \{\mathit{newId} \mapsto n\}) \\
&\quad \mathbf{in}\ (\mathit{newq}, \mathit{newc}, b, \mathit{newInf}, u))
\end{aligned}
$$

For the error case it is probable that we would want to report some error, but this hasn't been given in the informal specification. We simply have:

$$
\begin{aligned}
\sigma : \Sigma \vdash \mathrm{AddError} \mathrel{\hat{=}} (&\mathbf{fun}\ p \in \textsc{Person}, f \in \textsc{File}, n \in \textsc{Priority} : \\
&\qquad\qquad \textsc{Unknown\_User\_Error})
\end{aligned}
$$

We have not said what UNKNOWN_USER_ERROR is, but we shall see more examples of this form of expression in the rest of the specification. It can be regarded as a special sort of expression, of the appropriate type, highlighting a part of the specification which has not yet been fully specified. But this explanation is not entirely satisfactory. Error-handling in a functional setting is a known problem and there do exist techniques to deal with it. One such approach will be considered in section 4.3.2.

The complete specification to add a job to the queue is then

$$\sigma : \Sigma \vdash \text{Add} \mathrel{\hat{=}} \text{AddOk} \overset{\leftarrow}{\cup} \text{AddError}$$

Another operation required of the queue system is to allocate a page quota to a person. We assume two possibilities. Either the person is a new user, or the person is already known as a user and is getting a new quota, with the number of pages used being reset to zero. In the first case we have

$$\begin{aligned}\sigma : \Sigma \vdash \text{NewUser} \mathrel{\hat{=}} (&\textbf{fun}\ p \in \text{PERSON}, q \in \mathbb{N} :\\ &p \notin \text{KnownUsers} \rightarrow \textbf{let}\ newu = (\text{QuotaOf} \oplus \{p \mapsto q\},\\ &\qquad\qquad \text{PagesUsedBy} \oplus \{p \mapsto 0\})\\ &\qquad \textbf{in}\ (q, c, b, inf, newu))\end{aligned}$$

In the second case, we give a new quota and reset the number of pages printed

$$\begin{aligned}\sigma : \Sigma \vdash \text{ResetQuota} \mathrel{\hat{=}} (&\textbf{fun}\ p \in \text{PERSON}, q \in \mathbb{N} :\\ &p \in \text{KnownUsers} \rightarrow \textbf{let}\ newu = (\text{QuotaOf} \oplus \{p \mapsto q\},\\ &\qquad\qquad \text{PagesUsedBy} \oplus \{p \mapsto 0\})\\ &\qquad \textbf{in}\ (q, c, b, inf, newu))\end{aligned}$$

The complete specification to allocate a quota is then

$$\sigma : \Sigma \vdash \text{Alloc} \mathrel{\hat{=}} \text{NewUser} \dot{\cup} \text{ResetQuota}$$

Further examination reveals that the two specifications Newuser and Resetquota are almost exactly the same. The Alloc specification is, in fact, equivalent to

$$\begin{aligned}\sigma : \Sigma \vdash \text{Alloc} \mathrel{\hat{=}} (&\textbf{fun}\ p \in \text{PERSON}, q \in \mathbb{N} :\\ &\textbf{let}\ newu = (\text{QuotaOf} \oplus \{p \mapsto q\},\\ &\qquad \text{PagesUsedBy} \oplus \{p \mapsto 0\})\\ &\textbf{in}\ (q, c, b, inf, newu))\end{aligned}$$

A proof of this equivalence will be given in section 5.4.2.

The operation which returns the print job that is active, the number of pages printed so far and the number still to be printed is given as

$$\begin{aligned}
\sigma : \Sigma \vdash \text{Active} \mathrel{\hat{=}} (\neg\text{EmptyQueue} \rightarrow\ & \mathbf{let}\ id = \text{CurrentId} \parallel n = \text{PagesPrinted} \\
& \&\quad size = \text{SizeOf}\, id \\
& \mathbf{in}\ (id, n, size - n) \\
\overset{\leftarrow}{[\!]}\ & \text{QUEUE\_EMPTY\_ERROR})
\end{aligned}$$

We now consider the 'print' operation, which puts the next page of the current document into the buffer to be printed, and moving on to the next job, with the highest priority, if the active job is finished. We first specify the case where the queue is not empty and the owner of the current job has enough quota left to print the next page

$$\begin{aligned}
\sigma : \Sigma \vdash \text{PrintOk} \mathrel{\hat{=}} (\neg\text{EmptyQueue} \rightarrow & \\
& \mathbf{let}\ id = \text{CurrentId} \parallel n = \text{PagesPrinted} \\
& \&\quad p = \text{OwnerOf}\, id \parallel f = \text{FileOf}\, id \\
& \&\quad quota = \text{QuotaOf}\, p \parallel pages = \text{PagesUsedBy}\, p\ \mathbf{in} \\
& quota > pages \rightarrow
\end{aligned}$$

We 'print' the next page and adjust the number of pages printed for the owner of the job

$$\begin{aligned}
& \mathbf{let}\ newb = f[n] \\
& \&\quad newu = \text{ChangeUser}(quota, pages + 1)\ \mathbf{in}
\end{aligned}$$

Now there are two cases. For the first possibility there is more of the current document still to print, so we just record that one more page has printed of the job

$$\begin{aligned}
& (n < \text{SizeOf}\, id \rightarrow \\
& \qquad \mathbf{let}\ newc = (id, n + 1) \\
& \qquad \mathbf{in}\ (q, newc, inf, newu, newb)
\end{aligned}$$

For the second possibility the next job with the highest priority is made current

$$\begin{aligned}
\overset{\leftarrow}{[\!]}\quad & \mathbf{let}\ newid = \text{GetNextId} \\
& \&\quad newc = (newid, 0) \\
& \&\quad newq = remove\ newid \\
& \&\quad newInf = \text{RemInf}\, id \\
& \mathbf{in}\ (newq, newc, newinf, newu, newb)))
\end{aligned}$$

where GetNextId gives the JobId of the first job in the PrintQueue with the highest priority, or zero if the queue is empty

$$\begin{aligned} q : \textsc{PrintQueue}, inf : \textsc{Jobs} \vdash \mathrm{GetNextId} \mathrel{\hat{=}} (q \neq \langle\rangle \to\ & \\ & \mathbf{let}\ pr = (\mathbf{fun}\ i \in \mathbb{N} : \mathrm{PriorityOf}\, q[i]) \\ & \mathbf{in}\ \sqcap / (maxWRT\, pr\{0..\#q - 1\})) \\ \overset{\leftarrow}{[]}\ & 0) \end{aligned}$$

The PrintOk function does not handle the cases when the user doesn't have enough quota or the printer queue is empty. These are treated separately

$$\sigma : \Sigma \vdash \mathrm{QuotaError} \mathrel{\hat{=}} (\neg \mathrm{EmptyQueue} \to \textsc{Quota\_Error})$$

And if the queue is empty, we already have the function from the Active specification

$$\sigma : \Sigma \vdash \mathrm{QEmpty} \mathrel{\hat{=}} \textsc{Error\_Queue\_Empty}$$

The complete specification to print a page is

$$\sigma : \Sigma \vdash \mathrm{Printpage} \mathrel{\hat{=}} \mathrm{Printok} \overset{\leftarrow}{[]} \mathrm{QuotaError} \overset{\leftarrow}{[]} \mathrm{QEmpty}$$

Our final specification is, given a JobId, remove that job from the printer queue. This can only happen if the job is in the queue, and it is not the active job

$$\begin{aligned} \sigma : \Sigma \vdash \mathrm{RemoveOk} \mathrel{\hat{=}} (\mathbf{fun}\ id \in \textsc{JobId} :\ & \\ & id \in \mathrm{JobsInQueue} \wedge id \neq \mathrm{CurrentId} \to \\ & \quad \mathbf{let}\ newq = \mathrm{RemQueue}\, id \\ & \quad \&\ \ newinf = (\mathrm{FileOf} \backslash id, \\ & \qquad\qquad \mathrm{OwnerOf} \backslash id, \\ & \qquad\qquad \mathrm{PriorityOf} \backslash id) \\ & \quad \mathbf{in}\ (newq, c, b, newinf, u)) \end{aligned}$$

An error is reported if either the job to be killed is the current job

$$\begin{aligned} \sigma : \Sigma \vdash \mathrm{RemoveCurrent} \mathrel{\hat{=}} (\mathbf{fun}\ id \in \textsc{JobId} :\ & \\ & id = \mathrm{CurrentId} \to \textsc{Current\_Job\_Error}) \end{aligned}$$

or if it isn't in the queue

$\sigma : \Sigma \vdash \text{RemoveFail} \mathrel{\hat{=}} (\textbf{fun}\ id \in \textsc{JobId} : \textsc{Job\_not\_in\_Queue\_Error})$

The complete specification to remove a job from the queue is given as

$\sigma : \Sigma \vdash \text{RemoveJob} \mathrel{\hat{=}} (\text{RemoveOk} \overleftarrow{\cup} \text{RemoveCurrent}) \overleftarrow{\cup} \text{RemoveFail}$

The full specification for the printer control system can be found in Appendix B.

4.3 Using Monads

The concept of a monad, which is simply a form of abstraction with certain properties, comes from category theory [8]. Monads have been used in computer science, for example, to structure the denotational semantics of programming languages [53, 52, 54] with the aim of providing a unified approach. Another application of monads is in the structuring of pure functional programs that mimic impure features such as state, exceptions and continuations [88, 89, 72, 48]. In this section we apply the same theory to structure the printer control specification of section (4.2). We use a monad to help hide the explicit printer state and to control error handling.

We take a very simple definition of a monad, where no knowledge of category theory is assumed. From [89], a monad is a triple $(M, unit, \star)$ where M is a type constructor, and $unit$ and $\star$ are polymorphic functions with types

$$\begin{array}{rcl} unit & :: & a \rightarrow Ma \\ (\star) & :: & Ma \rightarrow (a \rightarrow Mb) \rightarrow Mb \end{array}$$

for a and b types. These operations must satisfy three laws

$$\begin{array}{lcll} unit\ a \star \lambda\, b.n & = & n[a/b] & (\textit{Left unit}) \\ m \star \lambda\, a.unit\ a & = & m & (\textit{Right unit}) \\ m \star (\lambda\, a.n \star \lambda\, b.o) & = & (m \star \lambda\, a.n) \star \lambda\, b.o & (\textit{Associative}) \end{array}$$

The third law is valid only when a does not appear free in o. These laws are only the basic laws, and can lead to a list of other laws useful for equational reasoning, as described in [88].

4.3.1 The State Monad

In pure functional languages, state may be handled explicitly by passing around a value representing the current state, as in the printer control example of the previous section (4.2). Descriptions of the monad to help hide this explicit state can be found in [88, 89, 72, 48]. The key idea is that of a state transformer.

A state transformer is an object of type $ST_S A$, for S the type of states and arbitrary type A, where $ST_S A$ is defined to be the function type $S \to (A \times S)$. So, a state transformer transforms a state and produces something of type A. Useful functions over state transformers, with their types, which are described in [88], include

$$\begin{aligned} \mathit{unit} &: A \to ST_S A \\ \mathit{unit} &\mathrel{\hat{=}} (\textbf{fun}\ a \in A : (\textbf{fun}\ s \in S : (a, s))) \end{aligned}$$

which, given a value a, returns that value without transforming the state. This function is called *returnST* in [48];

$$\begin{aligned} \mathit{fetch} &: ST_S S \\ \mathit{fetch} &\mathrel{\hat{=}} (\text{fun}\ s \in S : (s, s)) \end{aligned}$$

which simply returns the state as the value without transforming the state;

$$\begin{aligned} \mathit{assign} &: S \to ST_S() \\ \mathit{assign} &\mathrel{\hat{=}} (\text{fun}\ s' \in S : (\text{fun}\ s \in S : ((), s'))) \end{aligned}$$

where () is the type containing only the value (). Given a state s', *assign* changes the state to s' and returns no value.

The important function for glueing together state transformers is the infix function $(\star)$

$$\begin{aligned} (\star) &: ST_S A \to (A \to ST_S B) \to ST_S B \\ m \star k &= (\textbf{fun}\ s \in S : \textbf{let}\ (a, s') = ms\ \textbf{in}\ k\ a\ s') \end{aligned}$$

Together *unit* and $(\star)$, with the constructor ST, form a monad, satisfying the laws given above, which can be used in equational reasoning, [88, 89].

A state transformer may have additional arguments, or other inputs, when its type will be a function type, returning a state transformer. For example, a state transformer of type $B \to ST_S A$ takes something in B, transforms the state and produces something in A. We can examine the specification of the printer control system in this light.

The Printer Control System using the State Monad

Assume the given sets, initial definitions and definitions for state are as before, but in their unfolded form. Our state type S for the state transformers is Σ.

The Add function now has the type

$$\textsc{Person} \times \textsc{File} \times \textsc{Priority} \rightarrow ST_{\Sigma}()$$

since, given a $\textsc{Person}$, $\textsc{File}$ and $\textsc{Priority}$, it will transform the state without producing any value. The specification becomes

$$\begin{array}{l}
\text{AddOk} \mathrel{\hat{=}} (\mathbf{fun}\ p \in \textsc{Person}, f \in \textsc{File}, n \in \textsc{Priority} : \\
\quad \mathit{fetch} \star (\mathbf{fun}\ (q, c, b, \mathit{inf}, u) \in \Sigma : p \in \text{KnownUsers}\, u \rightarrow \\
\qquad \mathbf{let}\ \mathit{newId} = \sqcap/(\mathbb{N}\backslash(\{0\} \cup \text{KnownJobs}\, \mathit{inf})) \\
\qquad \&\quad \mathit{newq} = (\neg\text{EmptyQueue}(q, c) \rightarrow q \frown \langle \mathit{newId} \rangle \overleftarrow{\Box}\ q) \\
\qquad \&\quad \mathit{newc} = (\neg\text{EmptyQueue}(q, c) \rightarrow c \overleftarrow{\Box}\ (\mathit{newId}, 0)) \\
\qquad \&\quad \mathit{newInf} = (\text{FileOf}\, \mathit{inf} \oplus \{\mathit{newId} \mapsto f\}, \\
\qquad\qquad\qquad \text{OwnerOf}\, \mathit{inf} \oplus \{\mathit{newId} \mapsto p\}, \\
\qquad\qquad\qquad \text{PriorityOf}\, \mathit{inf} \oplus \{\mathit{newId} \mapsto n\}) \\
\qquad \mathbf{in}\ \mathit{assign}(\mathit{newq}, \mathit{newc}, b, \mathit{newInf}, u)))
\end{array}$$

The initial *fetch* returns the state as a value, and is used to make the state explicit. This 'value' is then passed to a function, of type $\Sigma \rightarrow ST_{\Sigma}()$, which uses the *assign* function to replace the input state by a new updated state, and produces the empty result ().

Unfortunately, the expression for AddOk given above is not correct according to our syntax rules. A potentially partial expression, here of the form $(P \rightarrow F)$ is permitted only at the top level of a function body, with the intention that the resulting partial function is to be combined immediately, using $\dot{\cup}$ or $\overleftarrow{\cup}$, with other partial functions to form a total specification. In the above, a partial function is correctly formed but immediately used as an argument to $\star$, which is not allowed according to this rule.

Instead, we must write the Add specification in one, as follows

$$\begin{array}{l}
\text{Add} \mathrel{\hat{=}} (\mathbf{fun}\ p \in \textsc{Person}, f \in \textsc{File}, n \in \textsc{Priority} : \\
\quad \mathit{fetch} \star (\mathbf{fun}\ (q, c, b, \mathit{inf}, u) \in \Sigma : \\
\qquad (p \in \text{KnownUsers}\, u \rightarrow \\
\qquad\quad \mathbf{let}\ \mathit{newId} = \sqcap/(\mathbb{N}\backslash(\{0\} \cup \text{KnownJobs}\, \mathit{inf}))
\end{array}$$

$$
\begin{array}{ll}
\& & newq = (\neg\mathrm{EmptyQueue}(q, c) \to q \frown \langle newId\rangle \overset{\leftarrow}{[]}\ q) \\
\& & newc = (\neg\mathrm{EmptyQueue}(q, c) \to c \overset{\leftarrow}{[]}\ (newId, 0)) \\
\& & newInf = (\mathrm{FileOf}\, inf \ominus \{newId \to f\}, \\
 & \qquad\qquad \mathrm{OwnerOf}\, inf \oplus \{newId \mapsto p\}, \\
 & \qquad\qquad \mathrm{PriorityOf}\, inf \oplus \{newId \mapsto n\}) \\
\mathbf{in} & assign(newq, newc, b, newInf, u)) \\
\overset{\leftarrow}{[]}\, assign(\textsc{Unknown\_User\_Error})))
\end{array}
$$

We assume, as in the origial example, that UNKNOWN_USER_ERROR is of type Σ. Unfolding this Add specification will result in (almost) the unfolded specification we already had. The only difference is the empty result () which doesn't appear in the original specification.

4.3.2 The Exception Monad

In an impure functional language, exceptions provide a way to handle errors easily. In a pure language, a similar effect can be achieved by making the result type of a function into a sum type. So, a function will either return a sensible result, or a string representing an error message. However, the code or specification can become complicated since tests must be included to decide whether an input to a function is a value or an error to be propagated. The details of these 'exceptions' can be hidden using the exception monad as described in [89].

We define the type $E\ A$, for arbitrary type A, to be the sum type *Raise String* | *Return A*. A value of this type is either a *String* prefixed by the keyword *Raise* or a value of type A prefixed by the keyword *Return*. The $unit_E$ of the exception monad simply returns the argument,

$$
\begin{array}{lcl}
unit_E & : & A \to E\ A \\
unit_E & = & (\mathbf{fun}\ a \in A : Return\ a)
\end{array}
$$

while $(\star_E)$ tests the result of the first function, passing it on if it is a sensible result and otherwise propagating the error message.

$$
\begin{array}{rcl}
(\star_E) & : & E\ A \to (A \to E\ B) \to E\ B \\
m \star_E k & = & \mathbf{case}\ m\ \mathbf{of} \\
 & & \quad Raise\ e \to Raise\ e \\
 & & \quad Return\ a \to k\ a
\end{array}
$$

The case-expression is used with values of the sum type to test, in this case, whether it is an exception or something from type A.

4.3.3 Combining State and Exceptions

In order to handle both state and exceptions in our printer control example we need to combine the two monads described in sections 4.3.1 and 4.3.2. Unfortunately, there is no automatic method to combine monads. Instead, we build a new monad, exhibiting properties of both [46].

We take as our type of state transformers $ST_S A$, for S the type of states and A an arbitrary type, defined to be the function type $S \rightarrow (\mathit{Raise}\ \mathit{String} \mid \mathit{Return}(A \times S))$. So, a state transformer in $ST_S A$ takes a state and either transforms it, returning a value of type A, or else produces an error.

We find that *unit*, *fetch* and *assign* are almost unchanged from the definitions given in section 4.3.1.

$$\begin{aligned} \mathit{unit} &: A \rightarrow ST_S A \\ \mathit{unit} &= (\textbf{fun}\ a \in A : (\textbf{fun}\ s \in S : \mathit{Return}(a, s))) \end{aligned}$$

$$\begin{aligned} \mathit{fetch} &: ST_S S \\ \mathit{fetch} &= (\textbf{fun}\ s \in S : \mathit{Return}(s, s)) \end{aligned}$$

$$\begin{aligned} \mathit{assign} &: S \rightarrow ST_S() \\ \mathit{assign} &= (\textbf{fun}\ s' \in S : (\textbf{fun}\ s \in S : \mathit{Return}((), s'))) \end{aligned}$$

Only $(\star)$ is changed so that exceptions, if encountered, are propagated.

$$\begin{aligned} (\star) &: ST_S A \rightarrow (A \rightarrow ST_S B) \rightarrow ST_S B \\ m \star k &= (\textbf{fun}\ s \in S : \textbf{case}\ m\ s\ \textbf{of} \\ &\qquad \mathit{Raise}\ e \rightarrow \mathit{Raise}\ e \\ &\qquad \mathit{Return}\ (a, s') \rightarrow k\ a\ s') \end{aligned}$$

We can also define a function *raise*

$$\begin{aligned} \mathit{raise} &: \mathit{String} \rightarrow ST_S() \\ \mathit{raise} &= (\textbf{fun}\ e \in \mathit{String} : (\textbf{fun}\ s \in S : \mathit{Raise}\ e)) \end{aligned}$$

so that $ST_S A$ is like an abstract data type with only these five operations defined for it.

The Printer Control System using the Combined Monad

Using this combined monad for state and exceptions, and with the same assumption that the state type Σ is defined as before, we rewrite the specification for adding a file to the printer queue. The new Add specification has type $\textsc{Person} \times \textsc{File} \times \textsc{Priority} \rightarrow ST_{\Sigma}()$.

$$
\begin{array}{l}
\text{Add} \mathrel{\hat{=}} (\textbf{fun}\ p \in \textsc{Person}, f \in \textsc{File}, n \in \textsc{Priority} : \\
\qquad \mathit{fetch} \star (\textbf{fun}\ (q, c, b, \mathit{inf}, u) \in \Sigma : \\
\qquad\qquad (p \in \text{KnownUsers}\, u \rightarrow \\
\qquad\qquad\qquad \textbf{let}\ \mathit{newId} = [\!]/(\mathbb{N} \backslash (\{0\} \cup \text{KnownJobs}\, \mathit{inf})) \\
\qquad\qquad\qquad \&\ \ \mathit{newq} = (\neg\text{EmptyQueue}(q, c) \rightarrow q \frown \langle \mathit{newId} \rangle\ \overleftarrow{[\!]}\ q) \\
\qquad\qquad\qquad \&\ \ \mathit{newc} = (\neg\text{EmptyQueue}(q, c) \rightarrow c\ \overleftarrow{\|}\ (\mathit{newId}, 0)) \\
\qquad\qquad\qquad \&\ \ \mathit{newInf} = (\text{FileOf}\, \mathit{inf} \oplus \{\mathit{newId} \mapsto f\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{OwnerOf}\, \mathit{inf} \oplus \{\mathit{newId} \mapsto p\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{PriorityOf}\, \mathit{inf} \oplus \{\mathit{newId} \mapsto n\}) \\
\qquad\qquad\qquad \textbf{in}\ \mathit{assign}(\mathit{newq}, \mathit{newc}, b, \mathit{newInf}, u)) \\
\qquad\qquad \overleftarrow{[\!]}\ \mathit{raise}\ \text{“User not known”}))
\end{array}
$$

This looks almost exactly like the last specification we had in section (4.3.1). However, with the new definitions of *fetch*, *assign* and $(\star)$, we now have that both state and errors are being handled correctly. Moreover, the details of handling state and errors are completely hidden in the specification.

4.3.4 Monads in the Specification Language

So far we have used the monads for state and exceptions simply as a structuring device for the printer specification. We are aware that, if the definitions are unfolded, we would get back to a purely functional specification similar to the one of section 4.2. The only difference being that functions which only change the state would also produce an empty result, as highlighted in section 4.3.1. But can we actually define the state/exception monad and associated functions within our specification language, and then include the monad laws in our list of equivalence laws?

In its current form, the specification language does not provide any mechanism to allow user defined types. Instead we have user-defined sets which allow us to define type-like sets, such as Σ. However, we cannot use this method to define the set of 'state transformers with exceptions', because they depend on two types, S the type of states, and A the type of results. Although S is known, in this case, to be the set Σ, A is completely arbitrary. We would have to define a set of 'state transformers with exceptions' for every possible type A, and we have no method for making such families of definitions.

A possible solution might be to anticipate the use of the 'state transformer monad with exceptions' in structuring a certain class of large specifications. In the same way that bags and sequences are defined as data types, it is possible to make $STs\,A$ a data type of the language, dependent on the types S and A. The five operations *unit*, *fetch*, *assign*, *raise* and $(\star)$ also require type rules and axioms to describe their behaviour, including the monad laws. The expression language is rich enough to allow these rules and axioms to be stated.

More generally, it would be useful to allow user defined types, in addition to the enumerated types we have already introduced, which were of the form

$$\mathit{TypeName} \quad ::= \quad v_1 \mid v_2 \mid \ldots \mid v_n$$

As well as defining a type by listing its values, it should be possible to define a type whose values depend on other types. These could be introduced in the form

$$\mathit{TypeName} \quad ::= \quad \mathit{TypeExpression}$$

so that every member of the set *TypeExpression* is now a value of *TypeName*. To a certain extent, we already have this possibility, where user-defined sets are used in type-like situations.

Now we want the ability to define a type using a *TypeExpression* which may be parameterised by type variables. Definitions would be of the form

$$\mathit{TypeName}\ A \quad ::= \quad \mathit{TypeExpression}[A] \tag{4.4}$$

where A is a type variable, or more generally, a list of type variables. We consider such a definition as introducing a family of types, one for each type A. For any type A, the values of type *TypeName* A are identified with the elements of the set given by *TypeExpression*$[A]$, and as such may have associated operations performed on them.

One problem with allowing such user defined types is that the principle of *unicity of types* is destroyed, *i.e.* it is no longer the case that every expression has exactly one type. It is not clear whether, for a type definition of the form (4.4) above, the elements of *TypeName* A and *TypeExpression*$[A]$ should be exactly identified for any A. Operations over elements of *TypeExpression*$[A]$ are now applicable to elements of *TypeName* A, but can functions defined over the new type *TypeName A* be equally applied to elements of *TypeExpression*[*A*]? These are issues which would require to be addressed.

Assuming that such user defined types are permitted, the state monad with exceptions can now be defined for the printer control example, using the specification language, as

$$ST\,A \ ::= \ \Sigma \rightarrow (\mathit{Raise\ String} \mid \mathit{Return}(A \times \Sigma))$$

and the five associated operations, ***unit***, ***fetch***, ***assign***, ***raise*** and $(\star)$ defined as previously. Unfortunately, with this approach the monad laws would require proof.

Comments

The first solution, of anticipating the use of the particular 'state transformer monad with exceptions' has the advantage that the type $ST_S\,A$, for each S and A, is an abstract data type with only the five operations provided. The monad laws, now axioms, can be used for reasoning about specifications. But, while this monad is very useful for the printer control specification, and for other specifications which use a concept of state and require error handling, another class of specification might need something different again. This solution does not offer a generic way of handling the problem.

However, it is reasonable to assume that the class of problems requiring state and error handling is large. Therefore the approach of simply including the 'state transformer monad with exceptions' as a facility built into the language may be considered as practical, without being universal.

The provision of type definitions, which may be parameterised, does provide an extra tool for specification. The state transformer type can now be defined entirely in the language, but so also can types for other monads, or other types useful for a given specification. However, with this approach, the monad laws need to be proved. In addition, the type $ST_S\,A$ is not an abstract type, since the type expression must be made explicit. While this is not necessarily a problem, it would be considerably more elegant to be able to package a type with the allowed operations over that type.

4.4 Conclusions

In this chapter we have tackled the important issue of writing large specifications. As well as the syntax introduced in chapter 2 and the informal specification modules of chapter 3, we have now provided machinery to allow the construction of specifications from partial functions. This facility promotes the *separation of concerns* approach to making specifications. The introduction of the union operations permits these partial functions to be combined to form complete specifications.

In practice, as in the example of the printing control system of section 4.2, we have found that partial functions are a good way to construct large specifications. This specification has been written entirely using the specification language, and partial functions have played an important role in the construction. We also found it useful to introduce some notational conveniences to make the specification more readable and to concentrate on the more important aspects of the problem.

In developing the example, we found a need for methods to control state and error handling in a less explicit manner. Following approaches used in structuring functional programs, we examined, in section 4.3, the use of certain monads in structuring the specification. The resulting specification is more readable, with certain details hidden, but still purely functional.

As the specification language is currently defined, there is no mechanism to allow user defined types, which would permit the definition of a particular monad in a specification. In section 4.3.4, we addressed ways of incorporating the monads for state and exceptions formally into the specification language. Some suggestions were made including approaches to allow user defined types or to add the "state transformer monad with exceptions" explicitly.

We have seen a description of the syntax of the specification language in chapter 2 with some small examples. In this chapter we saw how larger specifications can be made, including a substantial example in section 4.2. The language gives a rich and expressive way to write specifications, but we also require the ability to reason about specifications, and a method of refining such specifications. We now turn our attention to the proof theory of the calculus, describing how properties of specifications can be proved, including refinement properties, which is the subject of chapter 5.

Chapter 5

Proofs and Refinement

In chapter 2, an expression language was defined which includes the usual mathematical expressions associated with integers, booleans, functions, sets, *etc.* , but also incorporates undefined expressions, non-determinism and partiality, which are used for the formulation of expressive and abstract specifications. In chapters 3 and 4 we showed how the expression language may be used to form such specifications.

We have stated that our aim is to provide a refinement calculus for this expression language. This means that we must provide a refinement relation for specifications, *i.e.* define what it means for one specification to refine another; and we must show how such refinements can be calculated.

In this chapter we address a number of aspects of this problem. First we describe what it means to prove a theorem of the language. Already, in chapter 2, we have described the expression language using axioms. In section 5.1 we give an overview of a proof theory based on the axioms for boolean expressions and show how theorems of the language are proved.

In general, in manipulating specifications for either the purpose of refinement or in order to prove a property of a specification, we need to use higher-level theorems than the axioms of chapter 2. So, in section 5.2, a number of such theorems, or transformation laws of the language, are provided.

In section 5.3 we describe what it means for one specification to refine another, and how a program may be calculated from a specification by stepwise and piecewise refinement.

Refinement is handled in our calculus by the introduction of a refinement operator, $\sqsubseteq$ into the language so that, for E and F of the same type, the expression $E \sqsubseteq F$ is a boolean expression. In keeping with the treatment in chapter 2, a number of axioms are provided to govern the behaviour of $\sqsubseteq$.

During the process of stepwise refinement it is not convenient to justify each step by referring to an axiom, so, just as with the transformations of equivalent expressions, a set of refinement laws is provided in section 5.3.2.

It is intended that the axioms, transformation laws and refinement laws together with the proof theory of section 5.1 should allow proofs of refinements and properties of specifications to be calculated quite easily. A number of examples, including reasoning with monads, reasoning about Δ, showing the introduction of recursion into a refinement and the refinement of the N-Queens example are given in section 5.4. The refinement from a simple specification to an imperative style of expression is demonstrated using the example of Bresenham's line drawing algorithm in section 5.5.

5.1 The Proof System

Proofs in the specification language take a different form from that expressed at the end of section 2.3. Instead, equational reasoning, or "substituting equals for equals", is employed.

A proof that an expression P_1 of type *Bool* is a theorem within our system may consist of a sequence of expressions, beginning with P_1 and ending with a known theorem P_n. Each member of the sequence (apart from the first) is obtained from its predecessor P_i by replacing P_i, or a sub-term of P_i, by an equivalent expression.

Such a proof is laid out as follows.

$$\begin{array}{ll}
 & P_1 \\
\equiv & \quad \text{“Reason why } P_1 \equiv P_2\text{”} \\
 & P_2 \\
\equiv & \quad \text{“Reason why } P_2 \equiv P_3\text{”} \\
 & \vdots \\
\equiv & \quad \text{“Reason why } P_{n-1} \equiv P_n\text{”} \\
 & P_n
\end{array}$$

By transitivity of $\equiv$ we conclude that $P_1 \equiv P_n$. Since P_n is a theorem, it follows that so also is P_1. If any of the $\equiv$ signs in the left column is replaced by $\Leftarrow$, with a corresponding justification, then by transitivity of implication we have a proof of $P_n \Rightarrow P_1$. Again, since P_n is a theorem, it follows from modus ponens that P_1 also holds.

A proof of a boolean expression of the form $E \equiv F$, for E and F of some type T, may proceed as above, or may consist of a sequence of expressions of type T beginning with E and ending with F. Again, each expression in the sequence is obtained from its predecessor by replacement of equivalent subexpressions.

A justification that equational reasoning is valid within our system, along with various strategies for proof, may be found in [64].

When manipulating specifications with detached parameters, such as the printer control specification of chapter 4, it should be clear that we are actually just manipulating function bodies. The detached parameters can always be eliminated. However, for convenience, we will write $\sigma : \Sigma \vdash E \equiv \sigma : \Sigma \vdash F$, to mean $(\mathbf{fun}\ \sigma \in \Sigma : E) \equiv (\mathbf{fun}\ \sigma \in \Sigma : F)$.

5.2 Transformation Laws

In order to provide a simple calculus for the easy manipulation and transformation of expressions as specifications, we need to provide some transformation laws which are easily applicable to specifications. The axioms of chapter 2 form a base for such laws, but it is not usually convenient to manipulate specifications from first principles. Some higher order theorems are required. In the following list of laws we assume the following conventions: E, E_1, E_2, F and G are any expressions, subject to appropriate syntax constraints; f is a function expression; P and Q are expressions of type *Bool*; v is a value; and S is a set expression.

We include three laws concerning **let** expressions. These laws can be proved by unfolding the meaning of local expressions as given in section 2.5.2, however it is useful in proofs involving long expressions to apply these in one step.

Law (Distribution of Function Application inside let Expressions) *If x does not occur free in f, but may be free in F then*

$$f(\mathbf{let}\ x = E\ \mathbf{in}\ F) \quad \equiv \quad \mathbf{let}\ x = E\ \mathbf{in}\ f\,F$$

Law (Swapping Local Definitions) *Suppose that x_1 and x_2 may be free in F. If x_1 does not occur free in E_2 and x_2 does not occur free in E_1 then*

$$\mathbf{let}\ x_1 = E_1\ \&\ x_2 = E_2\ \mathbf{in}\ F \quad \equiv \quad \mathbf{let}\ x_2 = E_2\ \&\ x_1 = E_1\ \mathbf{in}\ F$$

Law (No Occurrence of Local Definition) *If x does not occur free in F then*

$$\delta\, E \Rightarrow ((\textbf{let } x = E \textbf{ in } F) = F)$$

For deterministic functions, we have a form of γ-reduction.

Law (γ-Reduction)

$$(\textbf{fun } x \in T : E)x \equiv E$$

We include a law concerning the behaviour of generalised choice with assumptions, as permitted in section 2.6.2.

Law (Properties of Generalised Choice)

$$\Delta\, S \Rightarrow ((S \neq \emptyset) \succ\!\!- [\!]/S \equiv [\!]/S \,[\!]\, \bot)$$

Information contained in the guard or assumption can be used to manipulate an expression.

Law (Using Context in Assumptions and Guards)

$$(P \Rightarrow (E \equiv E')) \Rightarrow (P \rightarrowtail E \equiv P \rightarrowtail E')$$

where '$\rightarrowtail$' represents either '$\rightarrow$' or '$\succ\!\!-$' throughout the formula.

Choice and guarding together permit the formation of alternation expressions. These can be introduced into a derivation using the following law.

Law (Alternation Introduction)

$$\Delta\, P \Rightarrow (E \equiv P \rightarrow E \,[\!]\, \neg P \rightarrow E)$$

More generally, for P_i, $1 \leqslant i \leqslant n$, boolean expressions

$$((\forall\, i : \mathbb{Z} \mid 1 \leqslant i \leqslant n \bullet \Delta\, P_i) \wedge (\exists\, i : \mathbb{Z} \mid 1 \leqslant i \leqslant n \bullet P_i))$$
$$\Rightarrow (E \equiv P_1 \rightarrow E \,[\!]\, \ldots \,[\!]\, P_n \rightarrow E)$$

We saw in section 2.6 that there is a relationship between conditionals and alternations.

Law (Alternation to Conditional)

$$\Delta P \Rightarrow (P \rightarrow E \mathbin{[\!]} \neg P \rightarrow F \equiv P \rightarrow E \mathbin{\overset{\leftarrow}{[\!]}} F)$$
$$(P \rightarrow E) \mathbin{\overset{\leftarrow}{[\!]}} F \equiv \textbf{if } P \textbf{ then } E \textbf{ else } F$$

Assumptions and guards both distribute over choice to the right.

Law (Distribution of Assumptions and Guards over Choice)

$$P \rightarrowtail (E \mathbin{[\!]} F) \equiv (P \rightarrowtail E) \mathbin{[\!]} (P \rightarrowtail F)$$

where '$\rightarrowtail$' represents either '$\hookrightarrow$' or '$\succ\!\!-$' throughout the formula.

Guarded expressions, being potentially partial, are restricted in where they may occur. However, expressions with assumptions are total, and so there are laws determining how assumptions distribute over various operations

Law (Distribution of Assumptions through Product Formation)

$$(P \succ\!\!- E, Q \succ\!\!- E') \equiv P \wedge Q \succ\!\!- (E, E')$$

Law (Distribution of Assumptions through Function Application)

$$(P \succ\!\!- f)(Q \succ\!\!- E) \equiv P \wedge Q \succ\!\!- f\ E$$

Other laws concerning the behaviour of assumptions include

Law (Double Assumptions)

$$P \succ\!\!- (Q \succ\!\!- E) \equiv P \wedge Q \succ\!\!- E$$
$$(P \succ\!\!- Q) \succ\!\!- E \equiv P \wedge Q \succ\!\!- E$$

There is also a law for expressions with both a guard and an assumption.

Law (Propagate Guard as Assumption)

$$(P \Rightarrow Q) \Rightarrow (P \rightarrow E \equiv P \rightarrow (Q \succ\!\!- E))$$

An expression with an assumption may be guarded, but the syntactic restrictions of section 2.6.2 do not allow expressions of the form $P \succ\!\!- (P' \rightarrow E)$.

We saw, in section 2.6.2, how it is possible to recognise syntactically expressions which are potentially partial. Such expressions need to be totalised, but all other expressions are already total. The next law gives the condition under which a totaliser can be removed.

Law (Removal of Totaliser) *If E is not potentially partial then*

$$E \overleftarrow{[\,]} F \equiv E$$

The following two laws are concerned with distributive properties of choice with biased choice.

Law (Right-Distribution of Biased Choice over Choice)

$$E \overleftarrow{[\,]} (F \,[\,]\, G) \equiv (E \overleftarrow{[\,]} F) \,[\,]\, (E \overleftarrow{[\,]} G)$$

Law (Choice with Biased Choice)

$$\begin{aligned} E \overleftarrow{[\,]} (E \,[\,]\, F) &\equiv E \overleftarrow{[\,]} F \\ E \,[\,]\, (E \overleftarrow{[\,]} F) &\equiv E \overleftarrow{[\,]} F \\ E \,[\,]\, (F \overleftarrow{[\,]} E) &\equiv E \,[\,]\, F \end{aligned}$$

5.3 Refinement

Given a specification S of the expression language, the ultimate goal is to find a specification P which is executable and which satisfies S, i.e. P implements S. An executable specification P is made up of expressions from that part of the specification language which forms the programming sub-language (see section 2.1). As such, it must be defined and deterministic. Since the original specification S may exhibit either of the properties of undefinedness or non-determinism, it follows that equivalence does not hold between S and P. Instead we need a refinement relation, $\sqsubseteq$, so that P *refines* S, $S \sqsubseteq P$.

Informally, for an expression E to be refined by an expression F, written $E \sqsubseteq F$, it should be the case that every possible 'evaluation' of F is also a possible 'evaluation' of E, or is better defined than some possible 'evaluation' of E. So, a specification is refined by reducing non-determinism or by increasing definedness. For example, we expect the following refinements to hold

$$\begin{aligned} 2 \mathbin{[\!]} 3 &\sqsubseteq 2 && (5.1) \\ [\!]/\mathbb{N} &\sqsubseteq 2 \mathbin{[\!]} 3 \\ (\textbf{fun } x \in \mathbb{Z} : x+2 \mathbin{[\!]} x+3) &\sqsubseteq (\textbf{fun } x \in \mathbb{Z} : x+2) \\ (\textbf{fun } x \in \mathbb{N} : x+2 \mathbin{[\!]} x+3) &\sqsubseteq (\textbf{fun } x \in \mathbb{Z} : x+2) && (5.2) \end{aligned}$$

The first two examples are simple cases of reducing non-determinacy, while the third example reduces non-determinacy within the body of a function. The last example reduces non-determinacy, but also increases definedness since the function on the left gives an undefined result for any negative integer, while that on the right is defined for every integer.

We advocate the process of program development by stepwise refinement, starting with an initial specification S_0 and building a sequence of specifications $S_0 \sqsubseteq S_1 \sqsubseteq \ldots \sqsubseteq S_n$ so that each S_i, for $1 \leqslant i \leqslant n$ is an acceptable replacement for S_{i-1}, and S_n is a program. Since the aim is to derive programs in steps, it is required that the refinement relation is transitive. Then, from a sequence of refinements of the form $S_0 \sqsubseteq S_1 \sqsubseteq \ldots \sqsubseteq S_n$, we can conclude that S_n is a correct implementation of the initial specification S_0. In fact, refinement is a pre-order, since every specification refines itself. In general, a refinement relation need not be anti-symmetric. In fact our relation is not since, for example, we have the refinements

$$\begin{aligned} 2 \mathbin{[\!]} \bot &\sqsubseteq \bot \\ \bot &\sqsubseteq 2 \mathbin{[\!]} \bot \end{aligned}$$

In the first case, the refinement is obtained by reducing non-determinism, while in the second definedness is increased. However, the two expressions are not equivalent.

As well as refinements proceeding stepwise, it is also important that refinement can occur piecewise. This means that an expression may be refined by refining one, or more, subexpressions,

$$(E \sqsubseteq F) \Rightarrow (G[E/x] \sqsubseteq G[F/x])$$

This states exactly the property that G must be monotonic (with respect to refinement) at the position x where the refined subexpression occurs. Refinement can occur only in monotonic positions.

Most of the constructs of the expression language, as defined in chapter 2, are monotonic. But there is a small number of operators which are non-monotonic. These include equivalence $\equiv$, non-equivalence $\not\equiv$, and the two delta operators Δ and δ. Implication $\Rightarrow$ and biased choice $\overset{\leftarrow}{[]}$ are non-monotonic in the first argument, and monotonic in the second. Function abstraction is monotonic only when the abstraction is over a monotonic position.

Subexpressions which occur in non-monotonic positions may be replaced only by equivalent expressions. This means that some care must be taken when refining expressions with non-monotonic elements, but in practice this is not a problem.

We now introduce the refinement relation as an operator of the language:

$$\frac{E : T \quad F : T}{E \sqsubseteq F : \mathit{Bool}}$$

An expression of the form $E \sqsubseteq F$ is always proper, and it should be clear that refinement does not distribute over choice.

The following axioms describe refinement of expressions.

The refinement relation is transitive

$$(E \sqsubseteq F) \wedge (F \sqsubseteq G) \Rightarrow (E \sqsubseteq G)$$

The general refinement axiom is

$$(E \sqsubseteq F) \Leftarrow (\neg\delta\, E \vee (E \,[]\, F \equiv E))$$

When E and F belong to a simple type, this is an equivalence, and may be used as the definition of refinement.

For function domains, with Δf and $\Delta\, g$,

$$(f \sqsubseteq g) \equiv (\forall x : T \mid \bullet f\, x \sqsubseteq g\, x)$$

When refining non-deterministic expressions, with $\Delta\, G$, $\Delta\, E$ and $\Delta\, S$ we have the axioms

$$(E \,[]\, F \sqsubseteq G) \equiv (E \sqsubseteq G \vee F \sqsubseteq G)$$
$$([]/S \sqsubseteq E) \equiv (\exists x : T \mid x \in S \bullet x \sqsubseteq E)$$

We assert that top $\top$ is the unique most-refined specification,

$$(\top \sqsubseteq E) \equiv (E \equiv \top)$$

We define the concept of *refinement equivalence* for expressions which refine each other,

$$E \sqsupseteq\!\!\!\sqsubseteq F \quad \hat{=} \quad E \sqsubseteq F \land E \sqsupseteq F$$

where $E \sqsupseteq F \mathrel{\hat{=}} F \sqsubseteq E$. Clearly refinement equivalence is weaker than $\equiv$.

5.3.1 Proving Refinements

Refinements proceed stepwise, as previously indicated, with a similar layout to transformation proofs as in section 5.1.

There are two additional inference rules to accommodate refinement:

$$\frac{E \sqsubseteq F \quad F \sqsubseteq G}{E \sqsubseteq G}$$

for the transitivity of refinement, and

$$\frac{E \sqsubseteq F}{G[E/x] \sqsubseteq G[F/x]}$$

where x is in a monotonic position in G.

A refinement then proceeds as a sequence of specifications, starting with the initial specification, expression E_1.

$$\begin{array}{ll}
 & E_1 \\
\sqsubseteq & \quad \text{“Reason why } E_1 \sqsubseteq E_2\text{”} \\
 & E_2 \\
\sqsubseteq & \quad \text{“Reason why } E_2 \sqsubseteq E_3\text{”} \\
 & \vdots \\
\sqsubseteq & \quad \text{“Reason why } E_{n-1} \sqsubseteq E_n\text{”} \\
 & E_n
\end{array}$$

and we may conclude that $E_1 \sqsubseteq E_n$. In the above, any of the $\sqsubseteq$ in the left margin may be replaced by equivalence $\equiv$.

5.3.2 Refinement Laws

Given a specification, refinement will proceed stepwise, as indicated above, using the inference rules and axioms for refinement. But, in general, it is not convenient to calculate each refinement from first principles. As in the case for simple transformations, a collection of theorems, or refinement laws, is required. This is what we now provide.

In the following list of refinement laws we assume the following conventions: E, F, F_1, F_2 and G are any expressions, subject to appropriate syntax constraints; P and Q are expressions of type *Bool*; v is a value; and S and S' are set expressions.

The first law says that an expression may be refined by reducing non-determinacy. This could take a number of forms.

Law (Reduce Non-Determinacy)

$$E \,[]\, F \sqsubseteq E$$

For generalised choice,

$$(S' \subseteq S) \Rightarrow ([]/S \sqsubseteq []/S')$$
$$(\forall x \in T \mid \bullet Q \Rightarrow P) \Rightarrow ([]/\{x \in T \,:\, P\} \sqsubseteq []/\{x \in T \,:\, Q\})$$
$$(v \in S) \Rightarrow ([]/S \sqsubseteq v)$$

Choice can also be introduced into a specification, but note that this does not increase non-determinacy.

Law (Introduce Choice)

$$(E \sqsubseteq F_1 \wedge E \sqsubseteq F_2) \Rightarrow (E \sqsubseteq F_1 \,[]\, F_2)$$

An expression of the form $P \succ\!\!- E$ may be refined by refining E or by weakening P.

Law (Weaken Assumption)

$$(P \Rightarrow Q) \Rightarrow (P \succ\!\!- E \sqsubseteq Q \succ\!\!- E)$$

By weakening the assumption to *True*, and so effectively removing it, the next law immediately follows.

Law (Remove Assumption)

$$P \mathrel{>\!\!-} E \sqsubseteq E$$

Dually, an expression of the form $P \rightarrow E$ may be refined by refining E or by strengthening P.

Law (Strengthen Guard)

$$(\delta\, Q \wedge Q \Rightarrow P) \Rightarrow (P \rightarrow E \sqsubseteq Q \rightarrow E)$$

Now any expression, which may be considered to have an implicit *True* guard, is refined by introducing a guard.

Law (Introduce Guard)

$$\delta\, P \Rightarrow (E \sqsubseteq P \rightarrow E)$$

Care must be taken when applying the previous two laws above since the refined expression can be considered *more partial*. In particular, in the second case, a potentially partial expression is introduced instead of a total expression.

A useful law allows the use of information in a guard or assumption to refine an expression.

Law (Using Context in Assumptions and Guards)

$$(P \Rightarrow E \sqsubset E') \Rightarrow (P \rightarrowtail E \sqsubseteq P \rightarrowtail E')$$

where '$\rightarrowtail$' represents either '$\rightarrow$' or '$\mathrel{>\!\!-}$' throughout the formula.

Non-determinacy can be reduced by taking the conjunction or disjunction of assumptions or guards, as governed by the following laws.

Law (Disjunction and Conjunction of Assumptions)

$$P \mathrel{>\!\!-} E \;[]\; Q \mathrel{>\!\!-} E \sqsubseteq P \vee Q \mathrel{>\!\!-} E$$
$$P \mathrel{>\!\!-} E \;[]\; Q \mathrel{>\!\!-} E \sqsupseteq\!\sqsubseteq P \wedge Q \mathrel{>\!\!-} E$$

Note the mutual refinement of the second clause.

Law (Disjunction and Conjunction of Guards)

$$P \to E \mathbin{[\!]} Q \to E \sqsubseteq P \lor Q \to E$$
$$P \to E \mathbin{[\!]} Q \to E \sqsubseteq P \land Q \to E$$

Law (Refine Function) *If x appears in a monotonic position in E, F and P,*

$$(\forall x : T \bullet E \sqsubseteq F) \Rightarrow ((\textbf{fun } x \in T : E) \sqsubseteq (\textbf{fun } x \in T : F))$$
$$(\forall x : T \bullet \delta P) \Rightarrow ((\textbf{fun } x \in T : E \mathbin{[\!]} F) \sqsubseteq (\textbf{fun } x \in T : P \to E \overleftarrow{[\!]} F))$$

To complement the equivalence law concerning right-distribution of biased choice over choice, we have the refinement laws

Law (Distribution between Choice and Biased Choice)

$$(E \mathbin{[\!]} F) \overleftarrow{[\!]} G \sqsupseteq (E \overleftarrow{[\!]} G) \mathbin{[\!]} (F \overleftarrow{[\!]} G)$$
$$E \mathbin{[\!]} (F \overleftarrow{[\!]} G) \sqsubseteq (E \mathbin{[\!]} F) \overleftarrow{[\!]} (E \mathbin{[\!]} G)$$

Finally, we give the law governing the introduction of recursion into a specification.

Law (Recursion Introduction) *Let E_x be an expression which contains a free occurrence of the variable x, and let E_x^y be the same expression but with value y substituted for x.*

$$(E_x \sqsubseteq F[(\textbf{fun } y \in T : y < x \succ\!\!- E_x^y)]) \Rightarrow (E_x \sqsubseteq \textbf{let } f = (\textbf{fun } x \in T : F[f]) \textbf{ in } f\ x)$$

where T is a well-founded set with respect to $<$, and $F[X]$ is monotonic with respect to refinement of subexpression X.

Proof We use the deduction theorem, and prove the consequent by assuming the antecedant. So, we assume

$$(E_x \sqsubseteq F[(\textbf{fun } y \in T : y < x \succ\!\!- E_x^y)])$$

Since T is well-founded we can use the principle of induction for well-founded sets,

$$((\forall x \in C : P\ x) \Leftarrow (\forall y \in C : y < x : P\ y)) \quad \equiv \quad (\forall x \in C : P\ x)$$

where $<$ is a well-founded ordering for C, and P is some property over elements of C. So, we take as our induction hypothesis:

$$(\forall\, y \in T : y < x : E_x^y \sqsubseteq \textbf{let } f = (\textbf{fun } x \in T : F[f]) \textbf{ in } f\ y) \tag{5.3}$$

The proof proceeds as follows. Let $x \in T$.

$E_x \sqsubseteq \textbf{let } f = (\textbf{fun } x \in T : F[f]) \textbf{ in } f\ x$

$\equiv$ "For convenience, detach $f = (\textbf{fun } x \in T : F[f])$"

$E_x \sqsubseteq f\ x$

$\equiv$ "Unfolding f"

$E_x \sqsubseteq (\textbf{fun } x \in T : F[f])x$

$\equiv$ "γ-reduction"

$E_x \sqsubseteq F[f]$

$\Leftarrow$ "Using the assumption, $\sqsubseteq$ is transitive"

$F[(\textbf{fun } y \in T : y < x \succ\!\!- E_x^y)] \sqsubseteq F[f]$

$\Leftarrow$ "$F[X]$ is monotonic with respect to refinement of subexpression X"

$(\textbf{fun } v \in T : v < x \succ\!\!- E_x^v) \sqsubseteq f$

$\equiv$ "Refinement axiom for proper functions, and substitution"

$(\forall\, y \in T : y < x \succ\!\!- E_x^y \sqsubseteq f\ y)$

$\Leftarrow$ "Axioms for assumptions, $\perp$ least wrt refinement"

$(\forall\, y \in T : y < x : E_x^y \sqsubseteq f\ y)$

$\equiv$ "Induction Hypothesis"

True

The recursion introduction law now follows by induction and the deduction theorem.

The refinement laws all follow quite easily from the axioms for refinement. The laws concerning assumptions and guards may be proved by a case analysis on the value of the assumption/guard. The laws for biased choice are proved by case analysis on the totality of the left argument.

5.4 Examples of Formal Reasoning

In this section we demonstrate the sort of proofs which may be formed using the axioms and laws of the language, and the properties of $\equiv$ and $\sqsubseteq$. These proofs range in complexity from simplification of expressions to proving properties of specifications and the introduction of recursion during the refinement of expressions.

5.4.1 Simple Proofs

We look at some simple reasoning about specifications by simple manipulation of expressions. For example, to illustrate some of the distributive properties of choice, a function applied to a non-deterministic argument is simplified. Note that, although the function has a non-deterministic body, it is itself deterministic.

$(\mathbf{fun}\ x \in \mathbb{Z} : x \mathbin{[\!]} x+1)(3 \mathbin{[\!]} 4)$

$\equiv$ **"Distribute Function Application over Choice"**

$(\mathbf{fun}\ x \in \mathbb{Z} : x \mathbin{[\!]} x+1)\, 3 \mathbin{[\!]} (\mathbf{fun}\ x \in \mathbb{Z} : x \mathbin{[\!]} x+1)\, 4$

$\equiv$ **"Substitution, $\Delta\, 3$ and $\Delta\, 4$"**

$(3 \mathbin{[\!]} 3+1) \mathbin{[\!]} (4 \mathbin{[\!]} 4+1)$

$\equiv$ **"Axioms for Integers"**

$(3 \mathbin{[\!]} 4) \mathbin{[\!]} (4 \mathbin{[\!]} 5)$

$\equiv$ **"Properties of Choice"**

$3 \mathbin{[\!]} 4 \mathbin{[\!]} 5$

From a brief example of section 2.6, illustrating the behaviour of guards and totalisers, the following function application is simplified.

$(\mathbf{fun}\ x \in \mathbb{Z} : \mathbf{if}\ x \geqslant 0 \to \text{'+'} \mathbin{[\!]} x \leqslant 0 \to \text{'−'}\ \mathbf{fi})\, 0$

$\equiv$ **"Substitution, $\Delta\, 0$"**

$\mathbf{if}\ 0 \geqslant 0 \to \text{'+'} \mathbin{[\!]} 0 \leqslant 0 \to \text{'−'}\ \mathbf{fi}$

$\equiv$ **"Axioms for $\geqslant$"**

$\mathbf{if}\ \mathit{True} \to \text{'+'} \mathbin{[\!]} \mathit{True} \to \text{'−'}\ \mathbf{fi}$

$\equiv$ **"Axioms for Guarding"**

$\mathbf{if}\ \text{'+'} \mathbin{[\!]} \text{'−'}\ \mathbf{fi}$

$\equiv$ "Definition of **if** ... **fi**"

$(\text{'+'} \mathbin{[\!]} \text{'−'}) \overset{\leftarrow}{[\!]} \perp$

$\equiv$ **"Removal of Totaliser"**

$\text{'+'} \mathbin{[\!]} \text{'−'}$

Returning to the Multiplication Example of section 3.3 we simplify

$\text{Multiply}(\langle 4\rangle \mathbin{[\!]} \langle 8\rangle, \langle 2,5\rangle)$

$\equiv$ **"Distribute Product Formation over Choice"**

$\text{Multiply}((\langle 4\rangle, \langle 2,5\rangle) \mathbin{[\!]} (\langle 8\rangle, \langle 2,5\rangle))$

$\equiv$ **"Distribute Function Application over Choice"**

Multiply($\langle 4\rangle, \langle 2,5\rangle$) $[\!]$ Multiply($\langle 8\rangle, \langle 2,5\rangle$)

$\equiv$ "Definition of Multiply, **Substitution** with proper arguments"

$[\!]/\{z \in \text{Number} : \text{Convert}\, z = \text{Convert}\langle 4\rangle * \text{Convert}\,\langle 2,5\rangle\}$

$[\!]\ [\!]/\{z \in \text{Number} : \text{Convert}\, z = \text{Convert}\langle 8\rangle * \text{Convert}\,\langle 2,5\rangle\}$

$\equiv$ "Definition of Convert, **Substitution** with proper terms"

$[\!]/\{z \in \text{Number} : \text{Convert}\, z = 4 * 25\}\ [\!]\ [\!]/\{z \in \text{Number} : \text{Convert}\, z = 8 * 25\}$

$\equiv$ **"Axioms for Evaluations of Sets"**

$[\!]/\{\langle 1,0,0\rangle\}\ [\!]\ [\!]/\{\langle 2,0,0\rangle\}$

$\equiv$ **"Properties of $[\!]/$"**

$\langle 1,0,0\rangle\ [\!]\ \langle 2,0,0\rangle$

These examples illustrate the use of some of the equivalence laws with small specifications. In chapter 4 we saw how the expression language could be used to build bigger specifications. It is important that the equivalence laws can be used to prove properties about large specifications also.

5.4.2 A Larger Example

In section 4.2, a purely functional specification of a printing control system was detailed. We now show how the equivalence laws can be used to reason about this specification.

First we prove an easy equivalence stated in section 4.2. The function Alloc was defined using two partial functions, NewUser and ResetQuota, in such a way that

$$\sigma : \Sigma \vdash \text{Alloc} \mathrel{\hat{=}} \text{NewUser} \mathbin{\dot{\cup}} \text{ResetQuota}$$

NewUser and ResetQuota were defined as

$$\begin{aligned}
\sigma : \Sigma \vdash \text{NewUser} &\mathrel{\hat{=}} (\textbf{fun}\ p \in \textsc{Person}, q \in \mathbb{N} : \\
&\qquad p \notin \text{KnownUsers} \rightarrow E) \\
\sigma : \Sigma \vdash \text{ResetQuota} &\mathrel{\hat{=}} (\textbf{fun}\ p \in \textsc{Person}, q \in \mathbb{N} : \\
&\qquad p \in \text{KnownUsers} \rightarrow E)
\end{aligned}$$

where E is a shorthand for the more complicated expression given in the specification. The details of E are not required in the following proof however. We said that the function Alloc, as defined above, is equivalent to the specification

$\sigma : \Sigma \vdash \text{Alloc} \mathrel{\hat{=}} (\textbf{fun}\ p \in \textsc{Person}, q \in \mathbb{N} : E)$

We reason that

$$\begin{array}{ll}
 & \sigma : \Sigma \vdash \text{Alloc} \\
\equiv & \quad \text{“Definition of Alloc”} \\
 & \sigma : \Sigma \vdash \text{NewUser} \mathbin{\dot{\cup}} \text{ResetQuota} \\
\equiv & \quad \text{“Definitions of NewUser, ResetQuota and } \dot{\cup} \text{”} \\
 & \sigma : \Sigma \vdash (\textbf{fun}\ p \in \textsc{Person}, q \in \mathbb{N} : \\
 & \qquad p \notin \text{KnownUsers} \to E \mathbin{[]} p \in \text{KnownUsers} \to E) \\
\equiv & \quad \text{“}\textbf{Alternation Introduction}, \Delta(p \in \text{KnownUsers})\text{”} \\
 & \sigma : \Sigma \vdash (\textbf{fun}\ p \in \textsc{Person}, q \in \mathbb{N} : E)
\end{array}$$

as required. The last step of this proof assumes that $p \in \text{KnownUsers}$ is a proper boolean expression for any state σ, which is reasonable.

The above is a proof that two specifications are equivalent. We now give an example of a proof that the specification satisfies a certain property. Again, we use the equivalence laws as tools for reasoning.

Let $\sigma \mathrel{\hat{=}} (q, c, b, inf, u)$ be any state such that $(\neg\text{EmptyQueue}\, q\, c)$. Let p be a $\textsc{Person}$ such that $(p \in \text{KnownUsers}\, u)$, then it should be the case that

$$\text{Active}(\text{Add}(\sigma, p, f, n)) \quad \equiv \quad \text{Active}\, \sigma$$

Rather than tackle the whole expression at once, each side of the equation is simplified in turn. Using the definition of Active, and the **Substitution** law, the expression on the right, $(\text{Active}\, \sigma)$, is equivalent to

$$\begin{array}{l}
\neg\text{EmptyQueue}\, q\, c \to \textbf{let}\ id = \text{CurrentId}\, c \parallel n = \text{PagesPrinted}\, c \\
\qquad\qquad\qquad\qquad\quad \&\ \ size = \text{SizeOf}\, id \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ (id, n, size - n) \\
\overset{\leftarrow}{[]}\ \textsc{Queue\_Empty\_Error}
\end{array}$$

From the given fact that the queue is not empty, the guard becomes *True*. Using the **Axioms for Guarding** and **Removing the Totaliser**, the expression becomes

$$\begin{array}{l}
\textbf{let}\ id = \text{CurrentId}\, c \parallel n = \text{PagesPrinted}\, c \\
\&\ \ size = \text{SizeOf}\, id \\
\textbf{in}\ (id, n, size - n)
\end{array}$$

This is the simplest expression which can be obtained, without unfolding the **let** expression.

Now taking the expression on the left of the proposed equivalence, (Active(Add(σ, p, f, n))), the definition of Add is used, followed by the **Substitution** axiom, which gives

$$
\begin{array}{l}
\text{Active}\,(p \in \text{KnownUsers}\,u \rightarrow \\
\qquad\qquad\qquad (\textbf{let}\ \mathit{newId} = [\!]/(\mathbb{N}\backslash(\{0\} \cup \text{KnownJobs}\,\mathit{inf})) \\
\qquad\qquad\qquad \&\quad \mathit{newq} = (\neg\text{EmptyQueue}\,q\,c \rightarrow q \frown \langle \mathit{newId}\rangle \overset{\leftarrow}{[\!]}\, q) \\
\qquad\qquad\qquad \&\quad \mathit{newc} = (\neg\text{EmptyQueue}\,q\,c \rightarrow c \overset{\leftarrow}{[\!]}\, (\mathit{newId}, 0)) \\
\qquad\qquad\qquad \&\quad \mathit{newInf} = (\text{FileOf}\,\mathit{inf} \oplus \{\mathit{newId} \mapsto f\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{OwnerOf}\,\mathit{inf} \oplus \{\mathit{newId} \mapsto p\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{PriorityOf}\,\mathit{inf} \oplus \{\mathit{newId} \mapsto n\}) \\
\qquad\qquad\qquad \textbf{in}\ (\mathit{newq}, \mathit{newc}, b, \mathit{newInf}, u)) \\
\quad \overset{\leftarrow}{[\!]}\ \text{UNKNOWN\_USER\_ERROR})
\end{array}
$$

We use the given facts that ($p \in \text{KnownUsers}\,u$) and that the queue is not empty. The main guard becomes *True*, as well as the two inner guards. So, using the **Axioms for Guarding and Removing the Totaliser**, the above expression becomes

$$
\begin{array}{l}
\text{Active}\ (\textbf{let}\ \mathit{newId} = [\!]/(\mathbb{N}\backslash(\{0\} \cup \text{KnownJobs}\,\mathit{inf})) \\
\qquad \&\quad \mathit{newq} = q \frown \langle \mathit{newId}\rangle \\
\qquad \&\quad \mathit{newc} = c \\
\qquad \&\quad \mathit{newInf} = (\text{FileOf}\,\mathit{inf} \oplus \{\mathit{newId} \mapsto f\}, \\
\qquad\qquad\qquad\qquad \text{OwnerOf}\,\mathit{inf} \oplus \{\mathit{newId} \mapsto p\}, \\
\qquad\qquad\qquad\qquad \text{PriorityOf}\,\mathit{inf} \oplus \{\mathit{newId} \mapsto n\}) \\
\qquad \textbf{in}\ (\mathit{newq}, \mathit{newc}, b, \mathit{newInf}, u))
\end{array}
$$

In order to use the definition of the Active function, it is easier to move it inside the **let** expression, using the **Distribution of Function Application inside let Expressions** law. This results in

$$
\begin{array}{l}
\textbf{let}\ \mathit{newId} = [\!]/(\mathbb{N}\backslash(\{0\} \cup \text{KnownJobs}\,\mathit{inf})) \\
\&\quad \mathit{newq} = q \frown \langle \mathit{newId}\rangle \\
\&\quad \mathit{newc} = c \\
\&\quad \mathit{newInf} = (\text{FileOf}\,\mathit{inf} \oplus \{\mathit{newId} \mapsto f\}, \\
\qquad\qquad\qquad \text{OwnerOf}\,\mathit{inf} \oplus \{\mathit{newId} \mapsto p\}, \\
\qquad\qquad\qquad \text{PriorityOf}\,\mathit{inf} \oplus \{\mathit{newId} \mapsto n\}) \\
\textbf{in}\ \text{Active}\,(\mathit{newq}, \mathit{newc}, b, \mathit{newInf}, u)
\end{array}
$$

The definition of Active is used next, and the **Substitution** axiom is applied again.

$$\begin{array}{ll}
\textbf{let} & \mathit{newId} = [\!]/(\mathbb{N}\backslash(\{0\} \cup \mathrm{KnownJobs}\,\mathit{inf})) \\
\& & \mathit{newq} = q \frown \langle \mathit{newId} \rangle \\
\& & \mathit{newc} = c \\
\& & \mathit{newInf} = (\mathrm{FileOf}\,\mathit{inf} \oplus \{\mathit{newId} \mapsto f\}, \\
 & \qquad\qquad \mathrm{OwnerOf}\,\mathit{inf} \oplus \{\mathit{newId} \mapsto p\}, \\
 & \qquad\qquad \mathrm{PriorityOf}\,\mathit{inf} \oplus \{\mathit{newId} \mapsto n\}) \\
\multicolumn{2}{l}{\textbf{in } (\neg \mathrm{EmptyQueue}\,\mathit{newq}\,\mathit{newc} \rightarrow \textbf{let } \mathit{id} = \mathrm{CurrentId}\,\mathit{newc} \parallel n = \mathrm{PagesPrinted}\,\mathit{newc}} \\
\multicolumn{2}{l}{\qquad\qquad\qquad\qquad\qquad\qquad\quad \& \quad \mathit{size} = \mathrm{SizeOf}\,\mathit{id}} \\
\multicolumn{2}{l}{\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in } (\mathit{id}, n, \mathit{size} - n)} \\
\multicolumn{2}{l}{\quad \overset{\leftarrow}{[\!]}\ \textsc{Queue\_Empty\_Error})}
\end{array}$$

Using the law for **Swapping Local Definitions,** the local definition for *newc* can be unfolded and substituted into the specification. The guard becomes (EmptyQueue *newq c*) which, according to the definition of EmptyQueue, is equivalent to (CurrentId $c \neq 0$). This is *True*, since we have assumed (EmptyQueue *q c*). So, using the **Axioms for Guarding and Removing the Totaliser** again, the expression becomes

$$\begin{array}{ll}
\textbf{let} & \mathit{newId} = [\!]/(\mathbb{N}\backslash(\{0\} \cup \mathrm{KnownJobs}\,\mathit{inf})) \\
\& & \mathit{newq} = q \frown \langle \mathit{newId} \rangle \\
\& & \mathit{newInf} = (\mathrm{FileOf}\,\mathit{inf} \oplus \{\mathit{newId} \mapsto f\}, \\
 & \qquad\qquad \mathrm{OwnerOf}\,\mathit{inf} \oplus \{\mathit{newId} \mapsto p\}, \\
 & \qquad\qquad \mathrm{PriorityOf}\,\mathit{inf} \oplus \{\mathit{newId} \mapsto n\}) \\
\multicolumn{2}{l}{\textbf{in } (\textbf{let } \mathit{id} = \mathrm{CurrentId}\,c \parallel n = \mathrm{PagesPrinted}\,c} \\
\multicolumn{2}{l}{\quad \& \quad \mathit{size} = \mathrm{SizeOf}\,\mathit{id}} \\
\multicolumn{2}{l}{\quad \textbf{in } (\mathit{id}, n, \mathit{size} - n)}
\end{array}$$

Using the fact that none of *newId*, *newq* or *newInf* occurs in the body of the specification, with the **No Occurrence of Local Definition** law, the specification reduces to

$$\begin{array}{ll}
\textbf{let} & \mathit{id} = \mathrm{CurrentId}\,c \parallel n = \mathrm{PagesPrinted}\,c \\
\& & \mathit{size} = \mathrm{SizeOf}\,\mathit{id} \\
\textbf{in} & (\mathit{id}, n, \mathit{size} - n)
\end{array}$$

as required.

In section 4.3 we saw how the monad for state and exceptions could be used to structure a large specification. We now look at how properties of such specifications might be formulated and reasoned about, using the same equivalence laws, augmented by the monad laws.

5.4.3 Reasoning with Monads

We return to the specification of the printing control system using monads as described in section 4.3. Suppose we have the following functions defined using the monad $ST_\Sigma A$ and the five functions *unit*, *fetch*, *assign*, *raise* and $(\star)$ as described in section 4.3.3,

$$\begin{aligned}\text{Add} &: \textsc{Person} \times \textsc{File} \times \textsc{Priority} \to ST_\Sigma() \\ \text{Remove} &: \textsc{JobId} \to ST_\Sigma() \\ \text{GetId} &: \textsc{File} \to ST_\Sigma\textsc{JobId}\end{aligned}$$

where Add is as specified in section 4.3.3; Remove deletes the supplied JobId from the printer queue if it is there, and otherwise reports an error; and GetId retrieves the JobId of the supplied File from the printer queue, leaving the printer queue unchanged.

We may want to express that, under certain conditions, adding a file to the printer queue and then removing that same job leaves the printer queue unchanged. Using the monad notation, this may be expressed as, under certain conditions,

$$\text{Add}(p, f, n) \star (\textbf{fun}\ \_ \in () : \text{GetId}\, f \star \text{Remove}) \;\equiv\; \text{unit}() \tag{5.4}$$

where $\_ \in ()$ indicates that the function is not expecting a value and unit() is the state transformer which leaves the state unchanged and returns no value. We may define the shorthand $m \,\text{\textsemicolon}\, E \mathrel{\hat{=}} m \star (\textbf{fun}\ \_ \in () : E)$ so that the above expression is written as the more elegant

$$\text{Add}(p, f, n) \,;\, (\text{GetId}\, f \star \text{Remove}) \;\equiv\; \text{unit}() \tag{5.5}$$

This proof may be carried out by equational reasoning using the equivalence laws of the specification language and the monad laws for $ST_\Sigma A$.

We recognise that there is a certain amount of difficulty involved in formulating such properties of specifications. Although the use of the state monad here is intended to hide the explicit treatment of state in the specification, making the specification more readable, it is clear that in order to write down property (5.4) above, a knowledge of the monad, and how it works, is required. In fact, while the use of the state monad with exceptions makes the specification easier to read, this style of specification prevents us from formulating properties in the usual functional style, as can be seen from (5.4) and (5.5) above.

Although the monad laws may now be used in proofs, it is not clear that proofs become easier, since these laws only apply to that part of a specification involving the monad. It is likely that the monad laws will be used only to unfold the monad definitions, to obtain a purely functional specification like that of section 4.2, so that the equivalence laws of section 5.2 can then be applied.

5.4.4 Reasoning about Δ

In section 3.3 the multiplication problem was specified as:

$$\text{Multiply} \;\hat{=}\; (\textbf{fun}\ x, y \in \text{Number} : []/\{z \in \text{Number} : \text{Convert}\, z = \text{Convert}\, x * \text{Convert}\, y\}) \tag{5.6}$$

where

$$\begin{aligned}
\text{Digit} &\;\hat{=}\; \{x \in \mathbb{Z} : 0 \leqslant x \wedge x \leqslant 9\}\\
\text{Number} &\;\hat{=}\; \{s \in \mathit{FSeq}_1\,\text{Digit} : s[0] \neq 0\}\\
\text{Convert} &\;\hat{=}\; (\textbf{fun}\ s \in \text{Number} : (+)/\langle i : \mathit{dom}\ s \ggg 10^{\#s-(i+1)} * s[i]\rangle)
\end{aligned}$$

It was stated that the set in (5.6) is a singleton set. We now intend to show how it is possible to prove such a statement.

Consider the set

$$\{z \in \text{Number} : \text{Convert}\, z = \text{Convert}\, x * \text{Convert}\, y\} \tag{5.7}$$

where $\Delta\, x$, $\Delta\, y$ and $\Delta\, z$, since they are all variables. We first show that all terms in (5.7) are proper. Let w be one of x, y or z.

$$\begin{aligned}
& \Delta(\text{Convert}\, w)\\
\doteq \quad & \text{“}\textbf{Substitution, } \Delta\, w\text{”}\\
& \Delta((+)/\langle i : \mathit{dom}\ w \ggg 10^{\#w-(i+1)} * w[i]\rangle)\\
\Leftarrow \quad & \text{“}\Delta((+)/)\text{, properties of the operators”}\\
& \Delta\, w \wedge \Delta(\mathit{dom}\ w) \wedge \Delta(\#w) \wedge (\forall\, i : \mathit{dom}\ w \mid \bullet \Delta(w[i]))\\
\Leftarrow \quad & \text{“}\textbf{Axioms for Sequences, Axioms for Logical Values}\text{”}\\
& \Delta\, w \wedge (\mathit{dom}\ w \neq \mathbb{N}) \wedge (\forall\, i : \mathit{dom}\ w \mid \bullet i \in \mathit{dom}\ w)\\
\Leftarrow \quad & \text{“Given } \Delta\, w,\ w \in \text{Number, quantification trivially true”}\\
& \mathit{True}
\end{aligned}$$

The axioms for sequences being used here are:

$$\Delta(\mathit{dom}\ S) \Leftarrow \Delta\, S$$

$$\Delta(\#S) \Leftarrow \Delta\, S \wedge (\text{dom } S \not\equiv \mathbb{N})$$
$$\Delta(S[j]) \Leftarrow \Delta\, S \wedge \Delta\, j \wedge j \in \mathit{dom}\, S$$

Notice that, because $\Delta\, z$ and $\Delta(\text{Convert}\, x * \text{Convert}\, y)$, the set (5.7) can be written as:

$$\{z \in \text{Number} : \text{Convert}\, z \equiv \text{Convert}\, x * \text{Convert}\, y\} \tag{5.8}$$

Now that we know that all terms of the set (5.8) are proper, we can reason that it is a singleton set in the usual manner. Let z_1 and z_2 be members of the set. Then, using the axiom for set membership, transitivity of equivalence and substitution,

$$(+)/\langle i : \mathit{dom}\, z_1 \ggg 10^{\#z_1-(i+1)} * z_1[i]\rangle \equiv (+)/\langle i : \mathit{dom}\, z_2 \ggg 10^{\#z_2-(i+1)} * z_2[i]\rangle$$

Using induction on the minimum of the lengths of the sequences, $\sqcap(\#z_1, \#z_2)$, and the fact that both sequences are elements of Number, it is possible to show that $z_1 \equiv z_2$.

In general, it will not be necessary to go into such detail about the Δ properties of expressions and sub-expressions. The purpose of the axioms in these cases is to ensure that reasoning is possible, and under what conditions normal reasoning can go ahead.

5.4.5 Simple Refinements

We now turn to refinement. The first few examples are very simple and demonstrate just a few of the laws. A slightly larger example, involving recursion, follows.

Taking example (5.1), given previously, we prove some simple refinements.

$$
\begin{array}{ll}
 & [\!]/\mathbb{N} \\
\sqsubseteq & \quad \textbf{“Reduce Non-Determinacy, } \{2,3\} \subseteq \mathbb{N}\textbf{”} \\
 & [\!]/\{2,3\} \\
\equiv & \quad \textbf{“Axioms for Generalised Choice”} \\
 & 2 \mathbin{[\!]} 3
\end{array}
$$

and from (5.2)

$$
\begin{array}{ll}
 & (\textbf{fun}\ x \in \mathbb{N} : x+2 \mathbin{[\!]} x+3) \\
\equiv & \quad \text{“Unfold partially defined function”} \\
 & (\textbf{fun}\ x \in \mathbb{Z} : (x \in \mathbb{N}) \succ\!\!- (x+2 \mathbin{[\!]} x+3)) \\
\sqsubseteq & \quad \textbf{“Remove Assumption”} \\
 & (\textbf{fun}\ x \in \mathbb{Z} : x+2 \mathbin{[\!]} x+3)
\end{array}
$$

as expected.

It is possible to prove the second **Refine Function** law using some other laws. Assuming that $(\forall x : T \mid \bullet \Delta P)$, we refine

$$\begin{array}{ll}
 & (\textbf{fun}\ x \in T : E \mathbin{[\!]} F) \\
\equiv & \quad \text{“}\textbf{Introduce Alternation}, \Delta P \text{ for any } x \text{ in } T\text{”} \\
 & (\textbf{fun}\ x \in T : P \rightarrow (E \mathbin{[\!]} F) \mathbin{[\!]} \neg P \rightarrow (E \mathbin{[\!]} F)) \\
\sqsubseteq & \quad \text{“}\textbf{Reduce Non-Determinacy}\text{”} \\
 & (\textbf{fun}\ x \in T : P \rightarrow E \mathbin{[\!]} \neg P \rightarrow F) \\
\equiv & \quad \text{“}\textbf{Alternation to Conditional}, \Delta P\text{”} \\
 & (\textbf{fun}\ x \in T : P \rightarrow E \mathbin{\overset{\langle}{[\!]}} F)
\end{array}$$

as stated.

We prove a form of distribution of function abstraction over choice

$$(\textbf{fun}\ x \in T : E \mathbin{[\!]} F) \ \sqsubseteq\ (\textbf{fun}\ x \in T : E) \mathbin{[\!]} (\textbf{fun}\ x \in T : F) \tag{5.9}$$

as follows. We have, from the **Reduce Non-Determinacy** law and monotonicity,

$$\begin{array}{lcl}
(\textbf{fun}\ x \in T : E \mathbin{[\!]} F) & \sqsubseteq & (\textbf{fun}\ x \in T : E) \\
(\textbf{fun}\ x \in T : E \mathbin{[\!]} F) & \sqsubseteq & (\textbf{fun}\ x \in T : F)
\end{array}$$

So, by simply applying the **Introduce Choice** law we arrive at exactly (5.9). We call this the **Under-determined Choice** law.

A more challenging refinement, using the **Recursion Introduction** law, is now described.

5.4.6 Refinement with Recursion

We want to refine the following specification, for x and y of type $Seq\ \mathbb{Z}$,

$$\begin{array}{lcl}
\text{zip}[x, y] & \hat{=} & [\!]/\{S \in Seq\ (\mathbb{Z} \times \mathbb{Z}) : \#S = (\#x \sqcap \#y) \\
 & & \quad \wedge (\forall i \in \{0 \ldots \#S - 1\} \bullet S[i] = (x[i], y[i]))\}
\end{array} \tag{5.10}$$

In the following derivation, we define the function tl for all non-empty sequences.

$$\begin{array}{lcll}
tl\ S & \hat{=} & \langle i : \{0 \ldots \#S - 2\} \ggg S[i+1]\rangle & \text{S finite} \\
 & & \langle i : dom\ S \ggg S[i+1]\rangle & \text{S infinite}
\end{array}$$

As a first step in the refinement, it makes sense to introduce an alternation, using the general form of the **Alternation Introduction** law. Possible cases are: $(x = \langle\rangle)$; $(y = \langle\rangle)$; or $(x \neq \langle\rangle \wedge y \neq \langle\rangle)$. Note that the guards are all well-defined.

$$\text{zip}[x, y] \equiv (x = \langle\rangle) \rightarrow \text{zip}[x, y] \,[]\, (y = \langle\rangle) \rightarrow \text{zip}[x, y] \,[]\, (x \neq \langle\rangle \wedge y \neq \langle\rangle) \rightarrow \text{zip}[x, y] \qquad (5.11)$$

Each case may be refined in turn, using the fact that choice is monotonic with respect to refinement of subexpressions.

We refine

$$\begin{array}{ll}
 & (x = \langle\rangle) \rightarrow \text{zip}[x, y] \\
\equiv & \quad \text{“Expand definition of zip}[x, y]\text{”} \\
 & (x = \langle\rangle) \rightarrow \\
 & []/\{S \in \mathit{Seq}\,(\mathbb{Z} \times \mathbb{Z}) : \#S = (\#x \sqcap \#y) \wedge (\forall\, i \in \{0 \ldots \#S - 1\} \bullet S[i] = (x[i], y[i]))\} \\
\sqsubseteq & \quad \text{“}\textbf{Using Context in Guard}, \#x = 0 \text{ and } \#y \geqslant 0\text{”} \\
 & (x = \langle\rangle) \rightarrow []/\{S \in \mathit{Seq}\,(\mathbb{Z} \times \mathbb{Z}) : (\#S = 0) \wedge \mathit{True}\} \\
= & \quad \text{“Singleton Set, }\textbf{Properties of Generalised Choice}\text{”} \\
 & (x = \langle\rangle) \rightarrow \langle\rangle
\end{array}$$

Using a similar refinement sequence for the second case, we have

$$(y = \langle\rangle) \rightarrow \text{zip}[x, y] \quad \sqsubseteq \quad (y = \langle\rangle) \rightarrow \langle\rangle$$

Now, turning to the last case of the alternation, we refine with the aim of forming an expression suitable for an application of the **Recursion Introduction** law.

$$\begin{array}{ll}
 & (x \neq \langle\rangle \wedge y \neq \langle\rangle) \rightarrow \text{zip}[x, y] \\
\equiv & \quad \text{“Expand Definition of zip}[x, y]\text{”} \\
 & (x \neq \langle\rangle \wedge y \neq \langle\rangle) \rightarrow \\
 & []/\{S \in \mathit{Seq}\,(\mathbb{Z} \times \mathbb{Z}) : \#S = (\#x \sqcap \#y) \wedge (\forall\, i \in \{0 \ldots \#S - 1\} \bullet S[i] = (x[i], y[i]))\} \\
\sqsubseteq & \quad \text{“}\textbf{Using Context in Guard}, \#S > 0\text{”} \\
 & (x \neq \langle\rangle \wedge y \neq \langle\rangle) \rightarrow []/\{S \in \mathit{Seq}\,(\mathbb{Z} \times \mathbb{Z}) : \#S = 1 + (\#\mathit{tl}\, x \sqcap \#\mathit{tl}\, y) \\
 & \qquad\qquad \wedge\, S[0] = (x[0], y[0]) \\
 & \qquad\qquad \wedge\, (\forall\, i \in \{1 \ldots \#S - 1\} \bullet S[i] = (\mathit{tl}\, x[i-1], \mathit{tl}\, y[i-1]))\} \\
= & \quad \text{“Set Manipulations, }\textbf{Distribution of Concatenation over Choice}\text{”} \\
 & (x \neq \langle\rangle \wedge y \neq \langle\rangle) \rightarrow
\end{array}$$

$$\langle(x[0], y[0])\rangle \frown [\!]/\{S \in \mathit{Seq}\,(\mathbb{Z} \times \mathbb{Z}) : \#S = (\#\mathit{tl}\ x \sqcap \#\mathit{tl}\ y) \wedge (\forall i \in \{0 \ldots \#S - 1\} \bullet S[i] = (\mathit{tl}\ x[i], \mathit{tl}\ y[i]))\}$$

$\equiv$ "Definition of zip$[x, y]$, with substitutions"

$$(x \neq \langle\rangle \wedge y \neq \langle\rangle) \rightarrow \langle(x[0], y[0])\rangle \frown \mathrm{zip}[x, y][^{\mathit{tl}\ x, \mathit{tl}\ y}_{x, y}]$$

$\equiv$ **"Axioms for Assumptions**, $(\#\mathit{tl}\ x < \#x) \wedge (\#\mathit{tl}\ y < \#y) \equiv \mathit{True}$"

$$(x \neq \langle\rangle \wedge y \neq \langle\rangle) \rightarrow \langle(x[0], y[0])\rangle \frown ((\#\mathit{tl}\ x < \#x) \wedge (\#\mathit{tl}\ y < \#y) \succ\!\!- \mathrm{zip}[x, y][^{\mathit{tl}\ x, \mathit{tl}\ y}_{x, y}])$$

$\equiv$ **"Substitution"**

$$\begin{aligned}&(x \neq \langle\rangle \wedge y \neq \langle\rangle) \rightarrow\\ &\qquad \langle(x[0], y[0])\rangle \frown\\ &\qquad (\mathbf{fun}\ x', y' \in \mathit{Seq}\,\mathbb{Z} : (\#x' < \#x) \wedge (\#y' < \#y) \succ\!\!- \mathrm{zip}[x, y][^{x', y'}_{x, y}])(\mathit{tl}\ x, \mathit{tl}\ y)\end{aligned}$$

The three parts of the specification are now combined, using monotonicity of choice with respect to refinement.

zip$[x, y]$

$\sqsubseteq$ "From (5.11) and partial refinements"

$$\begin{aligned}&(x = \langle\rangle) \rightarrow \langle\rangle\\ &[\!]\ (y = \langle\rangle) \rightarrow \langle\rangle\\ &[\!]\ (x \neq \langle\rangle \wedge y \neq \langle\rangle) \rightarrow\\ &\qquad \langle(x[0], y[0])\rangle \frown\\ &\qquad (\mathbf{fun}\ x', y' \in \mathit{Seq}\,\mathbb{Z} : (\#x' < \#x) \wedge (\#y' < \#y) \succ\!\!- \mathrm{zip}[x, y][^{x', y'}_{x, y}])(\mathit{tl}\ x, \mathit{tl}\ y)\end{aligned}$$

$\sqsubseteq$ **"Recursion Introduction"**

$$\begin{aligned}&\mathbf{let}\ f = (\mathbf{fun}\ x, y \in \mathit{Seq}\,\mathbb{Z} : (x = \langle\rangle) \rightarrow \langle\rangle\\ &\qquad\qquad [\!]\ (y = \langle\rangle) \rightarrow \langle\rangle\\ &\qquad\qquad [\!]\ (x \neq \langle\rangle \wedge y \neq \langle\rangle) \rightarrow \langle(x[0], y[0])\rangle \frown f(\mathit{tl}\ x, \mathit{tl}\ y))\\ &\mathbf{in}\ f(x, y)\end{aligned}$$

which is a reasonable implementation of the zip function.

5.4.7 The N-Queens Revisited

The N-Queens problem, to place N queens on an $N \times N$ chessboard such that no queen can take any of the others, where $N \geqslant 4$, was specified in section 3.3 using the expression language. In this section we aim to derive an algorithm for the problem.

This example serves to illustrate a number of properties. Firstly, it shows how reasoning about potentially partial expressions might proceed in practice. In fact, this reasoning is usually informal, but serves to exhibit possible danger points and invariants to be observed in a derivation.

Secondly, the derivation is interesting in that almost all the steps are equivalences rather than refinements. The two places where refinement occurs are: in a **Recursion Introduction** step; and the final choice of one solution from the set of all solutions. So, what is happening is that the original specification is being manipulated, ready for the recursion step.

Thirdly, the specification uses sets of sets of pairs as the basic data structure. This means that a lot of the reasoning uses the **Axioms for Sets**. However, most programming languages don't supply sets as a basic data structure, so it is likely that the final expression derived here would need to be further refined, using data refinement. The target data structure is likely to be a sequence of mappings.

Finally, during the derivation we make reference to the application of the **Axioms for Sets** and the **Axioms for Logical Expressions** without demonstrating how the axioms are actually applied. This is to aid clarity and to present the derivation in a reasonable length.

The initial specification as given in section 3.3 is:

$$[]/\{Pl \in \text{Placing} \;:\; \text{SafePlacing}\, Pl\} \tag{5.12}$$

where we have the following definitions:

$$\begin{aligned}
\text{Position} &\;\hat{=}\; \{1..N\} \times \{1..N\} \\
\text{Placing} &\;\hat{=}\; \{Pl \in \mathbb{P}\,\text{Position} \;:\; \#Pl = N\} \\
\text{SafePlacing} &\;\hat{=}\; (\mathbf{fun}\; Pl \in \text{Placing} : (\forall\, p_1, p_2 : Pl \mid \bullet \text{CantTake}\, p_1\, p_2))
\end{aligned}$$

The function CantTake describes the property that two queens cannot take each other.

A Note on Partiality

Our initial specification (5.12) is potentially partial, being a choice over a set. The specification should be given as

$$[]/\{Pl \in \text{Placing} \;:\; \text{SafePlacing}\, Pl\} \;\overset{\leftarrow}{\lceil}\; \bot$$

This is a problem because, since $\overleftarrow{[]}$ is not monotonic in its first argument, any derivations of (5.12) should be equivalences, not refinements. This is not appropriate since the set of all possible placings may contain more than one element, and we want to choose just one.

In fact, refining the left argument of the operator $\overleftarrow{]}$ is not usually a problem, as long as we can be sure that any refinements do not result in the partial value $\top$. So, we need to ensure that any refinements of expression (5.12) are always total. In this case it means ensuring that the set is non-empty.

Luckily, knowledge of the problem domain assures us that at least one solution exists for any $N \geqslant 4$. This is given as an assumption in the problem statement. And so we may proceed to refine, with caution.

Preliminaries

As a preliminary to the derivation, we notice the following.

$$\text{SafePlacing} \sqsubseteq (\textbf{fun } Pl \in \mathbb{P}\,\text{Position} : (\forall p_1, p_2 : Pl \mid \bullet \text{CantTake}\, p_1\, p_2))$$

by the **Weaken Assumption** law. We define

$$\text{Safe} \mathrel{\hat{=}} (\textbf{fun } Pl \in \mathbb{P}\,\text{Position} : (\forall p_1, p_2 : Pl \mid \bullet \text{CantTake}\, p_1\, p_2))$$

and note the following:

$$\begin{aligned} &\text{Safe}\, \emptyset \equiv \mathit{True} \\ &Pl' \subseteq Pl \Rightarrow (\text{Safe} Pl \Rightarrow \text{Safe} Pl') \end{aligned}$$

for proper Pl' and Pl. These are easily illustrated from the definition of CantTake as given in section 3.3. Also

$$\text{Safe} Pl \equiv \text{SafePlacing} Pl$$

when $Pl \in$ Placing.

The Derivation

We intend to build up the set of all possible solutions for a given N, without saying how to choose a particular solution. From (5.12), we take the set of all possible solutions and derive:

$$\begin{array}{ll}
 & \{Pl \in \text{Placing} : \text{SafePlacing}\, Pl\} \\
\equiv & \quad \text{“Definition of Placing, set theory”} \\
 & \{Pl \in \mathbb{P}\,\text{Position} : \#Pl = N \wedge \text{SafePlacing}\, Pl\} \\
\equiv & \quad \text{“Above observation”} \\
 & \{Pl \in \mathbb{P}\,\text{Position} : \#Pl = N \wedge \text{Safe}\, Pl\} \\
\equiv & \quad \text{“}\#Pl = N \wedge \text{Safe}\, Pl \Rightarrow \textbf{fst} * Pl = \{1..N\}\text{”} \\
 & \{Pl \in \mathbb{P}\,\text{Position} : \#Pl = N \wedge \text{Safe}\, Pl \wedge \textbf{fst} * Pl = \{1..N\}\} \\
\equiv & \quad \text{“}\textbf{Substitution}, \Delta\, N\text{”} \\
 & (\textbf{fun}\ m \in \mathbb{N} : \{Pl \in \mathbb{P}\,\text{Position} : \#Pl = m \wedge \text{Safe}\, Pl \wedge \textbf{fst} * Pl = \{1..m\}\})\ N
\end{array}$$

We are interested in the body of this function, the set, which we call Q.

$$Q \mathrel{\hat{=}} \{Pl \in \mathbb{P}\,\text{Position} : \#Pl = m \wedge \text{Safe}\, Pl \wedge \textbf{fst} * Pl = \{1..m\}\} \tag{5.13}$$

The intention is to manipulate the set Q so that a recursion can be introduced. Working just with Q alone, to ease readability, we use the **Alternation Introduction** law. Since $m \in \mathbb{N}$ we also use the **Context in Assumption** law to obtain:

$$Q \equiv (m = 0) \rightarrow Q \,[]\, (m > 0) \rightarrow Q \tag{5.14}$$

Notice that both guards are proper, since m is a variable.

We refine each case in turn, using the fact that choice is monotonic with respect to refinement. For the simple case:

$$\begin{array}{ll}
 & (m = 0) \rightarrow Q \\
\equiv & \quad \text{“Expand definition of } Q\text{”} \\
 & (m = 0) \rightarrow \{Pl \in \mathbb{P}\,\text{Position} : \#Pl = m \wedge \text{Safe}\, Pl \wedge \textbf{fst} * Pl = \{1..m\}\} \\
\equiv & \quad \text{“}\textbf{Using Context in Guard}\text{”} \\
 & (m = 0) \rightarrow \{Pl \in \mathbb{P}\,\text{Position} : \#Pl = 0 \wedge \text{Safe}\, Pl \wedge \textbf{fst} * Pl = \emptyset\} \\
\equiv & \quad \text{“Since } \#Pl = 0 \Rightarrow Pl = \emptyset\text{, and Safe}\,\emptyset\text{”} \\
 & (m = 0) \rightarrow \{\emptyset\}
\end{array}$$

For the second case we want to introduce a recursion.

We need to make each Pl smaller, reducing by one element. For each Pl there is a proper subset Pl' such that $Pl \equiv Pl' \cup \{(m, n)\}$ for some $n \in \{1..N\}$. This follows from **fst** $* Pl = \{1..m\}$. From Safe Pl we further conclude that there is only one such n, and so

$\mathbf{fst} * Pl' = \{1..m-1\}$ and $\#Pl' = m-1$. In addition, since $Pl' \subset Pl$, from the observations about Safe positionings, Pl' must also be safe. So, we can take all the safe sets of positions of size $m-1$ (since $m > 0$), add in the position (m, n) for each n in turn, and test to see if the extended set is safe.

Formally,

$$
\begin{array}{ll}
 & (m > 0) \to Q \\
\equiv & \quad \text{“Expand definition of } Q\text{”} \\
 & (m > 0) \to \{Pl \in \mathbb{P}\,\mathrm{Position} \;:\; \#Pl = m \wedge \mathrm{Safe}\,Pl \wedge \mathbf{fst} * Pl = \{1..m\}\} \\
\equiv & \quad \text{“}\textbf{Using Context in Guard, Axioms for Sets}\text{ and observations”} \\
 & (m > 0) \to \{Pl' \in \mathbb{P}\,\mathrm{Position}, n \subset \{1..N\} : \\
 & \qquad \#Pl' = m-1 \wedge \mathrm{Safe}(Pl' \cup \{(m,n)\}) \wedge \mathbf{fst} * Pl' = \{1..m-1\} : \\
 & \qquad Pl' \cup \{(m,n)\}\} \\
\equiv & \quad \text{“Safe}(Pl' \cup \{(m,n)\}) \Rightarrow \mathrm{Safe}Pl'\text{, }\textbf{Axioms for Logical Values}\text{”} \\
 & (m > 0) \to \{Pl' \in \mathbb{P}\,\mathrm{Position}, n \in \{1..N\} : \\
 & \qquad \#Pl' = m-1 \wedge \mathrm{Safe}Pl' \wedge \mathbf{fst} * Pl' = \{1..m-1\} \wedge \mathrm{Safe}(Pl' \cup \{(m,n)\}) : \\
 & \qquad Pl' \cup \{(m,n)\}\} \\
\equiv & \quad \text{“Definition of } Q \text{ with substitutions, }\textbf{Axioms for Sets}\text{”} \\
 & (m > 0) \to \{Pl' \in \mathbb{P}\,\mathrm{Position}, n \in \{1..N\} : \\
 & \qquad Pl' \in Q[^{m-1}_{m}] \wedge \mathrm{Safe}(Pl' \cup \{(m,n)\}) : Pl' \cup \{(m,n)\}\} \\
\equiv & \quad \text{“}\textbf{Axioms for Sets}\text{”} \\
 & (m > 0) \to \{Pl' \in Q[^{m-1}_{m}], n \in \{1..N\} : \mathrm{Safe}(Pl' \cup \{(m,n)\}) : Pl' \cup \{(m,n)\}\} \\
\equiv & \quad \text{“}\textbf{Axioms for Assumptions}\text{, } (m-1 < m) = \mathit{True}\text{”} \\
 & (m > 0) \to \{Pl' \in (m-1 < m \succ Q[^{m-1}_{m}]), n \in \{1..N\} : \\
 & \qquad\qquad\qquad \mathrm{Safe}(Pl' \cup \{(m,n)\}) : Pl' \cup \{(m,n)\}\} \\
\equiv & \quad \text{“}\textbf{Substitution}\text{, } \Delta(m-1)\text{”} \\
 & (m > 0) \to \{Pl' \in ((\mathbf{fun}\ m' \in \mathbb{N} : m' < m \succ Q[^{m'}_{m}])(m-1)), n \in \{1..N\} : \\
 & \qquad \mathrm{Safe}(Pl' \cup \{(m,n)\}) : Pl' \cup \{(m,n)\}\}
\end{array}
$$

Now, combining the two cases, from (5.14):

$$
\begin{array}{ll}
 & Q \\
\equiv & \quad \text{“Partial Derivations”} \\
 & (m = 0) \to \{\emptyset\} \\
 & [\!]\ (m > 0) \to \{Pl' \in ((\mathbf{fun}\ m' \in \mathbb{N} : m' < m \succ Q[^{m'}_{m}])m-1), n \in \{1..N\} : \\
 & \qquad \mathrm{Safe}(Pl' \cup \{(m,n)\}) : Pl' \cup \{(m,n)\}\} \\
\sqsubseteq & \quad \text{“}\textbf{Recursion Introduction}\text{”}
\end{array}
$$

$$\begin{array}{l} \textbf{let } \mathit{queens} = (\textbf{fun } m \in \mathbb{N} : \\ \qquad m = 0 \rightarrow \{\emptyset\} \\ \qquad [\!]\; m > 0 \rightarrow \{Pl' \in \mathit{queens}(m-1), n \in \{1..N\} : \mathrm{Safe}(Pl' \cup \{(m,n)\}) : \\ \qquad\qquad Pl' \cup \{(m,n)\}\}) \\ \textbf{in } \mathit{queens}\; m \end{array}$$

Now, returning to the initial derivation of the set of all possible solutions for fixed value N,

$$\begin{array}{ll} & \{Pl \in \mathrm{Placing} : \mathrm{SafePlacing}\, Pl\} \\ \equiv & \quad \text{"Previous derivation"} \\ & (\textbf{fun } m \in \mathbb{N} : \{Pl \in \mathbb{P}\,\mathrm{Position} : \#Pl = m \wedge \mathrm{Safe}\, Pl \wedge \textbf{fst} * Pl = \{1..m\}\})\, N \\ \sqsubseteq & \quad \text{"Above refinements, abstraction monotonic wrt refinement"} \\ & (\textbf{fun } m \in \mathbb{N} : \\ & \quad \textbf{let } \mathit{queens} = (\textbf{fun } m \in \mathbb{N} : \\ & \qquad m = 0 \rightarrow \{\emptyset\} \\ & \qquad [\!]\; m > 0 \rightarrow \{Pl' \in \mathit{queens}(m-1), n \in \{1..N\} : \\ & \qquad\qquad \mathrm{Safe}(Pl' \cup \{(m,n)\}) : Pl' \cup \{(m,n)\}\}) \\ & \quad \textbf{in } \mathit{queens}\; m)N \\ \equiv & \quad \text{"}\textbf{Substitution}, \Delta\, N\text{"} \\ & \textbf{let } \mathit{queens} = (\textbf{fun } m \in \mathbb{N} : \\ & \qquad m = 0 \rightarrow \{\emptyset\} \\ & \qquad [\!]\; m > 0 \rightarrow \{Pl' \in \mathit{queens}(m-1), n \in \{1..N\} : \\ & \qquad\qquad \mathrm{Safe}(Pl' \cup \{(m,n)\}) : Pl' \cup \{(m,n)\}\}) \\ & \textbf{in } \mathit{queens}\; N \end{array}$$

In the recursive function above, the majority of the work is being done by the function Safe in the computation of Safe$(Pl' \cup \{(m,n)\})$. In fact, it is doing much more work than is necessary, since it is already known that Safe$Pl' \equiv \mathit{True}$. Using this fact, and also that $\textbf{fst} * Pl' = \{1..m-1\}$, we simplify:

$$\begin{array}{ll} & \mathrm{Safe}(Pl' \cup \{(m,n)\}) \\ \equiv & \quad \text{"Definition of Safe, }\textbf{Substitution}, \Delta(Pl' \cup \{(m,n)\})\text{"} \\ & (\forall\, p_1, p_2 : Pl' \cup \{(m,n)\} \mid \bullet \mathrm{CantTake}\, p_1\, p_2) \\ = & \quad \text{"}\textbf{Axioms for Sets}\text{"} \\ & (\forall\, p_1, p_2 : Pl' \mid \bullet \mathrm{CantTake}\, p_1\, p_2) \wedge (\forall\, p : Pl' \mid \bullet \mathrm{CantTake}\, p\, (m,n) \wedge \mathrm{CantTake}\, (m,n)\, p) \\ \equiv & \quad \text{"Definition of Safe"} \\ & \mathrm{Safe}Pl' \wedge (\forall\, p : Pl' \mid \bullet \mathrm{CantTake}\, p\, (m,n) \wedge \mathrm{CantTake}\, (m,n)\, p) \\ \equiv & \quad \text{"Safe}Pl' \equiv \mathit{True}\text{, by assumption"} \end{array}$$

$$
\begin{array}{ll}
 & (\forall p : Pl' \mid \bullet \text{CantTake}\, p\, (m, n) \wedge \text{CantTake}\, (m, n)\, p) \\
\equiv & \quad \text{“Definition of CantTake, } \textbf{Substitution}\text{, all terms proper”} \\
 & (\forall p : Pl' \mid \bullet (\textbf{fst}\, p = m \vee \textbf{snd}\, p = n \vee \mid \textbf{fst}\, p - m \mid = \mid \textbf{snd}\, p - n \mid) \Rightarrow p = (m, n)) \\
\equiv & \quad \text{“Know that fst} * Pl' = \{1..m-1\}\text{, so fst}\, p = m \equiv \mathit{False} \text{ and } p = (m, n) \equiv \mathit{False}\text{”} \\
 & (\forall p : Pl' \mid \bullet \textbf{snd}\, p \neq n \wedge (m - \textbf{fst}\, p) \neq \mid \textbf{snd}\, p - n \mid)
\end{array}
$$

This is a much simpler condition to check.

We now define, for simplicity, the function Check, as follows:

$$
\begin{array}{lll}
\text{Check} & \hat{=} & (\textbf{fun}\ Pl \in \mathbb{P}\,\text{Position}, pos \in \text{Position} : \\
 & & \quad (\forall p : Pl \mid \bullet \textbf{snd}\, p \neq \textbf{snd}\, pos \wedge (\text{fst}\, pos - \textbf{fst}\, p) \neq \mid \textbf{snd}\, p - \textbf{snd}\, pos \mid))
\end{array}
$$

The Final Specification

Returning to the initial specification (5.12) we can now present the complete final specification.

$$
\begin{array}{ll}
 & [\!]/\{Pl \in \text{Placing} : \text{SafePlacing}\, Pl\} \\
\sqsubseteq & \quad \text{“Above Refinements”} \\
 & [\!]/(\textbf{let}\ \mathit{queens} = (\textbf{fun}\ m \in \mathbb{N} : \\
 & \qquad\qquad m = 0 \rightarrow \{\emptyset\} \\
 & \qquad\qquad [\!]\ m > 0 \rightarrow \{Pl' \in \mathit{queens}(m-1), n \in \{1..N\} : \\
 & \qquad\qquad\qquad \text{Safe}(Pl' \cup \{(m, n)\}) : Pl' \cup \{(m, n)\}\}) \\
 & \quad \textbf{in}\ \mathit{queens}\ N) \\
\equiv & \quad \text{“}\textbf{Substitution}\text{, Proper terms, } \textbf{Distribute Function Application inside let}\text{,} \\
 & \quad\ \text{Above simplification of Safe}(Pl' \cup \{(m, n)\})\text{”} \\
 & \textbf{let}\ \mathit{queens} = (\textbf{fun}\ m \in \mathbb{N} : \\
 & \qquad m = 0 \rightarrow \{\emptyset\} \\
 & \qquad [\!]\ m > 0 \rightarrow \{Pl' \in \mathit{queens}(m-1), n \in \{1..N\} : \\
 & \qquad\qquad \text{Check}\, Pl'\, (m, n) : Pl' \cup \{(m, n)\}\}) \\
 & \textbf{in}\ [\!]/(\mathit{queens}\, N)
\end{array}
$$

Comments

The above derivation is based very heavily on the axioms for sets. In general, sets do not form part of a programming language. What is required is some form of data refinement which will map each set and set operation to a data type and associated operation of the target language. An appropriate data type is likely to be that of sequences.

Notice that all the steps, except the application of the **Recursion Introduction** law, are equivalences rather than refinements. This is because, in building up the set of all solutions, we are adding no information to the original specification.

The final step, which has not been derived, would be to choose a single solution from the set of all solutions. This, necessarily, requires a refinement step since there is currently no information to say which solution would be preferred. However, after the data refinement has taken place, resulting in a sequence of all solutions (according to some ordering), the final refinement might be to choose the first placing in the sequence.

5.5 Towards Imperative Programming

In this section we illustrate the derivation of imperative style expressions using the example of Bresenham's line drawing algorithm [17, 85, 77]. This derivation originally appeared in [19], and is used here with modifications.

The example serves to demonstrate a number of points. First, the basic specification involves the use of real numbers, which are not included in the expression language. We assume that the real numbers used can be reasoned about in the usual way. Our target language does not include real numbers, and so part of our goal is to derive an implementation which uses integers only.

We assume, in this example, that all terms are well-defined. This makes reasoning easier, since all terms are, in addition, assumed to be proper. These assumptions are reasonable in the context.

Finally, our target language is taken to be a lazy functional language. This means that we use some functions which are not part of our specification language, but which are assumed to be a standard part of the functional language. Laziness is assumed because of the usual definition of these functions, which deal with possibly infinite sequences. Since our expression language already deals with infinite sequences, this does not present a problem.

Before the problem is described we anticipate the need for two additional refinement laws which did not appear in sections 5.2 and 5.3.2. These are given as follows.

Law (If Refinement)

$$(\textbf{if } P \textbf{ then } E_1 \textbf{ else } E_2) \sqsubseteq (\textbf{if } Q \textbf{ then } F_1 \textbf{ else } F_2)$$

Whenever

$$(P \wedge Q) \Rightarrow E_1 \sqsubseteq F_1 \qquad (P \wedge \neg Q) \Rightarrow E_1 \sqsubseteq F_2$$
$$(\neg P \wedge Q) \Rightarrow E_2 \sqsubseteq F_1 \qquad (\neg P \wedge \neg Q) \Rightarrow E_2 \sqsubseteq F_2$$

and with $\Delta\, P$ *and* $\Delta\, Q$.

This can be proved from the refinement laws **Disjunction of Guards**, **Using Context in Guards** and the transformation laws for **Alternations to Conditionals.**

Law (Application thru Conditional)

$$\Delta\, P \Rightarrow (f(\textbf{if } P \textbf{ then } E \textbf{ else } F) \equiv (\textbf{if } P \textbf{ then } f\ E \textbf{ else } f\ F))$$

This can be proved by case analysis on the guard.

5.5.1 Background to the Derivation Style

Our aim in this example is to transform an expression of the shape $f * \langle m \ldots n \rangle$ into a more iterative style of functional program, where calculation of $f(i+1)$ can re-use some of the work that went into calculating $f\ i$, for integer i such that $m \leqslant i < n$. Suppose that calculating $f(i+1)$ from $f\ i$ is performed by applying a (simple) function, called *next* say, i.e.

$$(\forall\, i : \mathbb{Z} \mid \bullet m \leqslant i < n \Rightarrow f(i+1) = next(f\ i))$$

then we would only have to apply f once, namely to m, the first integer in the sequence. After that, we could simply keep applying *next*. Naturally, this only reduces work if the function *next* is simpler (cheaper) than the origical function f.

This idea is expressed formally using the functions *take* and *iterate* which are part of the standard Haskell prelude and can be defined in any lazy functional programming language. The following theorem is stated from [19]:

Theorem (Map to Iterate)

$$(use \circ make)\langle m \ldots n \rangle = (take\#\langle m \ldots n \rangle \circ (use*) \circ iterate\ next)(make\ m)$$

if $m \leqslant i < n \Rightarrow make(i+1) = (next \circ make)i$.

This theorem states a more general notion than that given above. It says that to map a function f over an integer range, all we have to do is find three functions, here called *make*, *use* and *next*, such that $use \circ make$ is our original function f, and *next* captures a recurrence relation on *make*.

5.5.2 The Specification

Given two integer pairs (x_1, y_1) and (x_2, y_2), the line drawing problem is to find the pixels which best approximate the line segment between them. The mathematical representation of the (infinite) line is defined by the equation

$$f\,x \;\hat{=}\; y_1 + m * (x - x_1) \tag{5.15}$$

where m is the slope of the line and can be calculated from

$$m \;\hat{=}\; (y_2 - y_1)/(x_2 - x_1)$$

For convenience, we use the following abbreviations: $d_y \hat{=} y_2 - y_1$ and $d_x \hat{=} x_2 - x_1$. However, the points of a mathematical line are given by pairs of real numbers, while pixels are pairs of integers. We want to calculate those pixels which are nearest to the mathematical line, i.e. those which approximate the line.

Let us assume, for simplicity, that the value of the slope of the line is between 0 and 1. Other line segments can be obtained by symmetry. The problem now is to find, for the sequence of integer x-values $\langle x_1 \ldots x_2 \rangle$, those y-values which best approximate the mathematical line given by (5.15) using only integer arithmetic.

The line segment will be represented well if every $x \in \mathbb{Z}$ between x_1 and x_2 is paired with some $y \in \mathbb{Z}$ closest to $f\,x$. For convenience we define $n \,\hat{=}\, \#\langle x_1 \ldots x_2 \rangle$. Our initial specification is given by the expression

$$(round \circ f) * \langle x_1 \ldots x_2 \rangle \tag{5.16}$$

which computes the integer y-values for $\langle x_1 \ldots x_2 \rangle$. The function $round : \mathbb{R} \rightarrow \mathbb{Z}$, which gives a proper result for all real numbers, is defined by:

$$round\,x \;\hat{=}\; \textbf{if } x - \lfloor x \rfloor > 0.5 \textbf{ then } \lfloor x \rfloor + 1 \textbf{ else } \lfloor x \rfloor \tag{5.17}$$

where the floor of $x \in \mathbb{R}$, denoted $\lfloor x \rfloor$, has the usual properties:

$$\lfloor x \rfloor \leqslant x < \lfloor x \rfloor + 1$$

There are two problems with our initial specification. The first is that it uses real arithmetic, but takes as input and output only integers. We would prefer to use integer arithmetic only. Secondly, the algorithm is inefficient, since f is being applied to each member of the list $\langle x_1 \ldots x_2 \rangle$. We aim to use the **Map to Iterate** theorem to derive Bresenham's line drawing algorithm, which is efficient and uses integer arithmetic only.

5.5.3 Refinements

We define the integer function $r : \mathbb{Z} \to \mathbb{Z}$ as follows:

$$r \;\hat{=}\; round \circ f \tag{5.18}$$

The initial specification (5.16) is now written:

$$r * \langle x_1 \ldots x_2 \rangle$$

We can use the **Map to Iterate** theorem if a recurrence relation can be found for r. This should use integer arithmetic only. Consider $r(x+1)$, where $x_1 \leqslant x < x_2$,

$$\begin{array}{ll}
 & r(x+1) \\
\equiv & \quad \text{“Definition of } r \text{ (5.18)”} \\
 & (round \circ f)(x+1) \\
\equiv & \quad \text{“Definition of } round \text{ (5.17)”} \\
 & \textbf{if } f(x+1) - \lfloor f(x+1) \rfloor > 0.5 \textbf{ then } \lfloor f(x+1) \rfloor + 1 \textbf{ else } \lfloor f(x+1) \rfloor \\
\sqsubseteq & \quad \text{“}\textbf{If Refinement}\text{, proof requirements below”} \\
 & \textbf{if } (f(x+1) - r\,x) > 0.5 \textbf{ then } r\,x + 1 \textbf{ else } r\,x \\
\equiv & \quad \text{“For suitable } e\text{, see below”} \\
 & \textbf{if } e\,x < 0 \textbf{ then } r\,x + 1 \textbf{ else } r\,x
\end{array}$$

In the above derivation, the **If Refinement** law can be used only if the guard is proper (which it is) and if the four proof requirements are satisfied. For example, we have to show

$$(f(x+1) - \lfloor f(x+1) \rfloor > 0.5) \wedge (f(x+1) - r\,x > 0.5) \Rightarrow (\lfloor f(x+1) \rfloor + 1 \equiv r\,x + 1)$$

This, and the other requirements, can be shown using the properties of floor and some real arithmetic. The basic idea is that, since the slope of the line is between 0 and 1, the next y-value, $r(x-1)$, must be either the same as the previous value, $r\,x$, or its successor, $r\,x+1$.

So, we have a recurrence relation for r, which depends on the value of $e\,x$. We now examine $e\,x$.

$$
\begin{array}{ll}
 & e\,x < 0 \\
\equiv & \quad \text{“From above derivation”} \\
 & f(x+1) - r\,x > 0.5 \\
\doteq & \quad \text{“Definition of } f \text{ (5.15)”} \\
 & y_1 + m * (x + 1 - x_1) - r\,x > 0.5 \\
\equiv & \quad \text{“}m = d_y/d_x\text{, multiply by } d_x\text{”} \\
 & d_x * y_1 + d_y * (x + 1 - x_1) - d_x * r\,x > 0.5 * d_x \\
\equiv & \quad \text{“arithmetic”} \\
 & 2 * d_x * r\,x + d_x - 2 * d_x * y_1 - 2 * d_y * (x + 1 - x_1) < 0
\end{array}
$$

So, we define:

$$e\,x \;\hat{=}\; 2 * d_x * r\,x + d_x - 2 * d_x * y_1 - 2 * d_y * (x + 1 - x_1) \tag{5.19}$$

The function e also satisfies a recurrence relation:

$$e(x+1) \;=\; e\,x + 2 * d_x * (r(x+1) - r\,x) - 2 * d_y \tag{5.20}$$

Note that this expression for e uses integer arithmetic only. We can now eliminate r from the recurrence relation for e. The difference between $r(x+1)$ and $r\,x$ is always either 0 or 1. So we have:

$$
\begin{array}{ll}
 & e(x+1) \\
= & \quad \text{“Recurrence Relation (5.20)”} \\
 & e\,x + 2 * d_x * (r(x+1) - r\,x) - 2 * d_y \\
\equiv & \quad \textbf{“Alternation Introduction, } \Delta(e\,x < 0)\textbf{, Alternation to Conditional”} \\
 & \textbf{if } e\,x < 0 \textbf{then } e\,x + 2 * d_x * (r(x+1) - r\,x) - 2 * d_y \\
 & \qquad\qquad \textbf{else } e\,x + 2 * d_x * (r(x+1) - r\,x) - 2 * d_y \\
\sqsubseteq & \quad \textbf{“Using Context in Guards}\text{, previous observations”} \\
 & \textbf{if } e\,x < 0 \textbf{ then } e\,x + 2 * d_x - 2 * d_y \textbf{ else } e\,x - 2 * d_y
\end{array}
$$

We know from definition (5.19) that $e\,x_1 = d_x - 2 * d_y$.

Now we have that the calculation of the next y-value, $r(x+1)$, depends on the previous y-value, $r\,x$ and the difference value $e\,x$. Therefore, at each iteration, we want to calculate $r(x+1)$ and the next difference value $e(x+1)$. Let us define a function $k : \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$ forming the pair:

$$k\,x \;\hat{=}\; (r\,x, e\,x) \tag{5.21}$$

and combine the two recurrence relations into one:

$$
\begin{aligned}
& (r(x+1), e(x+1)) \\
\equiv \quad & \text{“Recurrence Relations”} \\
& (\textbf{if } e\,x < 0 \textbf{ then } r\,x + 1 \textbf{ else } r\,x, \\
& \ \textbf{if } e\,x < 0 \textbf{ then } e\,x + 2 * d_x - 2 * d_y \textbf{ else } e\,x - 2 * d_y) \\
\equiv \quad & \text{“\textbf{Product Formation thru Conditional}”} \\
& \textbf{if } e\,x < 0 \textbf{ then } (r\,x + 1, e\,x + 2 * d_x - 2 * d_y) \textbf{ else } (r\,x, e\,x - 2 * d_y)
\end{aligned}
$$

Now we can use the **Map to Iterate** theorem with:

$$
\begin{aligned}
next \ & \hat{=} \ (\textbf{fun } r, e \in \mathbb{Z} : \textbf{if } e < 0 \ \textbf{then } (r\,x + 1, e\,x + 2 * d_x - 2 * d_y) \\
& \qquad\qquad\qquad\qquad\qquad\quad \textbf{else } (r\,x, e\,x - 2 * d_y)) \\
make \ & \hat{=} \ (\textbf{fun } x \in \mathbb{Z} : (r\,x, e\,x)) \\
use \ & \hat{=} \ \textsf{fst}
\end{aligned}
$$

which gives us, from our first specification (5.16):

$$
\begin{aligned}
& (round \circ f) * \langle x_1 \ldots x_2 \rangle \\
\equiv \quad & \text{“Definitions of } r \text{ and } k\text{, (5.18) and (5.21)”} \\
& (\textsf{fst} \circ k) * \langle x_1 \ldots x_2 \rangle \\
\equiv \quad & \text{“\textbf{Map to Iterate} theorem”} \\
& \textbf{let } next = (\textbf{fun } r, e \in \mathbb{Z} : \textbf{if } e < 0 \textbf{ then } (r\,x + 1, e\,x + 2 * d_x - 2 * d_y) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } (r\,x, e\,x - 2 * d_y)) \\
& \textbf{in } (take\ n \circ (\textsf{fst}*) \circ iterate\ next)(y_1, d_x - 2 * d_y)
\end{aligned}
$$

This implementation of specification (5.16) is efficient and uses only integer arithmetic. It corresponds to Bresenham's line drawing algorithm [17].

In [19] it is shown how an imperative version of this program can be obtained through further transformations which make use of the state monad.

5.6 Conclusions

In this chapter we have provided the apparatus for proving properties of and refining specifications of the language defined in chapter 2.

A proof system, consisting of the axioms of chapter 2, a number of inference rules and a method of writing down proofs have been provided as a means of proving true boolean expressions of the language. Using a deductive form of reasoning, proofs proceed by substituting equivalent terms, "substituting equals for equals". The basic axioms of the language are extended by a list of transformation laws, useful for manipulating specifications.

A goal of the refinement calculus is to supply the means of calculating a program P from a specification S. Usually we do not have that P and S are equivalent, but rather we have the relation that P implements or refines S. In this chapter we have introduced a new operator $\sqsubseteq$ into the language, so that $S \sqsubseteq P$ is equivalent to the boolean value *True* whenever P is a valid implementation or refinement of S. The operator $\sqsubseteq$ is transitive, allowing stepwise refinement. In addition, the majority of language constructs are monotonic with respect to refinement, meaning that piecewise refinement can occur.

For a small number of operations, including $=$, $\not\equiv$, Δ and δ, arguments may be replaced only with equivalent expressions, not by refined expressions. In practice, this is not a problem, but some care should be taken when refinement is piecewise. The non-monotonic operators are essential for specification and for reasoning, and the care taken during piecewise refinement is a small price to pay for their expressive power. The multiplication example provided an instance where reasoning about proper expressions, using Δ, was necessary.

The example manipulations and refinements in section 5.4 demonstrate how the calculus might be used. Using the example of the zip function, we showed how recursion can be introduced into a refinement. The refinement of the N-queens example showed both the introduction of recursion and how sets can be manipulated in the expression language. It also indicted where data refinement would be used.

The proofs associated with the printing control example demonstrate that laws of the proposed calculus can be used with larger specifications, reasoning equationally as before. Chapter 4 introduced the state monad with exceptions as a way of structuring large specifications, and this was shown to be useful in making specifications more readable. However, in section 5.4.3, we find that the use of monads make properties of specifications less easy to formulate. Although the monad laws can be added to the list of equivalence laws, it is likely that they would only be used to unfold the monad definitions, resulting in a purely functional specification which is then manipulated using the laws of the calculus. Therefore, in reasoning about large specifications, the use of monads does not provide any extra machinery, and may even hinder the formulation of expressions.

Finally, the example of Bresenham's line drawing algorithm shows how programs in an imperative style can be derived from functional specifications.

A refinement calculus for the development of functional programs has now been presented. This comprises the specification language of chapter 2, the refinement relation and the provision of a set of refinement laws – including basic axioms, the transformation laws and the laws of section 5.3.2. What remains is a justification of their validity in terms of a denotational semantics and proofs of the laws of the calculus. This is what is now addressed in chapter 6.

Chapter 6

Semantics

In this chapter we describe a denotational semantics for the expression language set out in chapters 2 and 5. The role of the semantics is to provide a model of the language which can be used to justify the axioms and rules of inference. This will show that the theory is consistent.

Other approaches to specification languages based on expressions have avoided the issue of semantics [68] or have given a semantics based on predicate transformers [90]. We take the approach, based on an example in [88], of mapping each expression of the language onto its set of possible values. An overview of the methodology and notation used is given in section 6.1. The semantic mapping is defined by structural induction in section 6.2.

The difficult problem of giving a semantics to recursive function definitions is tackled in section 6.3. This involves some applications of domain theory and, since our expressions denote sets, powerdomains in particular. We order the sets of our semantic domains using the Egli-Milner ordering, and apply the fixpoint theorem for monotonic functions to give a formal account of recursive functions in the specification language.

In section 6.4 we examine refinement of expressions and use the Smyth ordering to give a semantic definition of the relation. In section 6.5 we use the semantic definitions to show that the semantics supports the axioms of the language and the inference rules proposed in chapter 5.

Finally, section 6.6 describes informally how a denotational semantics might be given to the specification modules introduced in chapter 3.

6.1 Methodology

In chapters 2 and 5 an expression language was described formally though the use of type rules and axioms. In this chapter we give a semantic presentation of the language, mapping each expression to some set using structural induction. Our aim is to demonstrate that these sets, in the semantic domain, provide a good model for the axioms and laws of the expression language. In this section we give an informal overview of the mapping used.

The semantic mapping, which we call $\mathcal{M}$, maps an expression E to its set of possible evaluations. We call such sets $\mathcal{M}$-sets, and $\mathcal{M}\ E$ is called the $\mathcal{M}$-set of expression E.

Each type T of the expression language has an associated semantic domain D_T. Each D_T contains a 'least' element, $\perp_{D_T}$ which is associated with the undefined value of T, $\perp_T$, of the expression language. A more formal treatment of domains will be given in section 6.3 where the semantics of recursion is considered. For the semantics of non-recursive expressions, however, it is sufficient to identify domains with maximal typed sets.

For example, the associated domain for the type *Bool* is the lifted boolean domain $\mathbf{Bool}_\perp$, which contains the elements *True*, *False* and $\perp_{\mathbf{Bool}}$; has operators $\neg$, $\vee$, $\wedge$, $\Rightarrow$; as well as quantifiers $\forall$, $\exists$. These values, operators and quantifiers in the semantic domain are distinct from their counterparts in the expression language, although they are written using the same symbols.

The domain **Bool**, the domain $\mathbb{Z}$ and the domain **Char** are standard primitive domains of most versions of domain theory.

Undefinedness in the expression language is handled by using lifted domains, which always have a least element. Non-determinism in the expression language is handled by mapping expressions onto sets of possible evaluations which exist in the associated domain. So, our mapping $\mathcal{M}$, in general, takes a type T onto the powerset of its associated domain D_T, the powerdomain $\mathcal{P}D_T$. For example, the $\mathcal{M}$-set of a boolean expression is in the powerdomain $\mathcal{P}\,\mathbf{Bool}_\perp$, i.e. it is a set of elements from $\mathbf{Bool}_\perp$. The powerdomain structure will be explained in more detail in section 6.3.

A proper expression in the expression language will be mapped by $\mathcal{M}$ to a singleton set in the semantic domain. This makes sense because a proper expression has exactly one possible evaluation. e.g.

$$\mathcal{M}(\mathit{True}) = \{\mathit{True}\}$$

A non-deterministic expression will be mapped to a set containing at least two elements, since it has more than one possible evaluation. e.g.

$$\mathcal{M}(\mathit{True} \mathbin{[]} \mathit{False}) = \{\mathit{True}, \mathit{False}\}$$

So non-deterministic choice in the expression language is modelled by set union in the semantic domain.

An expression which is undefined in the expression language will be mapped to a set containing the least element of the associated domain. e.g.

$$\mathcal{M}(\mathit{True} \mathbin{[]} \bot_{\mathit{Bool}}) = \{\mathit{True}, \bot_{\mathbf{Bool}}\}$$

The meaning of the miraculous expression $\top$ is given by the empty set of the semantic domain. This is because it has no possible evaluations.

Intuitively, an expression E is well-defined if $\bot$ is not in its set of possible evaluations, *i.e.* $\bot \notin \mathcal{M}\,E$. An expression E is total if its set of possible evaluations is non-empty, *i.e.* $\mathcal{M}\,E \neq \emptyset$. If the $\mathcal{M}$-set of an expression E is a singleton set, then E is deterministic.

Strictness, for example of products, in the expression language will be modelled by taking the smash product of $\mathcal{M}$-sets. In a smash product domain $D_1 \otimes D_2$ there is no distinction between the pairs $(d_1, \bot_{D_2})$, $(\bot_{D_1}, d_2)$ and $\bot_{D_1 \otimes D_2}$, *i.e.* it is the strict product domain. The smash product operator, $\otimes$ will be explained in more detail in section 6.3.

Distribution of operators over operands in the expression language will be modelled by mapping the denotation of the operator over the $\mathcal{M}$-set of the operand. For example

$$\mathcal{M}(1 + (3 \mathbin{[]} 4)) = (+) * (\{1\} \otimes \{3, 4\})$$

which takes the smash products of the denotations of the operands (so enforcing strictness) and then maps the addition operator of the integer domain over the resulting set. This gives

$$\begin{aligned}\mathcal{M}(1 + (3 \mathbin{[]} 4)) &= (+) * (\{1\} \otimes \{3, 4\}) \\ &= (+) * \{(1, 3), (1, 4)\} \\ &= \{4, 5\}\end{aligned}$$

as expected.

Notation

In the following we will make use of a notation for set comprehensions borrowed from Wadler [88]. This is based on the list comprehension notation used in functional programming languages, as in [12]. We use this notation in order to distinguish the set comprehensions of the specification language from those in the semantic domain.

For S a singleton set in the semantic domain, we use ϵS to mean the single element of that set.

We define the shorthand notation $\text{cond}(c, S, T)$, where c is a condition and S and T are sets:

$$\text{cond}(c, S, T) \quad \hat{=} \quad \text{if } c \text{ then } S \text{ else } T$$

All of c, S, T here are objects in the semantic world, and not at the level of specifications.

Some nice properties of cond are the following:

$$\begin{aligned}
\text{cond}(\neg c, S, T) &= \text{cond}(c, T, S) \\
\text{cond}(c, S \cup S', T) &= \text{cond}(c, S, T) \cup \text{cond}(c, S', T) \\
\text{cond}(c \vee c', S, \emptyset) &= \text{cond}(c, S, \emptyset) \cup \text{cond}(c', S, \emptyset) \\
\text{cond}(c, \text{cond}(c', S, \emptyset), \emptyset) &= \text{cond}(c \wedge c', S, \emptyset)
\end{aligned}$$

We also have that, if from c we can deduce $S = S'$ then:

$$\text{cond}(c, S, T) \quad = \quad \text{cond}(c, S', T)$$

These properties will be used in proofs.

Notice here that we are talking about sets in the semantic domain, and hence equality $(=)$ is the usual equality of sets, not to be confused with the equality operator of the expression language. All conditions c are well-defined.

6.2 Semantics of Expressions

In this section we treat each expression of the specification language and describe its $\mathcal{M}$-set using structural induction. We begin with proper values of the types *Bool*, $\mathbb{Z}$ and *Char*. For v any such value:

$$\mathcal{M}\, v \quad = \quad \{\mathrm{v}\}$$

Here, the 'v' on the left is a value in the specification language, while that on the right, 'v' is the corresponding value from the associated semantic domain. In general the two will not be distinguished.

Examples of instances of this mapping are:

$$\mathcal{M}\ \mathit{True} = \{\mathit{True}\}$$
$$\mathcal{M}\ 3 = \{3\}$$
$$\mathcal{M}\ '\&' = \{'\&'\}$$

Undefinedness and Non-Determinism

The bottom expression is mapped onto the set containing the least element of the associated domain.

$$\mathcal{M}\perp_T \quad = \quad \{\perp_{D_T}\}$$

The miracle expression is mapped onto the empty set.

$$\mathcal{M}\top \quad = \quad \emptyset$$

The set of possible outcomes of an expression $E \,[\!]\, F$ contains the possible outcomes of E and the possible outcomes of F.

$$\mathcal{M}(E \,[\!]\, F) \quad = \quad \mathcal{M}\,E \cup \mathcal{M}\,F$$

So, if $\perp$ is a possible outcome of either E or F, then it is also in the set of possible outcomes for $E \,[\!]\, F$.

We now want to describe the $\mathcal{M}$-mappings for $\equiv$, δ and Δ. Consider a statement of the form $E \equiv F$ of the expression language. This should be *True* if $\mathcal{M}\,E$ and $\mathcal{M}\,F$ are the same, and *False* otherwise. But the denotation of the expression *True* is given by the set containing *True* in the semantic domain. Therefore the mapping for equivalence, $\equiv$, must be onto a (singleton) set.

The denotation of $\delta\,E$ should be the set $\{\mathit{True}\}$ if the $\mathcal{M}$-set of E contains the least element of the associated domain, and $\{\mathit{False}\}$ otherwise. The denotation of $\Delta\,E$ should be the set $\{\mathit{True}\}$ if the $\mathcal{M}$-set of E is a singleton set not containing the least element of the associated domain, and $\{\mathit{False}\}$ otherwise. Both $\mathcal{M}(\delta\,E)$ and $\mathcal{M}(\Delta\,E)$ should be singleton sets.

From the above analysis, we have the following mappings:

$$\begin{array}{rcl} \mathcal{M}(E \equiv F) & = & \{\mathcal{M}\,E = \mathcal{M}\,F\} \\ \mathcal{M}(\delta\,E) & = & \{\bot \notin \mathcal{M}\,E\} \\ \mathcal{M}(\Delta\,E) & = & \{\#\mathcal{M}\,E = 1 \wedge \mathcal{M}\,E \neq \{\bot\}\} \end{array}$$

The denotation for equality, $=$, does not necessarily result in a singleton set, since in the expression language equality distributes over choice, *e.g.*

$$((3 \mathbin{[]} 4) = (3 \mathbin{[]} 4)) \equiv (\mathit{True} \mathbin{[]} \mathit{False})$$

and is, in addition, strict. So, for equality, we have:

$$\mathcal{M}(E = F) \quad = \quad (=) * (\mathcal{M}\,E \otimes \mathcal{M}\,F)$$

This takes the $\mathcal{M}$-sets of E and F, forms all possible pairs and compares them, pairwise, for equality.

Semantics of Boolean Expressions

The $\mathcal{M}$-semantics for Boolean expressions are not very elegant, because most of the operators are not strict and do not distribute over choice. For negation, however, there is no problem

$$\mathcal{M}(\neg P) \quad = \quad (\neg) * \mathcal{M}\,P$$

where $\neg$ in the semantic domain is strict.

Possible outcomes for disjunction are given by extension

$$\begin{array}{lcl} \mathit{True} \in \mathcal{M}(P \vee Q) & = & \mathit{True} \in \mathcal{M}\,P \vee \mathit{True} \in \mathcal{M}\,Q \\ \mathit{False} \in \mathcal{M}(P \vee Q) & = & \mathit{False} \in \mathcal{M}\,P \wedge \mathit{False} \in \mathcal{M}\,Q \\ \bot \in \mathcal{M}(P \vee Q) & = & (\bot \in \mathcal{M}\,P \wedge \mathcal{M}\,Q \neq \{\mathit{True}\}) \\ & & \vee\,(\bot \in \mathcal{M}\,Q \wedge \mathcal{M}\,P \neq \{\mathit{True}\}) \end{array}$$

Notice that the boolean operators on the left of these equations are those of the specification language, while those on the right are part of the semantic language.

For example, consider the expression $(\mathit{True} \mathbin{[]} \mathit{False}) \vee \mathit{False}$. The $\mathcal{M}$-set of this expression must contain *True* because *True* is in the $\mathcal{M}$-set of the first disjunct; and it must contain

False because *False* is in the $\mathcal{M}$-sets of both disjuncts. It does not contain $\perp$ because $\perp$ is not in either of the $\mathcal{M}$-sets. We conclude that:

$$\mathcal{M}((\mathit{True} \mathbin{[\!]} \mathit{False}) \vee \mathit{False}) = \{\mathit{True}, \mathit{False}\}$$

Conjunction can be expressed in terms of negation and disjunction, while implication is expressed in terms of disjunction, negation and Δ. It turns out that the mapping for implication is the following:

$$\begin{array}{lcl}
\mathit{True} \in \mathcal{M}(P \Rightarrow Q) & = & \mathit{True} \in \mathcal{M}\,P \Rightarrow \mathit{True} \in \mathcal{M}\,Q \\
\mathit{False} \in \mathcal{M}(P \Rightarrow Q) & = & \mathcal{M}\,P = \{\mathit{True}\} \wedge \mathit{False} \in \mathcal{M}\,Q \\
\perp \in \mathcal{M}(P \vee Q) & = & \mathcal{M}\,P = \{\mathit{True}\} \wedge \perp \in \mathcal{M}\,Q
\end{array}$$

Again, this is given by extension. The mappings are included here because they will be used when we show that the Modus Ponens inference rule is valid in the model (see section 6.5).

Universal quantification is given by the following:

$$\begin{array}{rcl}
\mathit{True} \in \mathcal{M}(\forall\, x : T \mid P \bullet Q) & = & (\forall\, x : D_T \mid \mathit{True} \in \mathcal{M}\,P \bullet \mathit{True} \in \mathcal{M}\,Q) \\
\mathit{False} \in \mathcal{M}(\forall\, x : T \mid P \bullet Q) & = & (\exists\, x : D_T \mid \mathit{True} \in \mathcal{M}\,P \bullet \mathit{False} \in \mathcal{M}\,Q) \\
\perp \in \mathcal{M}(\forall\, x : T \mid P \bullet Q) & = & (\exists\, x : D_T \mid \mathit{True} \in \mathcal{M}\,P \bullet \perp \in \mathcal{M}\,Q) \\
 & & \wedge\, (\forall\, x : D_T \mid \mathit{True} \in \mathcal{M}\,P \bullet \mathcal{M}\,Q \neq \{\mathit{False}\})
\end{array}$$

And for existential quantification:

$$\begin{array}{rcl}
\mathit{True} \in \mathcal{M}(\exists\, x : T \mid P \bullet Q) & = & (\exists\, x : D_T \mid \mathit{True} \in \mathcal{M}\,P \bullet \mathit{True} \in \mathcal{M}\,Q) \\
\mathit{False} \in \mathcal{M}(\exists\, x : T \mid P \bullet Q) & = & (\forall\, x : D_T \mid \mathit{True} \in \mathcal{M}\,P \bullet \mathit{False} \in \mathcal{M}\,Q) \\
\perp \in \mathcal{M}(\exists\, x : T \mid P \bullet Q) & = & (\exists\, x : D_T \mid \mathit{True} \in \mathcal{M}\,P \bullet \perp \in \mathcal{M}\,Q) \\
 & & \wedge\, (\exists\, x : D_T \mid \mathit{True} \in \mathcal{M}\,P \bullet \mathcal{M}\,Q \neq \{\mathit{True}\})
\end{array}$$

We notice that the quantification on the left is that of the specification language, and hence three-valued, whereas that on the right is quantification in the semantic domain, and hence two-valued. Further, we notice that the x on the left is a variable identifier of the specification language, while that on the right is of the semantic language, which makes the predicates P and Q something of a hybrid. The intention is that x in the semantic language and $\mathcal{M}\,x$ for x in the specification language, should correspond.

Semantics of Integer Expressions

For operations over the integers, with $\oplus$ one of $+, -, \times, \sqcup, \sqcap$ and $\oslash$ one of $/$, *mod*, *div*, we have

$$\begin{aligned}
\mathcal{M}(E \oplus F) &= \oplus * (\mathcal{M}\,E \otimes \mathcal{M}\,F) \\
\mathcal{M}(E \oslash F) &= \oslash * (\mathcal{M}\,E \otimes \mathcal{M}\,F \backslash \{0\}) \cup \mathrm{cond}(0 \in \mathcal{M}\,F, \{\bot\}, \emptyset)
\end{aligned}$$

In the first case, we take the smash product, to enforce strictness, and then map the semantic function $(\oplus)$ over the set, which models distribution over choice. We assume that the application of $(\oplus)$ to $\bot_{\mathbb{Z}\times\mathbb{Z}}$ is $\bot_{\mathbb{Z}}$. In the second case we do the same thing, but remove zero as a possible divisor. Then, if zero is a possible outcome of F, we add $\bot_{\mathbb{Z}}$ to the resulting $\mathcal{M}$-set.

For example:

$$\begin{aligned}
\mathcal{M}(3/(3 \mathbin{[\,]} 0)) &= (/) * (\{3\} \otimes \{3,0\}\backslash\{0\}) \cup \mathrm{cond}(0 \in \{3,0\}, \{\bot_{\mathbb{Z}}\}, \emptyset) \\
&= (/) * (\{3\} \otimes \{3\}) \cup \{\bot_{\mathbb{Z}}\} \\
&= \{1, \bot_{\mathbb{Z}}\}
\end{aligned}$$

as required.

Semantics of Pairs

For pairs, again strictness is enforced by using the smash product. The associated operators are mapped over the resulting sets, modelling distribution. Note that the domain operators *fst* and *snd* are strict.

$$\begin{aligned}
\mathcal{M}(E, F) &= \mathcal{M}\,E \otimes \mathcal{M}\,F \\
\mathcal{M}(\mathbf{fst}\,p) &= \mathit{fst} * \mathcal{M}\,p \\
\mathcal{M}(\mathbf{snd}\,p) &= \mathit{snd} * \mathcal{M}\,p
\end{aligned}$$

For example:

$$\begin{aligned}
\mathcal{M}(\mathbf{fst}(3, 5 \mathbin{[\,]} \bot_{\mathbb{Z}})) &= \mathit{fst} * \mathcal{M}(3, 5 \mathbin{[\,]} \bot_{\mathbb{Z}}) \\
&= \mathit{fst} * (\{3\} \otimes \{5, \bot_{\mathbb{Z}}\}) \\
&= \mathit{fst} * \{(3,5), \bot_{\mathbb{Z}\times\mathbb{Z}}\} \\
&= \{3, \bot_{\mathbb{Z}}\}
\end{aligned}$$

as expected.

Semantics of Functions

The meaning of a function is given by the set of its possible graphs. Each graph is a set of pairs (x, y) where y is a possible value of the function at x, if it is defined and total, or y is $\perp$ if the value at x is undefined. Thus

$$\begin{array}{lcl} \mathit{graph}(\text{fun } x \in T : E) & \hat{=} & \{(a, b) \mid a \leftarrow D_T \backslash \{\perp_{D_T}\}, b \leftarrow \mathcal{M}(E[a/x])\} \\ \mathcal{M}(\textbf{fun } x \in T : E) & = & \{\mathit{graph}(\textbf{fun } x \in T : E)\} \end{array}$$

The $\mathcal{M}$-set of a deterministic function expression is a singleton set containing one graph. The domain of a graph g, $\mathit{dom}(g)$, is the set of all x's such that there is a pair (x, y) in g, i.e. the set of all x's that have a total value under the function given by g. The image of a value a in a graph g, $\mathit{Im}(a, g)$, is the set of possible values of the function given by g at a. For a set of values A and a set of graphs G, $\mathit{IM}(A, G)$ is the union of each $\mathit{Im}(a, g)$ for $a \in A$ and $g \in G$. For two graphs g_1 and g_2 such that the domain type of g_1 is the same as the result type of g_2, $\mathit{compose}(g_1, g_2)$ is as expected. We define

$$\begin{array}{lcl} \mathit{dom}(g) & \hat{=} & \mathit{fst} * g \\ \mathit{Im}(a, g) & \hat{=} & \text{cond}(a \neq \perp, \{b \mid (a, b) \leftarrow g\}, \_) \\ \mathit{IM}(A, G) & \hat{=} & \bigcup(\mathit{Im} * (A \times G)) \\ \mathit{compose}(g_1, g_2) & \hat{=} & \{(a, c) \mid (a, b) \leftarrow g_2, (b, c) \leftarrow g_1\} \end{array}$$

Properties of IM include

$$\begin{array}{lcl} \mathit{IM}(A \cup A', G) & = & \mathit{IM}(A, G) \cup \mathit{IM}(A', G) \\ \mathit{IM}(A, G \cup G') & = & \mathit{IM}(A, G) \cup \mathit{IM}(A, G') \end{array}$$

which will be useful in proofs. Now we have that the application of a function to an expression is obtained simply by looking up all the possible results in the corresponding graph(s). Function composition is obtained by mapping *compose* across the set of pairs of the corresponding graphs.

$$\begin{array}{lcl} \mathcal{M}(f\,E) & = & \mathit{IM}(\mathcal{M}\,E, \mathcal{M}\,f) \\ \mathcal{M}(f_1 \circ f_2) & = & \mathit{compose} * (\mathcal{M}\,f_1 \otimes \mathcal{M}\,f_2) \end{array}$$

Syntactically, a total function is one whose body is a total expression. Semantically, this condition is expressed as: for f a function of type $T \to T'$, f is a total function if

$$(\forall\, g \in \mathcal{M} f, a \in D_T : Im(a, g) \neq \emptyset)$$

or, equivalently

$$(\forall\, g \in \mathcal{M} f, a \in D_T : a \in dom(g))$$

According to the syntactic rules, the application $f\ E$ cannot be formed unless the function f is a total function.

Semantics of Generalised and Biased Choice

We've already seem that the choice operator is modelled by set union in the semantic domain, so any possible outcome of E or F will be a possible outcome of $E \mathbin{[\!]} F$. It follows that generalised choice over a set S will have S as its set of possible outcomes. Although we have not yet said what the meaning of a set expression is, we assert that $\bigcup \mathcal{M}\, S$ is the same as the set S. The $\mathcal{M}$-set for a biased choice is obtained by looking at $\mathcal{M}\, E$. If it is not empty, then E has a non-empty set of possible results (possibly including $\bot$) and must be total, *i.e.* $\mathcal{M}\, E \neq \emptyset$. In this case, the $\mathcal{M}$-set is just $\mathcal{M}\, E$. Otherwise we take $\mathcal{M}\, F$.

$$\begin{array}{lcl} \mathcal{M}([\!]/S) & = & \bigcup \mathcal{M}\, S \\ \mathcal{M}(E \overset{\leftarrow}{[\!]} F) & = & \mathrm{cond}(\mathcal{M}\, E \neq \emptyset, \mathcal{M}\, E, \mathcal{M}\, F) \end{array}$$

Notice that the only way infinite sets arise in the semantic domain is from the meaning of a generalised choice over an infinite set. This will be important in our treatment of recursive functions.

Semantics of Guards and Assumptions

The $\mathcal{M}$-set for a guarded expression $P \to E$ is a little more complicated, since there are three possibilities. If the guard is true, then the resulting $\mathcal{M}$-set is just $\mathcal{M}\, E$. If the guard is false, then the result should be non-total, *i.e.* the empty set. But if the guard is improper, then the resulting $\mathcal{M}$-set should contain just $\bot$. The $\mathcal{M}$-semantics for assumptions is similar, but they behave the same way whenever the assumption is non-true, giving an undefined result.

$$\begin{aligned}\mathcal{M}(P \to F) &= \text{cond}(\mathcal{M}\,P = \{\mathit{True}\}, \mathcal{M}\,F, \text{cond}(\mathcal{M}\,P = \{\mathit{False}\}, \emptyset, \{\bot\}))\\ \mathcal{M}(P \succ F) &= \text{cond}(\mathcal{M}\,P = \{\mathit{True}\}, \mathcal{M}\,F, \{\bot\})\end{aligned}$$

Semantics of Sets, Bags and Sequences

In order to simplify the semantics of the data structures sets, bags and sequences, we treat them, essentially, in the same way that simple values are treated. So, the $\mathcal{M}$-set of a set in the expression language, is a set of sets in the semantic domain. Similarly, a bag of the expression language is denoted by a set of bags, and a sequence in the expression language is denoted by a set of sequences in the semantic domain. We have, for sets

$$\begin{aligned}\mathcal{M}\{x \in T : P\} &= \{\{x \in D_T \backslash \{\bot_{D_T}\} : \mathcal{M}\,P = \{\mathit{True}\}\}\}\\ \mathcal{M}(\bigcup/A) &= (\bigcup/) * \mathcal{M}\,A\\ \mathcal{M}(x \subset A) &= (\in) * (\mathcal{M}\,x \otimes \mathcal{M}\,A)\end{aligned}$$

For bags

$$\begin{aligned}\mathcal{M}[\![x : T \ggg E]\!] &= \{[\![x : D_T \ggg a]\!] \mid a \leftarrow \mathcal{M}\,E\}\\ \mathcal{M}(B.E) &= \{b.a \mid b \leftarrow \mathcal{M}\,B, a \leftarrow \mathcal{M}\,E\}\end{aligned}$$

And for sequences

$$\begin{aligned}\mathcal{M}\langle i : I \ggg E\rangle &= \{\langle i : D_I \ggg a\rangle \mid a \leftarrow \mathcal{M}\,E\}\\ \mathcal{M}(\mathit{dom}\,S) &= (\mathit{dom}) * \mathcal{M}\,S\\ \mathcal{M}(S[j]) &= \{s[j'] \mid s \leftarrow \mathcal{M}\,S, j' \leftarrow \mathcal{M}\,j\}\end{aligned}$$

where D_I is the initial subset of the natural numbers in the semantic domain corresponding to the initial subset of the natural numbers I in the expression language.

6.3 Semantic Domains and Recursion

Our aim in this section is to give a meaning to recursive function expressions of the specification language. These are syntactically of the form

$\textbf{let } f = E[f] \textbf{ in } F[f]$

where f has type $A \to B$. Traditionally, the semantics of such a function is the least fixpoint of some functional in the semantic domain. Our goal, then, is to be able to apply the Fixpoint Theorem (theorem 2, to follow). This requires a theory of cpo's and monotonic functions such as can be found in any text on denotational semantics (*e.g.* [23, 35, 66, 74, 82, 86]).

6.3.1 Cpo's and Fixpoints

We assume the reader is familiar with the basic concepts of partial orders and partially ordered sets (posets), chains of elements from a poset, least upper bounds etc. We will usually write a partially ordered set using the notation $(D, \sqsubseteq)$ where D is a set of elements, and $\sqsubseteq$ is a partial ordering over D. If the ordering is obvious, we shall simply write D for the poset $(D, \sqsubseteq)$. In addition, the relation $\sqsubseteq_D$ may be used to represent the associated partial ordering for the set D. Subscripts may be dropped if the meaning is clear from the context. We now give a definition of a complete partial order.

Definition 1 *A partially ordered set $(D, \sqsubseteq)$ is a complete partial order, cpo, if every increasing chain of elements of D, $\langle d_n \rangle$, has a least upper bound (lub).*

Note that from this definition, since empty chains of elements of D have not been excluded, every cpo has a least element, written $\bot_D$, or $\bot$ if the subscript is obvious.

Every set X gives a *flat* cpo, $(X_\bot, \sqsubseteq)$, where $X_\bot \doteq X \cup \{\bot\}$ and $x \sqsubseteq y$ iff $x = \bot$ or $x = y$. Examples of such flat domains include $\mathbb{Z}_\bot$, $\mathbf{Bool}_\bot$ and $\mathbf{Char}_\bot$, which will henceforth be written without subscripts. A more interesting class of cpo is $(\mathbb{P}\,S, \subseteq)$ for any set S, the set of all subsets of S ordered by ordinary set inclusion. The least element of $\mathbb{P}\,S$ is the empty set, and the least upper bound operation is set union.

An important concept in the theory of fixpoints is that of a monotonic function.

Definition 2 *Let $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ be cpo's. A function $f : D \to E$ is monotonic iff, for every $x, y \in D$, if $x \sqsubseteq_D y$ then $f\,x \sqsubseteq_E f\,y$.*

The functions generally needed for the semantics of programming languages are continuous, *i.e.* they preserve limits of increasing chains.

Definition 3 *Let $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ be cpo's. A function $f : D \to E$ is continuous iff, for each chain $\langle d_n \rangle$ of elements of D, $f(\bigsqcup d_n) = \bigsqcup f\,d_n$.*

It should be clear that every continuous function is necessarily monotonic.

We have seen that any flat partial order is a cpo. Likewise any partial order, with a least element, which only has eventually constant increasing infinite chains, is also a cpo. In fact, all monotonic functions over such cpo's are continuous.

For any function $f : D \to D$, an element d of D is a fixpoint of f iff $f\ d = d$. Such a d is the least fixpoint if, for any other fixpoint d' of f, $d \sqsubseteq_D d'$. We now state the fixpoint theorem (see [47]).

Theorem 1 *Let $(D, \sqsubseteq)$ be a cpo with least element $\bot_D$. Every continuous function $f : D \to D$ has a least fixpoint which is $\bigsqcup f^n \bot_D$.*

This theorem is used widely to give denotational semantics to programming languages, particularly to iterative and recursive programming constructs. Domains with continuous functions provide denotations for almost all useful programming constructs. The exception, however, is unbounded non-determinism, which we use as a tool for specification rather than as part of the programming language. In this case we deal with monotonic, rather than continuous, functions. We use the fixpoint theorem for monotonic functions, stated in [50, 67] and attributed to Hitchcock and Park.

Theorem 2 *Let $(D, \sqsubseteq)$ be a cpo. Then for any monotonic mapping $f : D \to D$, the set of fixpoints of f contains a least element.*

A proof of this theorem can be found in [67]. The least fixpoint is given by $f^\alpha \bot_D$ for some ordinal α. So, unlike the case for continuous functions, the fixpoints of monotonic functions are not necessarily obtainable as the lubs of countable chains.

6.3.2 Domain Constructors

We have seen examples of some simple domains, such as the flat domains $\mathbb{Z}$, *Bool* and *Char*. New domains can be constructed using operators on domains. We look at some of the most common domain constructors here.

Products and Smash Products

Given two posets $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$, their product domain $(D \times E, \sqsubseteq_{D\times E})$ is the set of pairs (d, e) such that $d \in D$ and $e \in E$, partially ordered coordinatewise, *i.e.* $(d, e) \sqsubseteq_{D\times E} (d', e')$

iff $d \sqsubseteq_D d'$ and $e \sqsubseteq_E e'$. If D and E are cpo's, then so also is $D \times E$. Note that $\langle (x, y)_n \rangle$ is a chain in $D \times E$ iff $\langle x_n \rangle$ is a chain in D and $\langle y_n \rangle$ is a chain in E. Least upper bounds of chains in $D \times E$ are given by $\bigsqcup (x, y)_n = (\bigsqcup x_n, \bigsqcup y_n)$.

Given cpo's D, E and F, a function $f : D \times E \to F$ is continuous iff it is continuous in each of its arguments individually. This result can be extended to general products.

In the product domain $D \times E$ the pairs $(d, \perp_E)$ and $(\perp_D, e)$ are distinct, if $d \neq \perp_D$ or $e \neq \perp_E$. However, in the smash product $D \otimes E$ such pairs are identified with the least element of the domain, $\perp_{D \otimes E}$. The elements of $D \otimes E$ are those pairs $(d, e) \in D \times E$ such that $d \neq \perp_D$ and $e \neq \perp_E$, and the element $\perp_{D \otimes E}$. The ordering is coordinatewise, and $\perp_{D \otimes E}$ is the least element of $D \otimes E$. It follows that the smash product is a cpo since it has the same least element and the same lubs of increasing sequences as the Cartesian product. This makes $D \otimes E$ a subcpo of $D \times E$.

Note that $\otimes$ preserves the flatness of domains, *i.e.* if D and E are flat cpo's, then so is $D \otimes E$.

Function Spaces

For D a set and $(E, \sqsubseteq_E)$ a poset, their function space $(D \to E, \sqsubseteq_{D \to E})$ is the set of functions from D to E with the pointwise partial ordering $f \sqsubseteq_{D \to E} g$ iff $(\forall\, x \in D . f\, x \sqsubseteq_E g\, x)$. If $(E, \sqsubseteq_E)$ is a cpo, then so also is $(D \to E, \sqsubseteq_{D \to E})$, with lubs of increasing sequences given by $(\bigsqcup f_n) x = \bigsqcup (f_n\, x)$, and least element $(\lambda\, x \in D . \perp_E)$.

For D a set and $(E, \sqsubseteq_E)$ a poset, the function space $(D \xrightarrow{m} E, \sqsubseteq_{D \to E})$ is the set of monotonic functions from D to E with the pointwise partial ordering inherited from $(D \to E, \sqsubseteq_{D \to E})$. If $(E, \sqsubseteq_E)$ is a cpo, then so also is $(D \xrightarrow{m} E, \sqsubseteq_{D \to E})$. It is, in fact, a subcpo of $(D \to E, \sqsubseteq_{D \to E})$.

For D a set and $(E, \sqsubseteq_E)$ a poset, the strict function space $(D \to_\perp E, \sqsubseteq_{D \to E})$ is the set of strict functions from D to E with the pointwise ordering inherited from $(D \to E, \sqsubseteq_{D \to E})$. If $(E, \sqsubseteq_E)$ is a cpo, then so also is $(D \to_\perp E, \sqsubseteq_{D \to E})$, and it is a subcpo of $(D \to E, \sqsubseteq_{D \to E})$.

6.3.3 Semantic Domains

The domains we use to describe the semantics of the specification language include the basic flat domains $\mathbb{Z}$, **Bool** and **Char**. We also use smash products to represent pairs, and domains isomorphic to lifted strict function spaces for functions. To represent the sets of the specification language we use the flat powerset domain $(\mathbb{F}\, S, \sqsubseteq_{\mathbb{F}\, S})$, where the ordering

$\sqsubseteq_{\mathbb{P}S}$ is the usual flat ordering. However, we also require a powerdomain structure $\mathcal{P}D$ to represent the non-determinacy of the specification language. This is because each non-deterministic expression E, of type T, of the specification language, is represented by a set of possible values in the domain $\mathcal{P}D_T$, where D_T is the domain corresponding to type T. In the following section, we examine a suitable candidate for $\mathcal{P}D$.

6.3.4 The Egli-Milner Powerdomain

We have given a semantics for a non-deterministic specification language without recursion, and we now want to include the semantics for recursive function expressions. Since the semantic domains for the language are powersets, we need to find a definedness ordering on sets which will give us the cpo structure necessary for the existence of fixpoints.

For D a cpo, we want to form a powerdomain $\mathcal{P}D$ which is a cpo, with basic operations singleton and union. Clearly, the elements of $\mathcal{P}D$ should be those of $\mathbb{P}D$. We have already seen two orderings which can be associated with $\mathbb{P}D$, the flat ordering $\sqsubseteq_{\mathbb{P}D}$ and the subset ordering $\subseteq$. Neither of these are suitable orderings for $\mathcal{P}D$ since we require that singleton is monotonic, *i.e.* if $a \sqsubseteq_D b$ then $\{a\} \sqsubseteq_{\mathcal{P}D} \{b\}$, which is not the case in general with either of the orderings given. We shall see that the ordering we desire on sets is the Egli-Milner ordering.

The Egli-Milner Ordering

Let D be a domain. We take as elements of $\mathcal{P}D$ non-empty subsets of elements of D. Now, for A and B in $\mathcal{P}D$, the Egli-Milner ordering is given by:

$$A \sqsubseteq_{EM} B \quad \textit{iff} \quad (\forall\, x \in A.\, \exists\, y \in B.\, x \sqsubseteq_D y) \wedge (\forall\, y \in B.\, \exists\, x \in A.\, x \sqsubseteq_D y) \tag{6.1}$$

We argue that this ordering is appropriate for our needs. Each set in the semantic language denotes the set of possible evaluations for some expression. A set A can be made *more defined* by making some of its elements more defined, and without losing any information content. This gives the first part of the definition, $(\forall\, x \in A.\, \exists\, y \in B.\, x \sqsubseteq_D y)$. For the second part we note that no information which does not potentially already exist can be added to A, $(\forall\, y \in B.\, \exists\, x \in A.\, x \sqsubseteq_D y)$.

If D is flat, the definition can be restated as:

$$\begin{aligned} A \sqsubseteq_{EM} B \quad \textit{iff} \quad & \textit{either} \quad \bot \notin A \wedge A = B \\ & \textit{or} \quad \bot \in A \wedge A \backslash \{\bot\} \subseteq B \backslash \{\bot\} \end{aligned}$$

From this definition it should be clear that the set $\mathcal{P}D$ has a least element $\{\perp_D\}$, and if $A_0 \sqsubseteq_{EM} A_1 \sqsubseteq_{EM} \ldots$ is a non-empty increasing chain then either $\perp \notin A_n$ for some n, when $\bigsqcup_i A_i = A_n$, or $\perp \in A_n$ for all n, when $\bigsqcup_i A_i = \bigcup_i A_i$. It can easily be shown that $\sqsubseteq_{EM}$ is a partial ordering. We conclude that $\mathcal{P}D$, for D flat, together with the ordering $\sqsubseteq_{EM}$ is a cpo.

The singleton function $\{\cdot\} : D \to \mathcal{P}D$ is continuous, so $\bigsqcup\{a_n\} = \{\bigsqcup a_n\}$, as is expected. In addition, the binary union function $\bigcup : \mathcal{P}D \times \mathcal{P}D \to \mathcal{P}D$ is also continuous. This means that chains of sets can be described in terms of chains of singleton sets, and the lubs of chains of sets can be given in terms of lubs of chains of elements, since singleton is continuous. The empty set is a special case, which we consider later.

Treatments of the Egli-Milner powerdomain [35, 37, 74, 82, 84] take the powerdomain for flat D, $\mathcal{P}D$ to consist of all non-empty subsets of D which are either finite or contain $\perp$. This is explained by the fact that for any computable function which has the possibility of producing an infinite set of outcomes, non-termination is also a possibility. However, this is not true for a specification language, where unbounded non-determinacy without non-termination is possible.

Including infinite, non $\perp$-containing sets in $\mathcal{P}D$ does not affect the cpo structure. For example, including the set $\{0, 1, 2, .., n, ..\}$ in $\mathcal{P}\mathbb{Z}$ does not affect the cpo structure which already exists, and by the Egli-Milner ordering we have that

$$\{\perp_{\mathbb{Z}}, 0, 1, 2, .., n, ..\} \sqsubseteq_{EM} \{0, 1, 2, .., n, ..\}$$

In fact the set $\{0, 1, 2, .., n, ..\}$ is not related by the Egli-Milner ordering to any other non $\perp$-containing infinite set of integers.

However, allowing non $\perp$-containing sets in $\mathcal{P}D$ means that not every set can be obtained as the limit of a chain of finite sets. From the above example, the limit of the chain

$$\{\perp_{\mathbb{Z}}\} \sqsubseteq_{EM} \{\perp_{\mathbb{Z}}, 0\} \sqsubseteq_{EM} \{\perp_{\mathbb{Z}}, 0, 1\} \sqsubseteq_{EM} \ldots \sqsubseteq_{EM} \{\perp_{\mathbb{Z}}, 0, 1, 2, .., n\} \sqsubseteq_{EM} \ldots$$

is the infinite set $\{\perp_{\mathbb{Z}}, 0, 1, 2, .., n, ..\}$. It is impossible to construct a chain of finite sets which has $\{0, 1, 2, .., n, ..\}$ as its limit. Since the meaning of recursion will be given by the limit of a chain of sets, a non $\perp$-containing infinite set cannot be introduced as the result of recursion.

Adding the Empty Set

The Egli-Milner powerdomain, extended with infinite non $\perp$-containing sets, contains only non-empty sets. For an expression E of the specification language, the semantics of E is given by the set of possible evaluations of E. The empty set would denote the absence of a value for E, as in the case where E corresponds to the fictitious value $\top$. Therefore, we include the empty set $\emptyset$ in the ordering for a powerdomain $\mathcal{P}D$.

Following Heckmann [37] this can be achieved by simply including $\emptyset$ in the elements of the powerdomain, and extending the ordering $\sqsubseteq_{EM}$ so that $\{\perp\} \sqsubseteq_{EM} \emptyset$, and no other element of the powerdomain is comparable to $\emptyset$.

6.3.5 Recursive Function Definitions

The reason that we are looking at the powerdomain $\mathcal{P}D$ for a domain D is so that we can give meaning to recursive definitions. Such definitions are, syntactically, of the form:

let $f = E[f]$ **in** $F[f]$

where f is a function of type $A \to B$, say. Then the meaning of f will be given by the least fixpoint of a functional $\mathcal{F}$ over the domain $\mathcal{P}(A \to \mathcal{P}B)$. This exists, by theorem 2, provided that $\mathcal{F}$ is monotonic, i.e. $\mathcal{F}$ is in $\mathcal{P}(A \to \mathcal{P}B) \overset{m}{\to} \mathcal{P}(A \to \mathcal{P}B)$, and that $\mathcal{P}(A \to \mathcal{P}B)$ is a cpo. Using straightforward syntactic restrictions we can ensure that $\mathcal{F}$ is monotonic. Unfortunately, using the extension to the Egli-Milner powerdomain, as described above, we can only guarantee that $\mathcal{P}D$ is a cpo if we know that D is a flat domain.

We can, however, make some simplifications. First we insist that, in the definition for f, the expression E must be deterministic. This is a reasonable syntactic restriction which can be imposed easily. Since f is a function, this means that f must be *externally* deterministic, though it can have a non-deterministic body. The direct consequence of this restriction is that the meaning of f must be a singleton set in $\mathcal{P}(A \to \mathcal{P}B)$.

Using the fact that singleton is continuous, it follows that the meaning of f is the singleton set containing the least fixpoint of a monotonic functional $\mathcal{F}'$, which is in the domain $(A \to \mathcal{P}B) \overset{m}{\to} (A \to \mathcal{P}B)$. This, in turn, exists if $A \to \mathcal{P}B$ is a cpo. We saw in section 6.3.2 that this holds if $\mathcal{P}B$ is a cpo, which is true by the extension of the Egli-Milner powerdomain if B is flat.

We propose to restrict recursive function definitions to those of type $A \to B$ where the domain corresponding to the type B is flat. From section 6.3.3 it should be clear that the only non-flat semantic domains we use are function domains, or smash products involving function domains. This restriction of B to flat domains would rule out such recursive function definitions as:

$$\begin{array}{lcl} \mathbf{let}\ f & = & (\mathbf{fun}\ x \in \mathbb{Z} : \\ & & \quad \mathbf{if}\ x > 10 \rightarrow (\mathbf{fun}\ y \in \mathbb{Z} : x + y) \\ & & \quad \mathbf{else\ let}\ y = []/\mathbb{Z}\ \mathbf{in}\ f\ y) \\ \mathbf{in}\ f & & \end{array}$$

We do not consider this to be a serious restriction to the expressive power of the language.

It is, in fact, possible to remove the restriction by constructing a powerdomain similar to the Plotkin powerdomain [73, 74].

A Powerdomain for Non-Flat Domains

Let D be a domain. We want to form the powerdomain $\mathcal{P}D$ which has as elements sets of elements of D, with an ordering $\sqsubseteq$ making $\mathcal{P}D$ into a cpo, with continuous singleton and union operators. We know, from section 6.3.4 that the ordering should be based upon the Egli-Milner ordering:

$$A \sqsubseteq_{EM} B \quad \textit{iff} \quad (\forall\, x \in A.\exists\, y \in B.x \sqsubseteq_D y) \wedge (\forall\, y \in B.\exists\, x \in A.x \sqsubseteq_D y)$$

and we have seen that this is sufficient to give an appropriate $\mathcal{P}D$ when D is flat.

However, when D is not flat, two problems occur. The first is that $\sqsubseteq_{EM}$ is not a partial order, but a preorder, as can be seen from the example: if $a \sqsubseteq_D b \sqsubseteq_D c$ then, from the definition of the Egli-Milner ordering, we have

$$\{a, b, c\} \sqsubseteq_D \{a, c\} \quad \text{and} \quad \{a, c\} \sqsubseteq_D \{a, b, c\}$$

This problem could be solved quite easily by taking the quotient domain obtained by dividing out by the induced equivalence and ordering by $\sqsubseteq_{EM}$.

The second problem is that the union operator is not continuous. If $\langle x_n \rangle$ is a chain in D, then continuity of union would require that any set in $\mathcal{P}D$ containing $x_0, x_1, ..., x_n, ...$ should also contain $\bigsqcup x_n$. This is a problem because it means that infinite sets cannot be obtained by generalised union over finite sets.

Based upon the Plotkin construction [73, 74] of a powerdomain $\mathcal{P}D$, for D not necessarily flat, we form equivalence classes using a preorder $\precsim_{EM}$ which is based on $\sqsubseteq_{EM}$. The induced equivalence $\simeq_{EM}$ is such that, from the above examples, $\{a, c\} \simeq_{EM} \{a, b, c\}$, and any set containing $x_0, x_1, ..., x_n, ...$ is equivalent to one containing $\bigsqcup x_n$. Each equivalence class has a *biggest* element, which can be taken as the representative element of the class.

For X a non-empty set in $\mathcal{P}D$, the representative element of its equivalence class is denoted by its closure X^*, which is defined by:

$$X^* \mathrel{\hat{=}} \{y \mid (\exists\, x \in X . x \sqsubseteq y) \wedge (\forall\, b \in D . b \sqsubseteq y \Rightarrow \exists\, x \in X . b \sqsubseteq x)\}$$

These $(\cdot)^*$-closed subsets of D can now be ordered by the Egli-Milner ordering to give an appropriate powerdomain for our needs. So, we take

$$\mathcal{P}D \mathrel{\hat{=}} \langle (\cdot)^*\text{-closed non-empty subsets of } D, \sqsubseteq_{EM} \rangle$$

The empty set is added to $\mathcal{P}D$ using the Heckmann construction as described in section 6.3.4.

Plotkin's construction limits the subsets of D to those which are finitely-generable. This requirement is necessary for computation issues. However, we allow all sets, including those which are infinite and non $\bot$-containing. Our powerdomain agrees with the Plotkin powerdomain on finitely-generable sets.

The main consequence of allowing sets which are not finitely-generable is that some functions may no longer be continuous. In particular, for a continuous function $f : D \to E$, it may not be the case that the extension $f* : \mathcal{P}D \to \mathcal{P}E$ is continuous. It is the case, however, that $f*$ is monotonic, which is sufficient for the fixpoint theorem 2 for monotonic functions. Obviously, $f*$ is continuous over the Plotkin version of the domain $\mathcal{P}D$.

6.3.6 Semantics of Recursive Function Definitions

We now give the $\mathcal{M}$-semantics for expressions of the form, for $f : A \to B$,

$$\textbf{let } f = E[f] \textbf{ in } F[f]$$

From the above discussion we know that the meaning of f is a singleton set containing the least fixpoint of a certain functional. So, the meaning of the above expression should be the $\mathcal{M}$-set for expression F, which will depend on the $\mathcal{M}$-set for f, with occurrences of $\mathcal{M} f$

replaced by this singleton set.

$$\mathcal{M}(\textbf{let } f = E[f] \textbf{ in } F[f]) \quad = \quad \mathcal{M}\,F[\mathcal{M}f][\{\mu\,G\}/\mathcal{M}f]$$

where G is the functional for which we require a fixpoint.

From the discussion in section 6.3.5, the least fixpoint of this G is actually the single element of the set $\mathcal{M}f$, which we write $\epsilon\mathcal{M}f$. This should be given its meaning from $\epsilon\mathcal{M}\,E[\mathcal{M}f]$. Now, since for any singleton set S, $\{\epsilon S\} = S$, we conclude that the functional G should be defined as $G \mathrel{\hat{=}} \lambda\,g.\epsilon\mathcal{M}\,E[\{g\}]$.

6.4 Refinement

For expressions E and F, we want to give a semantics for refinement, written $E \sqsubseteq F$, with the intended meaning that expression E can be transformed into expression F such that every possible outcome of F is at least as defined as some possible outcome of E. This means that F must be at least as defined as E and should involve no more non-determinacy than E.

For E and F expressions of a simple type (corresponding to a flat domain) we expect that $E \sqsubseteq F$ iff $\bot$ is a possible outcome of E, or the set of possible outcomes of F is included in those of E, i.e.

$$E \sqsubseteq F \quad \textit{iff} \quad \bot \in \mathcal{M}\,E \vee \mathcal{M}\,E \supseteq \mathcal{M}\,F \tag{6.2}$$

as described by the general axiom for refinement, given in section 5.3. For example, we have $\bot \,[]\, 3 \sqsubseteq 5$ and $2 \,[]\, 3 \sqsubseteq 2$.

The refinement relation between expressions needs to be a preorder, *i.e.* have the properties of reflexivity and transitivity. However it will not be anti-symmetric since *e.g.* $\bot \,[]\, 2 \sqsubseteq \bot \,[]\, 5$ and $\bot \,[]\, 5 \sqsubseteq \bot \,[]\, 2$ but these expressions do not have equivalent semantics.

We find that a suitable definition for the refinement relation is based on the Smyth ordering for powerdomains [83].

6.4.1 The Smyth Ordering

The refinement relation between expressions of the specification language must be defined in terms of a relation at the semantic level. Because we have represented the nondeterminism of an expression by a set of possible values, we require a relation between sets in the powerdomain $\mathcal{P}D$, where D is the domain corresponding to the type of the expression. This relation, as already suggested, should be a preorder for $\mathcal{P}D$.

At the level of sets in $\mathcal{P}D$, the relationship we require is that set B "refines" set A iff everything in B is "better" or "more refined" than something in A. This means that refinement cannot add any information which was not already potentially present in A, but some of the information content in A can be lost, corresponding to a decrease in the nondeterminism of an expression. This intuitive notion is exactly the Smyth ordering for sets in $\mathcal{P}D$, first described in [83]:

$$A \sqsubseteq_S B \quad \hat{=} \quad \forall\, y \in B . \exists\, x \in A . x \leqslant_D y \tag{6.3}$$

where $\leqslant_D$ for the domain D will be defined in the following paragraph. The Smyth ordering corresponds exactly to the second half of the definition for the Egli-Milner ordering (6.1), which was used to form the powerdomain $\mathcal{P}D$.

We now define the ordering $\leqslant_D$ for a domain D by considering, in turn, each possible form that D may take:

- For a flat domain D, $\leqslant_D$ is exactly the definedness ordering $\sqsubseteq_D$.
- For a product domain $D \times E$ the ordering is coordinatewise:

  $$(d, e) \leqslant_{D \times E} (d', e') \text{ iff } d \leqslant_D d' \text{ and } e \leqslant_E e'$$

- For the smash product domain $D \otimes E$, the ordering $\leqslant_{D \otimes E}$ is also determined coordinatewise.
- For a function domain, which in our case will be of the form $D \to \mathcal{P}E$, the ordering is pointwise:

  $$f \leqslant_{D \to \mathcal{P}E} g \text{ iff } (\forall\, x \in D . f\, x \sqsubseteq_S g\, x)$$

Clearly, the ordering $\leqslant_D$ for each domain D is very similar to the definedness ordering $\sqsubseteq_D$. The only difference being that $\leqslant_{\mathcal{P}D}$ over a powerdomain is taken as the Smyth ordering $\sqsubseteq_S$, rather than the Egli-Milner ordering $\sqsubseteq_{EM}$ used for definedness.

We now use the Smyth ordering (6.3) to give a formal definition of the refinement relation for expressions.

6.4.2 Semantics of Refinement

Based on the Smyth ordering we give a semantics for the refinement relation between expressions of the specification language. For expressions E and F of the same type, with meanings $\mathcal{M}\,E$ and $\mathcal{M}\,F$, we define

$$\mathcal{M}(E \sqsubseteq F) \quad = \quad \{\mathcal{M}\,E \sqsubseteq_S \mathcal{M}\,F\}$$

We must now show that this definition agrees with the axioms given in section 5.3.

6.5 Soundness

We have now given a semantics to all aspects of the expression language. We now intend to demonstrate that this is a good semantics for the language, that it provides an adequate model for the axioms and laws of the language.

From the approach to the semantics, where each expression has been modelled by its set of possible evaluations, it follows quite easily that our axioms hold in the model. It is exactly this fact that we intend to demonstrate in the current section. Every axiom is an expression of type *Bool* and so has a meaning in the semantic domain $\mathcal{P}\mathbf{Bool}$. We are required to show that each axiom is mapped to the $\mathcal{M}$-set $\{\mathit{True}\}$. We will also need to show that the inference rules of the logic preserve truth in the $\mathcal{M}$-semantics.

Most axioms are of the form $E \equiv F$, which is given meaning in the semantic domain as $\{\mathcal{M}\,E = \mathcal{M}\,F\}$. Accordingly, in order to demonstrate the truth of the axiom, it suffices to show that $\mathcal{M}\,E = \mathcal{M}\,F$.

Some further axioms are of the form $P \Rightarrow Q$. In this case it suffices to show that $\mathcal{M}\,Q = \{\mathit{True}\}$ under the assumption that $\mathcal{M}\,P = \{\mathit{True}\}$.

Some proofs are very similar in how they progress, *e.g.* those which deal with distributivity of some operator over choice. In such cases we group the relevant axioms together and give the proof for just one representative axiom.

6.5.1 Undefinedness and Non-Determinism

The axioms for δ and Δ follow immediately from their semantic descriptions. To show the validity of an axiom of the form $\Delta\,E$, we just check that $\mathcal{M}\,E$ is a singleton set not containing $\bot$. To show the validity of an axiom of the form $\delta\, E$, we just check that $\bot \notin \mathcal{M}\, E$. Such proofs are trivial.

The basic properties of choice follow directly from the use of set union to model nondeterminism. For example, to show that the choice operator is commutative we have the proof as follows.

Commutativity of Choice

$$E \mathbin{[\!]} F \quad\equiv\quad F \mathbin{[\!]} E$$

Here it suffices to show that

$$\mathcal{M}(E \mathbin{[\!]} F) \quad=\quad \mathcal{M}(F \mathbin{[\!]} E)$$

Proof

$$\begin{array}{ll}
 & \mathcal{M}(E \mathbin{[\!]} F) \\
= & \quad \text{“Semantics”} \\
 & \mathcal{M}\, E \cup \mathcal{M}\, F \\
= & \quad \text{“Set union is commutative”} \\
 & \mathcal{M}\, F \cup \mathcal{M}\, E \\
= & \quad \text{“Semantics”} \\
 & \mathcal{M}(F \mathbin{[\!]} E)
\end{array}$$

□

The other basic properties, reflexivity and associativity, are equally trivial to show from the properties of set union.

We now show that the semantics supports the Δ axiom for $[\!]$.

Δ Property for Choice For E and F total

$$\Delta(E \mathbin{[\!]} F) \quad\equiv\quad \Delta\, E \wedge \Delta\, F \wedge (E \equiv F)$$

To prove this equivalence from the semantics, we need to prove

$$\mathcal{M}(\Delta(E \mathbin{[\!]} F)) \quad=\quad \mathcal{M}(\Delta\, E \wedge \Delta\, F \wedge (E \equiv F))$$

Proof From the semantics for Δ and for $\wedge$, if is enough to show

$$\#\mathcal{M}(E \,[\!]\, F) = 1 \wedge \bot \notin \mathcal{M}(E \,[\!]\, F)$$

is the same as

$$\#\mathcal{M}\,E = 1 \wedge \bot \notin \mathcal{M}\,E \wedge \#\mathcal{M}\,F = 1 \wedge \bot \notin \mathcal{M}\,F \wedge \mathcal{M}\,E = \mathcal{M}\,F$$

under the assumption that neither $\mathcal{M}\,E$ nor $\mathcal{M}\,F$ is empty. This is trivial.

□

The strictness property of $[\!]$ is supported by the following proof.

Strictness of Choice

$$\delta(E \,[\!]\, F) \quad = \quad \delta\,E \wedge \delta\,F$$

Here it is sufficient to show that

$$\bot \notin \mathcal{M}(E \,[\!]\, F) \quad = \quad \bot \notin \mathcal{M}\,E \wedge \bot \notin \mathcal{M}\,F$$

Proof

$$\begin{array}{ll}
 & \bot \notin \mathcal{M}(E \,[\!]\, F) \\
= & \quad \text{“Semantics”} \\
 & \bot \notin (\mathcal{M}\,E \cup \mathcal{M}\,F) \\
= & \quad \text{“Properties of set union”} \\
 & \bot \notin \mathcal{M}\,E \wedge \bot \notin \mathcal{M}\,F
\end{array}$$

□

6.5.2 The Equivalence Axioms

Equivalence in the expression language is modelled by equality of $\mathcal{M}$-sets. So the equivalence axioms follow immediately from properties of $=$ in the semantic domain. We give a representative proof.

Symmetric Equivalence

$$(E \equiv F) \;\equiv\; (F \equiv E)$$

Here we need to show that

$$\mathcal{M}(E \equiv F) \;=\; \mathcal{M}(F \equiv E)$$

Proof

$$\begin{array}{ll}
 & \mathcal{M}(E \equiv F) \\
= & \quad \text{“Semantics”} \\
 & \{\mathcal{M}\,E = \mathcal{M}\,F\} \\
= & \quad \text{“$=$ is symmetric”} \\
 & \{\mathcal{M}\,F = \mathcal{M}\,E\} \\
= & \quad \text{“Semantics”} \\
 & \mathcal{M}(F \equiv E)
\end{array}$$

□

6.5.3 Strictness Proofs

Many of the operators described in chapter 2.4 are strict, and there are a number of axioms which deal with strictness. Examples of these axioms are the following:

$\delta(E \oplus F) \Rightarrow \delta\,E \wedge \delta\,F$	integer operators
$\delta(E, F) \equiv \delta\,E \wedge \delta\,F$	product formation
$\delta(E \in A) \equiv \delta\,E \wedge \delta\,A$	set membership

Most axioms concerning strictness follow immediately from the use of smash products. Proofs of their validity are similar to each other, so there is no need to include them all here. As a representative example we have the following proof of the strictness of product formation.

Strictness of Product Formation For E and F expressions,

$$\delta(E, F) \;\equiv\; \delta\,E \wedge \delta\,F$$

In order to prove this, from the semantics for δ, it suffices to show that

$$\perp \notin \mathcal{M}(E,F) \quad = \quad \perp \notin \mathcal{M}\,E \wedge \perp \notin \mathcal{M}\,F$$

Proof

$$\begin{array}{ll}
 & \perp \notin \mathcal{M}(E,F) \\
= & \quad \text{“Semantics”} \\
 & \perp \notin \mathcal{M}\,E \otimes \mathcal{M}\,F \\
= & \quad \text{“Properties of smash products”} \\
 & \perp \notin \mathcal{M}\,E \wedge \perp \notin \mathcal{M}\,F
\end{array}$$

□

Strictness of function application is demonstrated by the following proof.

Strictness of Function Application

$$\delta(f\,E) \quad \Rightarrow \quad \delta\,f \wedge \delta\,E$$

Here it is sufficient to show that

$$\perp \notin \mathcal{M}\,f \wedge \perp \notin \mathcal{M}\,E \quad \Leftarrow \quad \perp \notin \mathcal{M}(f\,E)$$

Proof

$$\begin{array}{ll}
 & \perp \notin \mathcal{M}\,f \wedge \perp \notin \mathcal{M}\,E \\
= & \quad \text{“Set Theory”} \\
 & \forall\, e \in M\,E, g \in M\,f . e \neq \perp \wedge g \neq \perp \\
\Leftarrow & \quad \text{“Properties of graphs”} \\
 & \forall\, e \in M\,E, g \in M\,f . e \neq \perp \wedge \perp \notin \{b \mid (e,b) \leftarrow g\} \\
= & \quad \text{“Properties of cond”} \\
 & \forall\, e \in M\,E, g \in M\,f . \perp \notin \text{cond}(e \neq \perp, \{b \mid (e,b) \leftarrow g\}, \perp) \\
= & \quad \text{“Definition of } Im\text{”} \\
 & \forall\, e \in M\,E, g \in M\,f . \perp \notin Im(e,g) \\
= & \quad \text{“Set Theory”} \\
 & \perp \notin \bigcup(Im * (\mathcal{M}\,E \times \mathcal{M}\,f)) \\
= & \quad \text{“Definition of } IM\text{”}
\end{array}$$

$$
\begin{array}{ll}
 & \bot \notin IM(\mathcal{M}\,E, \mathcal{M}\,f) \\
= & \quad \text{“Semantics”} \\
 & \bot \notin \mathcal{M}(f\,E)
\end{array}
$$

□

6.5.4 Distribution

There are many axioms which describe the property of distribution over the choice operator. This is modelled in the semantics using map over sets. As in the case for the validation of strictness axioms, most of the axioms concerning distribution are shown to be valid in the model using a similar style of proof. A representative example is that of the distribution of function application over choice.

Distribution of Function Application over Choice

$$f(E \mathbin{[]} F) \;=\; f\,E \mathbin{[]} f\,F$$

Again, we need to show

$$\mathcal{M}(f(E \mathbin{[]} F)) \;=\; \mathcal{M}(f\,E \mathbin{[]} f\,F)$$

Proof

$$
\begin{array}{ll}
 & \mathcal{M}(f(E \mathbin{[]} F)) \\
= & \quad \text{“Semantics”} \\
 & IM(\mathcal{M}(E \mathbin{[]} F), \mathcal{M}\,f) \\
= & \quad \text{“Definition of } IM\text{”} \\
 & \bigcup(Im * (\mathcal{M}(E \mathbin{[]} F) \times \mathcal{M}\,f)) \\
= & \quad \text{“Semantics”} \\
 & \bigcup(Im * ((\mathcal{M}\,E \cup \mathcal{M}\,F) \times \mathcal{M}\,f)) \\
= & \quad \text{“Properties of } \times\text{”} \\
 & \bigcup(Im * ((\mathcal{M}\,E \times \mathcal{M}\,f) \cup (\mathcal{M}\,F \times \mathcal{M}\,f))) \\
= & \quad \text{“Properties of } *\text{”} \\
 & \bigcup(Im * (\mathcal{M}\,E \times \mathcal{M}\,f) \cup Im * (\mathcal{M}\,F \times \mathcal{M}\,f)) \\
= & \quad \text{“Distribute } Im*\text{”} \\
 & \bigcup(Im * (\mathcal{M}\,E \times \mathcal{M}\,f)) \cup \bigcup(Im * (\mathcal{M}\,F \times \mathcal{M}\,f))
\end{array}
$$

$$
\begin{array}{ll}
= & \text{“Definition of } \mathit{IM}\text{”} \\
 & \mathit{IM}(\mathcal{M}\,E, \mathcal{M}\,f) \cup \mathit{IM}(\mathcal{M}\,F, \mathcal{M}\,f) \\
= & \text{“Semantics”} \\
 & \mathcal{M}(f\,E) \cup \mathcal{M}(f\,F) \\
= & \text{“Semantics”} \\
 & \mathcal{M}(f\,E \mathbin{[\!]} f\,F)
\end{array}
$$

□

Other distribution axioms, such as

$(E \mathbin{[\!]} F) = G \equiv (E = G) \mathbin{[\!]} (F = G)$	equality
$(E \mathbin{[\!]} F) \oplus G \equiv (E \oplus G) \mathbin{[\!]} (F \oplus G)$	integer operations
$(E \mathbin{[\!]} F, G) \equiv (E, G) \mathbin{[\!]} (F, G)$	product formation
$(f \mathbin{[\!]} g)E \equiv f\,E \mathbin{[\!]} g\,E$	function application to the right
$E \in (A_1 \mathbin{[\!]} A_2) \equiv (E \in A_1) \mathbin{[\!]} (E \in A_2)$	set membership

will have similar proofs in the model.

6.5.5 Products and Functions

For the type constructors which form products and functions we demonstrate that the remaining axioms hold in the models we have given them.

A product type is modelled using the corresponding (smash) product domain. So, the axioms for proper products follow immediately. An example is the proof of one of the projection axioms.

Products For E and F expressions such that $\Delta\,E$ and $\Delta\,F$,

$$\mathbf{fst}(E, F) \;\equiv\; E$$

To prove this equivalence from the semantics, we need to prove

$$\mathcal{M}(\mathbf{fst}(E, F)) \;=\; \mathcal{M}\,E$$

using the fact that $\mathcal{M}\,E$ and $\mathcal{M}\,F$ are singleton sets not containing $\perp$.

Proof

$\mathcal{M}(\mathbf{fst}(E, F))$

$=$ "Semantics"

$fst * \mathcal{M}(E, F)$

$=$ "Semantics"

$fst * (\mathcal{M}\, E \otimes \mathcal{M}\, F)$

$=$ "$\mathcal{M}\, E$ and $\mathcal{M}\, F$ singleton sets"

$fst * (\{\epsilon\mathcal{M}\, E\} \otimes \{\epsilon\mathcal{M}\, F\})$

$=$ "Definition of $\otimes$, $\perp \notin \mathcal{M}\, E$, $\perp \notin \mathcal{M}\, F$"

$fst * \{(\epsilon\mathcal{M}\, E, \epsilon\mathcal{M}\, F)\}$

$=$ "Definition of $fst*$, $\perp \notin \mathcal{M}\, E$, $\perp \notin \mathcal{M}\, F$"

$\{\epsilon\mathcal{M}\, E\}$

$=$ "Definition of ϵ"

$\mathcal{M}\, E$

□

Other proofs for products are similar.

A function type is modelled using graphs, a common semantic model for functions. Again, the axioms for proper functions follow immediately from the properties of graphs. An example proof is that of function application by substitution.

Substitution If expression F has type T, such that $\Delta\, F$, then

$$(\mathbf{fun}\ x \in T : E)F \quad \equiv \quad E[F/x]$$

Again, we need to show that

$$\mathcal{M}((\mathbf{fun}\ x \in T : E)F) \quad = \quad \mathcal{M}(E[F/x])$$

using that $\mathcal{M}\, F$ is a singleton set not containing $\perp$.

Proof

$\mathcal{M}((\mathbf{fun}\ x \in T : E)F)$

$=$ "Semantics"

$IM(\mathcal{M}\, F, \mathcal{M}(\mathbf{fun}\ x \in T : E))$

$=$ "Semantics, with $g = graph(\mathbf{fun}\ x \in T : E)$"

$IM(\mathcal{M}\, F, \{g\})$

$=$ “$\mathcal{M}\,F$ a singleton set”

$IM(\{\epsilon\mathcal{M}\,F\},\{g\})$

$=$ “Definition of IM”

$\bigcup(Im * \{(\epsilon\mathcal{M}\,F, g)\})$

$=$ “Mapping over a singleton set”

$Im(\epsilon\mathcal{M}\,F, g)$

$=$ “Definition of Im, $\bot \notin \mathcal{M}\,F$”

$\{b \mid (\epsilon\mathcal{M}\,F, b) \leftarrow g\}$

$=$ “Definition of g, Set Theory, $\Delta\,F$”

$\{b \mid \epsilon\mathcal{M}\,F \leftarrow T\backslash\{\bot\}, b \leftarrow \mathcal{M}(E[F/x])\}$

$=$ “$\epsilon\mathcal{M}\,F \in T\backslash\{\bot\}$, Set Theory”

$\mathcal{M}(E[F/x])$

□

Other axioms for proper functions can be proved similarly.

Sets, bags and sequences are all mapped to flat powerset domains of their own associated domains, and so proofs of their axioms will also follow easily. We omit these proofs since they are tedious rather than interesting.

6.5.6 Assumptions and Guards

The axioms for assumptions and guarding follow directly from the semantics. For example, we show two of the axioms for assumptions.

True Assumption

$$\mathit{True} \succ\!\!- E \;\;=\;\; E$$

Here we need to show that

$$\mathcal{M}(\mathit{True} \succ\!\!- E) \;\;=\;\; \mathcal{M}\,E$$

Proof

$\mathcal{M}(\mathit{True} \succ\!\!- E)$

$=$ “Semantics”

$$\begin{array}{ll} & \text{cond}(\{\mathit{True}\} = \{\mathit{True}\}, \mathcal{M}\,E, \{\bot\}) \\ = & \quad \text{“Properties of cond”} \\ & \mathcal{M}\,E \end{array}$$

□

Improper Assumption

$$\neg\Delta\,P \quad\Rightarrow\quad (P \succ\!\!- E \equiv \bot)$$

Here we need to show that

$$\mathcal{M}(P \succ\!\!- E) \quad = \quad \mathcal{M}\,\bot$$

assuming that $\#\mathcal{M}\,P > 1 \vee \bot \in \mathcal{M}\,P$.

Proof

$$\begin{array}{ll} & \mathcal{M}(P \succ\!\!- E) \\ = & \quad \text{“Semantics”} \\ & \text{cond}(\mathcal{M}\,P = \{\mathit{True}\}, \mathcal{M}\,E, \{\bot\}) \\ = & \quad \text{“}\mathcal{M}\,P \neq \{\mathit{True}\}\text{, from assumption”} \\ & \{\bot\} \\ = & \quad \text{“Semantics”} \\ & \mathcal{M}\,\bot \end{array}$$

□

Similar proofs exist for the axioms of guarding.

6.5.7 Generalised Choice and Biased Choice

The axioms for generalised choice $[]/$ follow immediately from the semantics.

The axioms concerning biased choice are also easily proved, for example.

Biased Choice

$$(E \equiv \top) \;\;\Rightarrow\;\; (E \overset{\leftarrow}{[]} F \equiv F)$$

Here we need to show that

$$\mathcal{M}(E \overset{\leftarrow}{[]} F) \;\;=\;\; \mathcal{M}\, F$$

under the assumption that $\mathcal{M}\, E = \emptyset$.

Proof

$\mathcal{M}(E \overset{\leftarrow}{[]} F)$

$=$ "Semantics"

$\mathrm{cond}(\mathcal{M}\, E \neq \emptyset, \mathcal{M}\, E, \mathcal{M}\, F)$

$=$ "By assumption, $\mathcal{M}\, E = \emptyset$"

$\mathcal{M}\, F$

□

6.5.8 Recursion

Recursion Unfolding For recursive function definitions, we have the expected unfolding:

$$\Delta\, E \Rightarrow (\mathbf{let}\ f = E[f]\ \mathbf{in}\ F[f] \equiv F[E[(\mathbf{let}\ f = E[f]\ \mathbf{in}\ f)]])$$

Here, we need to show that

$$\mathcal{M}(\mathbf{let}\ f = E[f]\ \mathbf{in}\ F[f]) = \mathcal{M}(F[E[(\mathbf{let}\ f = E[f]\ \mathbf{in}\ f)]])$$

under the assumption that $\mathcal{M}\, E$ is a singleton set not containing $\perp$.

Proof

$\mathcal{M}(\mathbf{let}\ f = E[f]\ \mathrm{in}\ F[f])$

$=$ "Semantics, let $G = \lambda\, g.\epsilon \mathcal{M}\, F[\{g\}]$"

$\mathcal{M}\, F[\mathcal{M} f][\{\mu\, G\}/\mathcal{M} f]$

$=$ "Substitution"

$\mathcal{M}\, F[\{\mu\, G\}]$

$=$ "$\mu\, G$ a fixpoint of G"

$\mathcal{M}\, F[\{\epsilon \mathcal{M}\, E[\{\mu\, G\}]\}]$

$=$ "For any singleton set S, $\{\epsilon S\} = S$, and $\mathcal{M}\, E$ a singleton set"

$\mathcal{M}\, F[\mathcal{M}\, E[\{\mu\, G\}]]$

$=$ "Semantics"

$\mathcal{M}(F[E[(\textbf{let}\ f = E[f]\ \textbf{in}\ f)]])$

□

6.5.9 Refinement

In this section we show how the semantic definition of refinement supports the axioms proposed in section 5.3

Transitivity The transitivity of $\sqsubseteq$, follows immediately from the transitivity of $\sqsubseteq_S$.

General Refinement The general axiom for refinement, stated as

$$(E \sqsubseteq F) \Leftarrow \neg\delta\, E \vee (E \,[]\, F \equiv E)$$

we split into two parts.

Using the semantics, in order to show

$$(E \sqsubseteq F) \Leftarrow \neg\delta\, E$$

at the language level, we prove

$$\bot \in \mathcal{M}\, E \Rightarrow \mathcal{M}\, E \sqsubseteq_S \mathcal{M}\, F$$

at the semantic level.

Proof

$\mathcal{M}\, E \sqsubseteq_S \mathcal{M}\, F$

$=$ Definition of $\sqsubseteq_S$

$\forall y \in \mathcal{M}\, F. \exists x \in \mathcal{M}\, E. x \leqslant_D y$

$\Leftarrow$ Supply $\bot$ as a witness, $\bot \leqslant_D y$ for any y

$\forall y \in \mathcal{M}\, F. \bot \in \mathcal{M}\, E$

$=$ Logic

$\bot \in \mathcal{M}\, E$

To prove the second part of the axiom

$$(E \sqsubseteq F) \Leftarrow E \mathbin{[]} F \equiv E$$

at the language level, we prove

$$(\mathcal{M}\,E \cup \mathcal{M}\,F = \mathcal{M}\,E) \Rightarrow \mathcal{M}\,E \sqsubseteq_S \mathcal{M}\,F$$

at the semantic level.

Proof

$$\begin{array}{ll}
 & \mathcal{M}\,E \sqsubseteq_S \mathcal{M}\,F \\
= & \quad \text{Definition of } \sqsubseteq_S \\
 & \forall\, y \in \mathcal{M}\,F . \exists\, x \in \mathcal{M}\,E . x \leqslant_D y \\
\Leftarrow & \quad \text{Supply } y \text{ as a witness, } y \leqslant_D y \text{ for any } y \\
 & \forall\, y \in \mathcal{M}\,F . y \in \mathcal{M}\,E \\
= & \quad \text{Set Theory} \\
 & \mathcal{M}\,F \subseteq \mathcal{M}\,E \\
= & \quad \text{Set Theory} \\
 & \mathcal{M}\,E \cup \mathcal{M}\,F = \mathcal{M}\,E
\end{array}$$

□

In the case where $\mathcal{M}\,E$ and $\mathcal{M}\,F$ are sets over a flat domain, it is trivial to show that the axiom

$$(E \sqsubseteq F) \equiv \neg\delta\, E \vee (E \mathbin{[]} F \equiv E)$$

holds.

Refinement of Functions The axiom describing the refinement of proper functions is stated as

$$(\Delta f \wedge \Delta g) \Rightarrow (f \sqsubseteq g) \equiv (\forall\, x : T \mid \bullet f\, x \sqsubseteq g\, x)$$

We prove this by showing

$$(\forall\, x \in D_T . \mathcal{M}\,E \sqsubseteq_S \mathcal{M}\,F) = \mathcal{M}(\textbf{fun}\ x \in T : E) \sqsubseteq_S \mathcal{M}(\textbf{fun}\ x \in T : F)$$

Proof

$$\begin{array}{ll}
 & \mathcal{M}(\mathbf{fun}\ x \in T : E) \sqsubseteq_S \mathcal{M}(\mathbf{fun}\ x \in T : F) \\
= & \quad \text{“Semantics”} \\
 & \{\lambda\, x.\mathcal{M}\, E\} \sqsubseteq_S \{\lambda\, x.\mathcal{M}\, F\} \\
= & \quad \text{“Definition of } \sqsubseteq_S\text{”} \\
 & \lambda\, x.\mathcal{M}\, E \leqslant_{T \to_P T'} \lambda\, x.\mathcal{M}\, F \\
= & \quad \text{“Definition of } \sqsubseteq_S\text{”} \\
 & (\forall\, x \in D_T : (\lambda\, x.\mathcal{M}\, E)x \sqsubseteq_S (\lambda\, x.\mathcal{M}\, F)x) \\
= & \quad \text{“}\gamma\text{-Reduction in the } \lambda \text{ Calculus”} \\
 & (\forall\, x \in D_T : \mathcal{M}\, E \sqsubseteq_S \mathcal{M}\, F)
\end{array}$$

We conclude from this that

$$((\mathbf{fun}\ x \in T : E) \sqsubseteq (\mathbf{fun}\ x \in T : F)) \equiv (\forall\, x \in T : E \sqsubseteq F)$$

Now, since any function expression f which is proper must be of the form $(\mathbf{fun}\ x \in T : E)$, and using γ-reduction, we conclude that the axiom is also valid from the semantics of refinement.

□

Refinement of Choice The axiom for refinement of choice states, for $\Delta\, G$

$$(E \,[]\, F \sqsubseteq G) \equiv (E \sqsubseteq G \vee F \sqsubseteq G)$$

In the semantic domain this requires a proof that

$$(\mathcal{M}\, E \cup \mathcal{M}\, F \sqsubseteq_S \mathcal{M}\, G) = (\mathcal{M}\, E \sqsubseteq_S \mathcal{M}\, G) \vee (\mathcal{M}\, F \sqsubseteq_S \mathcal{M}\, G)$$

which is a trivial exercise, using the fact that $\mathcal{M}\, G$ is a singleton set.

Refinement of Generalised Choice The axiom regarding refinement of generalised choice was given as

$$([]/S \sqsubseteq E) \equiv (\exists\, x : T \mid x \in S \bullet x \sqsubseteq E)$$

for $\Delta\, E$ and $\Delta\, S$.

We give an overview of the proof that the semantics of refinement supports this axiom. *Proof* We need to show that

$$\mathcal{M}([]/S) \sqsubseteq_S \mathcal{M}\, E = (\mathcal{M}(\exists\, x : T \mid x \in S \bullet x \sqsubseteq E) = \{\mathit{True}\})$$

We know that $\mathcal{M}\, S$ and $\mathcal{M}\, E$ are singlton sets containing $\epsilon\mathcal{M}\, S$ and $\epsilon\mathcal{M}\, E$ respectively, which are non-bottom.

We take the left hand side and reason:

$$\begin{array}{ll}
 & \mathcal{M}([]/S) \sqsubseteq_S \mathcal{M}\, E \\
= & \quad \text{“Semantics, } \Delta\, E\text{”} \\
 & \bigcup \mathcal{M}\, S \sqsubseteq_S \{\epsilon\mathcal{M}\, E\} \\
= & \quad \text{“}\Delta\, S,\ \bigcup\{\epsilon\mathcal{M}\, S\} = \epsilon\mathcal{M}\, S\text{”} \\
 & \epsilon\mathcal{M}\, S \sqsubseteq_S \{\epsilon\mathcal{M}\, E\} \\
= & \quad \text{“Definition of } \sqsubseteq_S\text{”} \\
 & \forall\, y \in \{\epsilon\mathcal{M}\, E\}.\, \exists\, x \in \epsilon\mathcal{M}\, S.x \leqslant_D y \\
= & \quad \text{“Logic”} \\
 & \exists\, x \in \epsilon\mathcal{M}\, S.x \leqslant_D \epsilon\mathcal{M}\, E
\end{array}$$

Taking the right hand side, we obtain:

$$\begin{array}{ll}
 & \mathcal{M}(\exists\, x : T \mid x \in S \bullet x \sqsubseteq E) = \{\mathit{True}\} \\
= & \quad \text{“Set Theory, Semantics”} \\
 & (\exists\, x : D_T \mid \mathcal{M}(x \in S) = \{\mathit{True}\} \bullet \mathcal{M}(x \sqsubseteq E) = \{\mathit{True}\}) \\
= & \quad \text{“}\Delta\, S\text{, Semantics, Set Theory”} \\
 & (\exists\, x : D_T \mid x \in \epsilon\mathcal{M}\, S \bullet \{x\} \sqsubseteq_S \mathcal{M}\, E) \\
= & \quad \text{“Logic, Definition of } \sqsubseteq_S\text{”} \\
 & \exists\, x \in \epsilon\mathcal{M}\, S.\forall\, y \in \mathcal{M}\, E.\, \exists\, x' \in \{x\}.x' \leqslant_D y \\
= & \quad \text{“}\Delta\, E\text{, Logic”} \\
 & \exists\, x \in \epsilon\mathcal{M}\, S.x \leqslant_D \epsilon\mathcal{M}\, E
\end{array}$$

as required.

□

Refining $\top$ The final axiom is stated as

$$(\top \sqsubseteq E) = (E \equiv \top)$$

It is trivial to show that the semantics supports this.

6.5.10 Inference Rules

Our final task is to show that the inference rules of section 2.3.2 are valid. In fact, it is fairly standard to prove that these inference rules preserve truth in the $\mathcal{M}$-semantics.

For example, consider the Modus Ponens inference rule, given as:

$$\frac{P \quad P \Rightarrow Q}{Q}$$

We need to show that if both P and $P \Rightarrow Q$ are true in the model, for arbitrary P and Q, then it is necessarily the case that Q is true. Let us assume that $\mathcal{M}\, P = \{\mathit{True}\}$ and $\mathcal{M}(P \Rightarrow Q) = \{\mathit{True}\}$. Recall the mappings given for implication:

$$\begin{array}{lcl}
\mathit{True} \in \mathcal{M}(P \Rightarrow Q) & = & \mathit{True} \in \mathcal{M}\, P \Rightarrow \mathit{True} \in \mathcal{M}\, Q \\
\mathit{False} \in \mathcal{M}(P \Rightarrow Q) & = & \mathcal{M}\, P = \{\mathit{True}\} \wedge \mathit{False} \in \mathcal{M}\, Q \\
\perp \in \mathcal{M}(P \vee Q) & = & \mathcal{M}\, P = \{\mathit{True}\} \wedge \perp \in \mathcal{M}\, Q
\end{array}$$

From the first identity, and our assumptions, we conclude that $\mathit{True} \in \mathcal{M}\, Q$. From the second identity, since $\mathit{False} \notin \mathcal{M}(P \Rightarrow Q)$, we conclude that $\mathit{False} \notin \mathcal{M}\, Q$. Similarly, from the third identity we conclude that $\perp \notin \mathcal{M}\, Q$. And so we have $\mathcal{M}\, Q = \{\mathit{True}\}$.

The truth of the Generalisation inference rule is similar.

6.6 Semantics of Specification Modules

In section 3.2 we considered the form of a specification and said that a specification could either be a simple expression, or a collection of named expressions, possibly with user-defined types.

Simple specifications are just expressions, and so they have already been given a formal semantics.

We now consider what have been termed specification modules. These are collections of named expressions which may also contain given types, global constants and datatype definitions, as described in section 3.2.1.

Consider first a specification module with just a collection of specifications. This has the general form

$$\begin{aligned} name_1 &\mathrel{\hat{=}} E_1 \\ name_2 &\mathrel{\hat{=}} E_2 \\ &\vdots \\ name_n &\mathrel{\hat{=}} E_n \end{aligned}$$

We may assume that these are independent of each other, *i.e.* $name_i$ does not appear free in E_j for any i, j; otherwise make E_i a local definition of E_j, thereby binding $name_i$.

Now, each E_i has a denotation in the semantic domain, $\mathcal{M}\, E_i$. We say that the denotation of the specification module is a record, or collection of named denotations. The names in the semantic domain are derived from the corresponding names in the syntactic domain. So, the denotation of the above module would be something like:

$$\begin{aligned} [\quad & (\mathbf{name}_1, \mathcal{M}\, E_1), \\ & (\mathbf{name}_2, \mathcal{M}\, E_2), \\ & \qquad \vdots \\ & (\mathbf{name}_n, \mathcal{M}\, E_n), \quad] \end{aligned}$$

We now consider the case where the specification module contains a global constant, with the general form:

$$\begin{array}{l} \mid g : T \\ \mathit{Spec} \end{array}$$

The specification *Spec* already has a denotation which we call $\mathcal{M}\, \mathit{Spec}$. This contains occurrences of $\mathcal{M}\, g$ which is in the domain $\mathcal{P}\, D_T$, where D_T is the domain corresponding to type T. Now, g is a constant, so it should be denoted by a singleton set in $\mathcal{P}\, D_T$, of the form $\{m_g\}$, for some m_g in D_T. Finally, we say that the denotation of the specification module is a function from elements in D_T to denotations. This may be written as

$$\lambda\, m_g : D_T . \mathcal{M}\, \mathit{Spec}[\{m_g\} / \mathcal{M}\, g]$$

Now consider a specification module containing a given type. This is of the form

$$[T]$$

Spec

Again, the specification (module) *Spec* already has a denotation, $\mathcal{M}$ *Spec* which depends on a domain D_T corresponding to the given type T. We assume that this domain exists and that appropriate mappings exist, taking proper values of T to singleton sets in $\mathcal{P}\, D_T$. We don't know anything about the domain D_T except that it is distinct from any other domain that we know about. The denotation of the above specification module might be based on the use of existential parameter, representing the domain D_T, to the meaning of *Spec*.

Finally, we consider a specification module containing a datatype definition. This has the general form

$T ::= v_1 \mid v_2 \mid \ldots \mid v_n$
Spec

As before, we assume that the specification (module) *Spec* has the denotation $\mathcal{M}$ *Spec*, this time based on the domain D_T corresponding to the datatype T. In this case we want to associate D_T with the lifted domain containing the elements $\{\bot_T, \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$. These $n + 1$ values are distinct, and are such that $\mathbf{v}_i$ is the domain element associated with the proper value v_i, i.e. $\mathcal{M}\, v_i = \{\mathbf{v}_i\}$.

Clearly this account does not form a formal semantics for specification modules. However, it indicates that the problem of giving such a semantics does exist and suggests ways in which the problem might be overcome.

6.7 Conclusions

In this chapter we have given a formal semantics to the specification language based on sets of possible evaluations from some domain. In this way, the erratic non-determinism of an expression may be captured. The issue of undefined expressions is treated explicitly, by allowing these sets to contain the least element of the domain.

Since our semantic objects are sets, we use powerdomain theory to give a meaning to recursive function definitions. The sets are ordered using a variation of the Egli-Milner ordering. This extends work previously done with powerdomains, in that we admit infinite sets which do not contain $\bot$, the least element of the domain. We claim that this is appropriate for a specification language, since monotonicity, rather than continuity, is sufficient to allow the application of the fixpoint theorem. In a program, such infinite, non $\bot$-containing sets will not be a problem, since they can only arise from generalised choice over an infinite set.

Two expressions of the expression language are equivalent exactly when their $\mathcal{M}$-sets correspond. Therefore, in order to show the validity of the axioms of the language, with respect to the semantics, we have compared $\mathcal{M}$-sets for equality. Since the semantics of the language was structured with the axioms in mind, many of the axioms follow quite naturally, as demonstrated in section 6.5. The reasoning used in the semantic domain is semi-formal, as in the usual mathematical style for sets and domains.

The refinement relation has been given meaning using the Smyth ordering. We have shown that this supports the axioms for refinement given in chapter 5.

We find that the semantics based on sets of possible evaluations is a simple one, but sufficient for the requirements of an expression language. It has been possible to describe recursive functions adequately, and to reason easily about such functions. The definition of refinement is very clear, and the proofs of the refinement axioms are straightforward.

We have also suggested how a denotational semantics might be given to specification modules, informally introduced in chapter 3. This would involve records of denotations, and methods to construct new domains from their associated syntactic types. A discussion of a formal approach to modules is included in the next chapter.

Chapter 7

Discussion and Conclusions

In this chapter we summarise, review and discuss the main points of this thesis, the refinement calculus as it stands on its own, and how it contributes to the area of formal methods in computing science. Section 7.1 gives an overview of the thesis, indicating what was achieved and how it was approached. An evaluation is given in section 7.2.1 and section 7.2.2 looks at how the calculus might be used. Section 7.3 compares the results to similar work in the area of formal program development in general, and in the area of expression refinement in particular. Some suggestions for future work on the calculus are described in section 7.4.

7.1 A Refinement Calculus for Expressions

In chapter 1 we described what we consider to be the components and attributes of a refinement calculus, and indicated that it was our intention to describe such a calculus for expressions. Following the approach used for the imperative refinement calculus we defined a specification language of expressions which includes more expressive, though non-executable, constructs useful for making specifications. Special features include ways for reasoning with and about undefined terms; non-deterministic expressions to allow for more abstract specifications; and partial expressions to allow the piecewise construction of specifications. The expression language is described in chapter 2.

Chapter 3 shows how the expressions are used to form specifications. A specification is described as a collection of expressions which may include user defined types and global constants. A number of small examples demonstrate how the various concepts might be employed.

In chapter 4 we showed how the language could be used to describe larger problems, by introducing the concept of partial functions, which may be combined using special union operators to form complete specifications. These partial functions are essentially a syntactic device for the structuring of specifications into conceptual units. However, we also discussed how it might be possible to define a special class of higher-order functions to manipulate partial functions.

The use of monads in functional programming has proved a useful tool in the structuring of large programs, by hiding the details of impure features such as state and exceptions. In chapter 4 we showed how the state monad with exceptions can be used to structure specifications of our language, and we indicated how it might be possible to define monads within the language itself.

We demonstrated, in chapter 5, how properties of specifications can be formulated and how expressions can be manipulated and reasoned about, using a proof system based on the logic of the language itself. A refinement relation is introduced and we indicate how a specification can be refined, in a stepwise and piecewise manner. Collections of transformation and refinement laws are provided to support the high level manipulation of expressions without always appealing to the basic axioms.

We have given a formal semantics to expressions of our language, based on sets of possible evaluations, in chapter 6. The use of sets handles explicitly the possible non-determinism of expressions, while undefinedness is accommodated by allowing the least value of a domain as an element of a semantic set. Totality is given a meaning in terms of definedness and non-determinism. The semantics of recursion is given by ordering the sets using the Egli-Milner ordering and applying the fixpoint theorem.

Refinement is given meaning at the semantic level using the Smyth ordering for powerdomains, which displays the required properties. Using this and the semantic definition of equivalence, we have shown that the axioms and laws of the calculus are supported by the semantics. We consider that the proofs involved are straightforward.

7.2 Discussion

We discuss the refinement calculus described in this thesis in terms of an evaluation of its shortcomings and achievements and how the calculus might be used.

7.2.1 Evaluation

A logic which accomodates both undefined and non-deterministic terms has been described in section 2.3. The logic includes many of the laws of 2-valued logic, and it is possible to reason equationally about terms, in the style of [26, 32]. A similar logic, with $\bot$ and a demonic form of $[\!]$, is presented in [64, 65]. Our work extends this by providing axioms for terms of types other than *Bool*.

The inclusion of $\bot$ and $[\!]$ in the expression language, as described in chapter 2, results in an expressive specification language which has been shown to be useful in the formulation of specifications. The admission of non-deterministic expressions is not a new concept. However, our choice construct is slightly different from other approaches since it is both truly non-deterministic and erratic. The introduction of non-deterministic expressions results in more abstract specifications, giving more freedom at the implementation stage. The rich set of data types also adds to the expressiveness of the language, although one obvious omission is the ability to define recursive data types, such as trees.

The distinction between possibly undefined and possibly partial expressions is not usually so explicit. We have treated partiality as the dual of undefinedness with respect to refinement, since top '$\top$' is the identity for choice, so $\top \mathbin{[\!]} E \sqsupseteq E$, while bottom '$\bot$' acts like a zero for choice, since $\bot \mathbin{[\!]} E \sqsupseteq \bot$. The concept of partial expressions is useful since specifications can be built in parts, while each part may be manipulated and refined as a complete unit.

However, since partial expressions are not implementable, we found it necessary in section 2.6.2 to control the occurrences of potentially partial expressions in specifications. This means the introduction of an operator which can be used to totalise such expressions, the biased choice operator $\overleftarrow{[\!]}$. While this is a useful tool in specifications, it is not monotonic with respect to refinement, in general. This is not desirable, but any construct used to totalise expressions will necessarily not be monotonic. It would be more elegant to treat partiality in the same unrestricted way that we have treated undefinedness.

Again making use of partial expressions, we have extended the concept to partial functions, which are used purely as a syntactic device to structure specifications. This promotes the aim of separation of concerns in the construction of large specifications. The use of partial functions was demonstrated in chapter 4 with a specification of a printing control system. This also made use of some notational shorthands, such as detached parameters and record definitions, in order to make the specification more readable.

Partial functions are combined using the union operators '$\dot{\cup}$' and '$\overleftarrow{\cup}$', which both have a syntactic definition. The '$\dot{\cup}$' operator, in particular, can be compared to the disjunction operator used for schemas in the Z specification language. The syntactic definitions could be considered over-simplified, certainly when compared to the category theoretic approach of Back and Butler [2] or the relational approach of Frappier [30] to the composition of specifications. We have not considered any other ways of combining partial functions, such as a version of the conjunction operator.

The use of the state monad with exceptions to structure the printer control specification, in chapter 4, demonstrates how a large specification can be made more readable. We have also made some suggestions concerning how the definition of the monad might be included into the language, rather than simply being a syntactic device with some useful associated laws. However, as pointed out in section 5.4.3, the use of monads, even with the associated monad laws, doesn't make the specification any easier to reason about. In fact, it becomes more difficult to formulate properties about the specification, since a knowledge of the monad and how it works is required.

In chapter 6 we gave a semantics for the expression language based on sets. The resulting semantics is very simple. The approach to the construction of the semantic objects, as sets, means that most of the axioms of the language follow immediately. Where proofs are required, they are reasonably straightforward.

A lot of assumptions had to be made concerning recursive function definitions in order to give them a reasonable semantics. We only allow recursive functions which are deterministic at the outer level, but may have non-deterministic bodies. In addition, we restrict recursive function definitions to those of type $A \to B$ where the domain corresponding to the type B is flat. As described in section 6.3, these restrictions were necessary to allow the semantics based on powerdomains to be simplified. It would be interesting to allow general recursive expression definitions, which would certainly add to the expressiveness of the specification language.

7.2.2 Applications

The aim of this thesis is to describe a refinement calculus for expressions. We have provided a specification language based on expressions, a refinement relation and a set of refinement laws allowing the stepwise and piecewise refinement of expressions. There are a number of areas in which the results of the thesis could be applied.

It is possible that this work on the refinement of expressions could be used as an extension to the refinement calculus for imperative programs. As suggested by Morris [64, 65], by admitting non-determinacy at the level of expressions, not just at the statement level, this would permit the development of imperative programs using a methodology combining procedural and functional refinement. The specification language would be more expressive and, since expressions are easier to manipulate than statements, derivations could be much simplified.

Another application is that this work could form the basis of a refinement calculus for functional programs. As mentioned earlier, a program in a pure functional language is just an expression. Therefore, by making the target language of the calculus a functional programming language, it would be possible to calculate a functional program from an initial specifcation in the expression language. The data types of our language are quite rich and are not all present, or not easily implementable, in a functional programming language. This means that some form of data refinement would be necessary in a refinement calculus for functional programs. In addition, most functional languages have features such as polymorphism or laziness which do not form part of the expression language. We have discussed reasons why full polymorphism is not used in the language in section 2.5.7. Comments on laziness are given in section 5.5 and in section 7.3.3 when we compare our work with Bunkenburg's thesis.

In [18] Bunkenburg looks at how to transform expressions of a certain form into imperative style programs. Again using the fact that expressions are easier to manipulate than statements, the refinement rules of our calculus could be used to derive expressions of the required form before transforming to an imperative program. An example of the use of this approach is the derivation of Bresenham's line drawing algorithm in [19]. Part of this derivation was described in section 5.5 A simple mathematical specification of a line is refined, using the refinement calculus for expressions, to an expression of a certain form which is then transformed to an imperative style program. A similar technique is used in [69].

We claim that the specification language alone, described in chapters 2, 3 and 4, is a useful language for the construction of specifications for software. Like the Z specification language, it may be used to build specifications in the model-oriented approach, as demonstrated by the printer control specification of chapter 4. Even without using the refinement laws to derive a program, the resulting specifications can be reasoned about using the equivalence laws in the equational reasoning style.

7.3 Comparison to Other Work

In this section we compare our approaches and results to general formal program development techniques and also to other work carried out in the area of expression refinement. We first consider other approaches to reasoning with undefined and non-deterministic terms. We then look at other frameworks for the formal development of programs from specifications. Finally, we compare our calculus, in more detail, with the calculi of Norvell and Hehner [68], Ward [90], and Bunkenburg [18].

7.3.1 Approaches to Formal Reasoning

Our basic specification language, as defined in chapters 2 and 5, includes constructors for expressions which are possibly not well defined, non-deterministic or miraculous. In the logic, which is used to reason about expressions of the language, such problematic expressions are handled explicitly. We do not try to hide them, or pretend that they don't exist. We found that the miraculous expression top, $\top$, is difficult to reason with, and so it has a special treatment, as discussed in section 2.6.2. But for undefined expressions, $\bot$, and those involving choice, $[]$, axioms have been provided which cater for their occurrences. The aim is to retain as many of the usual axioms as possible, so that when all terms are well-defined and deterministic the logic reduces to classical logic.

There are many possible alternatives to the treatment of undefined expressions, as illustrated by the work of Cliff Jones in the area of handling partial functions [22, 42]. One approach is to attempt to keep to classical logic by restricting the domain of a function. In fact, we do this when we write the shorthand function

$$(\mathbf{fun}\ n \in \mathbb{N} : []/\{x \in \mathbb{Z} : x^2 \leqslant n < (x+1)^2\})$$

The intention is that the function is only ever applied to natural numbers, and never to a negative integer. However, there is no guarantee that the function won't be applied to such a negative argument since the type rules permit it. In our calculus the logic also tells us what happens when the function is applied to a negative integer, the result is the undefined integer, $\bot_{\mathbb{Z}}$.

The approach taken in the Z specification language [27, 75, 44] is to avoid function application entirely by treating functions as relations. This means, instead of writing $f\ x = y$, the function is treated as its graph and properties are forumlated by testing whether the pair (x, y) is a member of that graph. This has the advantage that it would also handle non-determinism quite easily. The disadvantage is that this approach leads to more complicated formulations of properties, making specifications more difficult to write.

Another approach is to use conditional forms of the familiar conjunction and disjunction operators, as in most programming languages. In evaluating an expression of the form

$P \wedge Q$, the left operand P is evaluated first. If it is *False*, then the whole expression is *False*. If it is *True*, then the result is the value of Q. If P is undefined, then the whole expression is undefined. Similarly for the disjunction operator. This approach is very implementation-oriented, and indeed our own conjunction and disjunction operators would probably be implemented (refined) in this way. However, for calculational purposes, these conditional operators have very unsatisfactory properties, the most obvious being that they are not symmetric.

The approach which we took was to treat the undefined value explicitly, using a logic close to classical logic. A similar approach is used in the logic of partial functions (LPF) [9] used for reasoning about specifications in VDM [40]. This uses non-strict extensions of the classical conjunction and disjunction operators (the same extensions as ours), and defines implication, as in classical logic, by

$$X \Rightarrow Y \equiv_{def} \neg X \vee Y$$

Unfortunately, this definition means that implication in LPF is not reflexive. We consider this to be a serious loss.

The implication defined in chapter 2 as

$$P \Rightarrow Q \equiv_{def} \neg P \vee \neg\Delta P \vee Q$$

is based on a definition from [1]. It was originally used in a three-valued version of the logic, but is also suitable for the seven values possible in our logic. This implication operator is reflexive and, although the bi-implication law

$$(P \equiv Q) \equiv (P \Rightarrow Q) \wedge (Q \Rightarrow P)$$

does not hold unless all terms are proper, many other laws of classical logic are retained. In particular, the deduction theorem holds. This says that in order to prove a theorem of the form $P \Rightarrow Q$, it is sufficient to prove Q under the assumption that P is available to us as a theorem.

The definition of implication aside, another way that our logic differs from LPF is that while LPF is three-valued, our logic also deals with non-deterministic logical values. Both Morris [65] and Bunkenburg [18] use a logic where terms may be non-deterministic. This logic has four distinct values, *True*, *False*, $\perp$ and *True* $[]$ *False*. The choice operator in this treatment is demonic, which makes $\perp$ a zero for choice.

The choice operator used in this thesis is erratic, giving seven values, *True*, *False*, $\perp$, *True* $[] \perp$, *False* $[] \perp$, *True* $[]$ *False* and *True* $[]$ *False* $[] \perp$. At first this may appear unnecessarily cumbersome, but in fact is not so difficult to work with since negation, disjunction and conjunction all distribute over choice. Of course, not all of the theorems of classical logic can be retained, but this directly follows from the fact that we are no longer in a two-valued world. When all logical terms are proper, our seven-valued logic reduces to classical logic.

7.3.2 Formal Program Development

Given a specification, the task of the programmer is to construct a program which implements that specification. Formal program development involves using rules and methodologies to develop a program in stages, with certain proof requirements at each step, such that the resulting program is guaranteed to satisfy the specification.

Program development methodologies for Z specifications are described in [27, 75]. The treatment of [75] involves a notion of refinement, of both data and operations. An abstract specification is refined in steps to a concrete specification which is suitable for "translation" into programming language code. Diller [27] describes how Z schemas can be transformed into formulae of a Floyd-Hoare logic, from which an implementation may be derived using the usual methods, e.g. [31, 45]. The weakness of such methodologies is in the gap between the final specification and the program. Since each is written in a different formal language, intermediate structures are necessarily hybrids. In particular, the last development step of [75] is an informal jump from specification to implementation.

The problem of having informal aspects in the development process is addressed, as in the imperative refinement calculus and in our own calculus, by having a specification language which is a superlanguage of a programming language. This is the case with the Extended ML specification language [79, 81, 80] which has as sublanguage the Standard ML programming language [71]. A methodology is provided which describes how a specification may be developed in stages by replacing non-algorithmic elements by executable code. At any stage in the development process there are three ways of proceeding – further decomposition of a problem into more manageable units; replace the special placeholder '?' by providing a functor body; or replace abstract code by a more 'algorithmic' version. Each way of proceeding is associated with a set of proof obligations.

The methodology for the development of programs from specifications in the Extended ML framework suffers from the problem that the process is still partly informal. The three general rules are expressed in an informal manner and, although they identify certain proof obligations associated with each type of step, the identification is done by observation. There is no mathematical notion of refinement between specifications.

In contrast, this thesis has attempted to follow the approach taken in the imperative refinement calculus [59, 56]. Both specification and program are expressed in the same language, in fact, we consider a program to be a special type of specification. A refinement relation is defined formally. Axioms and theorems are provided which allow properties of specifications to be rigorously demonstrated and programs to be formally calculated from specifications.

7.3.3 Refinement of Expressions

Other work in the area of refinement calculi for expressions includes that of Norvell and Hehner [68], Ward [90] and Bunkenburg [18], as discussed in section 1.2.2.

In the cases of [68] and [90], a simple language of expressions is extended with constructs for forming non-deterministic expressions, resulting in a specification language similar to that of chapter 2. In fact, the resulting calculi, consisting of language, refinement relation and rules for manipulation of expressions, are similar to that part of our calculus described in chapter 2 and parts of chapter 5. Our main contribution to the field of expression refinement, in comparison to these two pieces of work, is in two areas: we use partial expressions and partial functions to address the issues involved in structuring large specifications; and we give the specification language a simple denotational semantics. We now discuss these two issues, and then go on to compare our work with the thesis of Bunkenburg.

Large Specifications

The problem of using the specification language to make large specifications is not addressed in either of [68, 90]. We have shown, in section 2.6, how expressions may be combined using the choice operator, and, in section 4.1, how partial functions may be combined using a special union operator, to build large specifications in parts. The technique has been demonstrated in section 4.2. In addition, the use of the state and exception handling monad to structure large specifications has been examined, the results of which are found in section 4.3. The possibility of describing partial expressions and functions arises from the use of the unimplementable expression '$\top$', the unit of choice.

Norvell and Hehner's bunch union, corresponding to non-deterministic choice, has the *null* specification as unit, while the *magic* specification of Ward's language is the unit of demonic choice. Both of these specifications are unimplementable, and they correspond directly to our fictitious value ‘$\top$’. However, neither approach goes any further than admitting that this extreme specification exists.

For an under-determined choice operator, described in section 1.2.2, there would be no distinction between choice ‘$[\!]$’ and the special union operator ‘$\dot{\cup}$’ used to combine partial functions. This is the case with the bunch union operator of [68]. Our choice operator is such that, for partial functions f and g of the same type

$$f \mathbin{\dot{\cup}} g \;\sqsubseteq\; f \mathbin{[\!]} g$$

Like the demonic choice of [90], we can say that our choice operator is *truly non-deterministic*, making our function abstractions more expressive than those with an under-determined semantics.

Our biased choice operator ‘$\overleftarrow{[\!]}$’, used for totalising expressions, is very similar in nature to the non-commutative choice operator ‘$\boxtimes$’ introduced by Nelson in [67] as an extension to Dijkstra's calculus [24]. For A and B programs, the operational semantics of $A \boxtimes B$ is ‘activate A if possible, else activate B’. Nelson uses this choice operator with partial commands, which may be compared with our partial expressions. In the refinement calculus for imperative programs the unimplementable specification, *miracle* or *magic*, is also used to aid the formulation and refinement of specifications in parts.

Semantics

The semantics of Norvell and Hehner's specification language is given axiomatically. The refinement laws, while reasonable, are given without proof. In particular, the introduction of recursion in some of the example refinements is not given any formal basis.

Ward, in contrast, gives a semantics based on weakest preconditions to his language. The resulting semantics is over-complicated, and we are not convinced that such a semantics is necessary for a language based on expressions. Functions of the language only get a meaning when applied to something else, so, semantically speaking, they are not treated as first class citizens. In order to give a meaning to recursive functions in [90], the ordering used to obtain a least fixpoint is the refinement ordering, which is not usual in most treatments of recursion.

We consider that our approach to the denotational semantics of the specification language is more intuitively clear, and results in a much simpler semantics. While many of Ward's refinement laws are similar to those of chapter 5, our proofs are shorter and less complicated.

Comparison with Bunkenburg's Thesis

Bunkenburg's recent thesis [18] describes a calculus for the derivation of imperative style functional programs. In some ways, Bunkenburg's approach, content and findings are comparable to those of this work, but his thesis differs significantly in scope and in many of the design decisions taken. Bunkenburg states that the aim of his thesis is to present a formalism for calculating programs, including imperative programs. His main achievement is based on combining imperative threads with the easy calculational style of expressions, through the use of the state monad.

The scope of Bunkenburg's work is much broader than treated in this work. He starts with a description of an expression language, similar to that of chapter 2, and a discussion of refinement for this language. It is this part of [18] which can be directly compared with this thesis. However, Bunkenburg swiftly moves on to treat imperative expressions and also includes a brief description of data refinement techniques used with his language.

Bunkenburg also uses powerdomain theory to give a denotational semantics to the language, including the imperative style components. We will compare the denotational semantics of chapter 6 with Bunkenburg's treatment.

In the following we outline and discuss the differences between the expression language component of [18] and that presented in this thesis.

The first major difference in Bunkenburg's expression calculus is that function application is non-strict. In our approach we observe the property of strictness. Strictness of function application, as a specification tool, has the advantage that any value which becomes bound to a variable within the function body will be well-defined (and deterministic in our calculus). This has much value in terms of ease of calculation, without losing a significant amount of expressive power. For example, Bunkenburg's approach means that a function such as

$$(\textbf{fun}\ x \in \mathbb{Z} : \textbf{if}\ x \equiv \bot \to 3\ \overset{\leftarrow}{[]}\ 4\ \textbf{fi})$$

is a sensible one. In our calculus it is possible to prove, using the fact that $x \not\equiv \bot$ for any x, that this function is the same as the constant function

$$(\textbf{fun}\ x \in \mathbb{Z} : 4)$$

A second difference between the two pieces of work is in the data types provided and the treatment of objects of each type. Bunkenburg provides primitive types, sums, tuples, functions, sets, recursive and polymorphic types. In our calculus there are primitive types, tuples, functions, sets, bags and sequences. The three latter types allow specification in the model-oriented style and, in particular, admit infinite objects. Bunkenburg constructs lists which, he claims, are infinite. In fact, the axioms provided are for finite lists only. Bunkenburg approximates infinite lists because he has lazy constructors.

At the level of specification it is convenient to calculate with infinite objects, but such objects cannot be directly implemented. At this point, the implementation stage, infinite objects must be data-refined to finite objects. The use of lazy evaluation is a good way of approximating infinite objects. We suggest that it is more appropriate at the implementation stage than at specification level. It has been our experience that infinite sets, sequences and bags have been useful specification tools.

As described in chapter 1, Bunkenburg informally treats his expressions as upward closed sets of outcomes. An upward closed set is such that, if the set contains an outcome v, then it also contains all outcomes better (more defined) than v. In contrast, we treat an expression E such that, when evaluated, it may have a number of possible outcomes. We don't identify E with sets of possible outcomes. This is purely a semantic model.

A further difference between the calculi is in the treatment of non-determinism. Bunkenburg's choice operator, $\sqcap$, is interpreted as demonic non-determinism and axiomatised as the greatest lower bound operator for Bunkenburg's upward closed sets. This has a number of consequences.

First, refinement equivalence, $\sqsupseteq\!\sqsubseteq$, is the same as equivalence, $\equiv$. This means that fewer expressions can be distinguished in Bunkenburg's calculus.

His refinement is a partial order, since it is now anti-symmetric, in addition to being reflexive and transitive. However, in order to have a "good" refinement ordering, suitable for stepwise refinement, it is sufficient to produce a pre-order, such as our relation.

There are other minor differences between the two calculi including the treatment of guarded expressions. Bunkenburg's guarding operator, $\rightarrow$, is defined

$$\begin{aligned} &\mathit{True} \rightarrow E \equiv E \\ &G \rightarrow E \equiv \bot\text{, if } G \not\equiv \mathit{True} \end{aligned}$$

which makes alternations easier, but the $\rightarrow$ operator is no longer monotonic in its left argument.

Guarding is the only way that partiality can be introduced into a specification. Bunkenburg's specification expressions are of the form $\sqcap x : T.E$ (E with x bound to an arbitrary outcome of type T), which is always total. Although initially this seems less expressive than our $[]/S$ (choose an arbitrary element of sets S), the same can be expressed in Bunkenburg's calculus as $\sqcap x : T . x \in S \rightarrow x$.

Bunkenburg's denotational semantics uses the theory of powerdomains to provide a model for his language. His semantics is broadly similar to ours. Both use the Smyth ordering for refinement. However, because Bunkenburg's choice is demonic, he also uses the Smyth ordering for definedness, upon which the theory of recursion is based. Therefore, Bunkenburg's theory suffers from the same problem as Ward's, where the refinement ordering is used to find the least fixpoint. In some way, demonic choice would appear to blur the distinction between the refinement ordering and the definedness ordering. Bunkenburg gives a semantics for recursive function definitions, but not for more general recursive expression definitions, although he allows these in the specification language.

Finally, although Bunkenburg starts with an expression language similar to that described in chapters 2 and 5 of this thesis, he does not treat the language in as thorough a manner as is presented here. We have attempted to investigate fully the behaviour of possibly undefined, non-deterministic and partial expressions in a rigorous manner. In contrast, Bunkenburg uses the language more as a starting point to which is added imperative style constructs. It is his treatment of imperative expressions which forms the major component of this thesis.

7.4 Future Work

In this section we look at some possible areas for future extensions to the work presented in the thesis.

Non-Deterministic Boolean Expressions

Although the use of non-deterministic boolean expressions is not encouraged, since they are unlikely to be of any use in specifications, they cannot be eliminated.

We investigate the behaviour of non-deterministic boolean expressions as guards or assumptions. For example, consider the expression

$$\begin{array}{l}\textbf{let}\ f = (\textbf{fun}\ x \in \mathbb{Z} : x + 3\ []\ x - 3) \\ \&\quad n = 2 \\ \textbf{in}\ \ (f\ n > 0 \rightarrow E_1)\ \overleftarrow{[]}\ E_2\end{array}$$

The guard in the subexpression ($f\ n > 0 \rightarrow E_1$) is non-deterministic and is equivalent to $\mathit{True} \,[]\, \mathit{False}$. Using our axioms for guarding, and since $\neg\Delta(\mathit{True} \,[]\, \mathit{False})$ the resulting expression is undefined.

A different axiomatisation for guards (and assumptions) replaces the axiom for non-proper guards (assumptions) with a strictness axiom and a distribution axiom.

$$\begin{aligned} &\bot_{Bool} \rightarrowtail E \equiv \bot_T \\ &(P_1 \,[]\, P_2) \rightarrowtail E = (P_1 \rightarrowtail E) \,[]\, (P_2 \rightarrowtail E) \end{aligned}$$

where '$\rightarrowtail$' represents either '$\rightarrow$' or '$\succ$' throughout the formula, and T is the type of E.

Now the subexpression becomes

$$\begin{array}{ll} & f\ n > 0 \rightarrow E_1 \\ \equiv & \quad \text{"Manipulation of Guard"} \\ & (\mathit{True} \,[]\, \mathit{False}) \rightarrow E_1 \\ \equiv & \quad \text{"Left-distribute } \rightarrow \text{"} \\ & \mathit{True} \rightarrow E_1 \,[]\, \mathit{False} \rightarrow E_1 \\ \equiv & \quad \textbf{"Axioms for Guarding"} \\ & E_1 \end{array}$$

In this, we could say that $[]$ in guarding is, in some sense, angelic with respect to $\top$, since it is $\top$-avoiding. Evaluation of the guarded expression *looks ahead* to determine which choice of guard gives a total result.

With assumptions we have the less interesting case that

$$\begin{array}{ll} & f\ n > 0 \succ E_1 \\ \equiv & \quad \text{"Manipulation of Assumption"} \\ & (\mathit{True} \,[]\, \mathit{False}) \succ E_1 \\ \equiv & \quad \text{"Left-distribute } \succ \text{"} \\ & \mathit{True} \succ E_1 \,[]\, \mathit{False} \succ E_1 \\ \equiv & \quad \textbf{"Axioms for Assumptions"} \\ & E_1 \,[]\, \bot \\ \sqsupseteq & \quad \textbf{"Reduce Non-Determinacy, Introduce Choice, } \bot \sqsubseteq \bot \textbf{ and } \bot \sqsubseteq E_1\text{"} \\ & \bot \end{array}$$

So, in this case, we could say that $[]$ in an assumption is demonic with respect to $\bot$ and $\sqsupseteq$, *i.e.* $\bot$-seeking in terms of refinement equivalence.

Partial Functions

As discussed in sections 1.3 and 7.2.1, there has been some interesting work carried out on how to describe specifications in parts, and how to combine these parts to form complete specifications. We allow the formation of partial functions as abstractions over partial expressions, and combine them using a union operator, which is defined syntactically. This operator is similar to the disjunction operator used to combine schemas in Z. There also exists a conjunction operator for schemas in Z. We consider how a corresponding intersection operator might be used in our language.

In chapter 1.3 we used partial functions to specify different cases of a problem. These are then combined, using the union operator, such that

$$(\textbf{fun}\ x \in T : P \to E)\ \dot{\cup}\ (\textbf{fun}\ x \in T : \neg P \to F)$$

is equivalent to

$$(\textbf{fun}\ x \in T : P \to E \mathbin{[\!]} \neg P \to F)$$

Given the two specification expressions

$$[\!]/\{x \in \mathbb{Z} : 0 \leqslant x \leqslant 20\} \qquad [\!]/\{x \in \mathbb{Z} : \text{even}\, x\}$$

an intersection of the two specifications should result in the expression

$$[\!]/\{x \in \mathbb{Z} : (0 \leqslant x \leqslant 20) \wedge \text{even}\, x\}$$

Apart from investigating whether or not such a facility would be useful, it would also be interesting to see if a suitable syntactic definition could be given in the language. Such an operator, among others, is described by Frappier in [30] using a relational approach. The main concern is that either of the two specification expressions could be refined to such a point that the intersection no longer exists, resulting in an unimplementable specification.

In section 4.1.3 we looked briefly at the manipulation of partial functions, and suggested a special class of higher-order functions which might be defined for this purpose. We could also examine the behaviour of partial functions when applied to non-deterministic arguments. For example, consider the following expression, which is not syntactically correct according to our syntax restrictions.

$$(\textbf{fun}\ x \in \mathbb{Z} : x = 0 \to E)(0 \parallel 1) \tag{7.1}$$

Since function application distributes over choice, it is reasonable to assume that this should be the same as

$$(\mathbf{fun}\ x \in \mathbb{Z} : x = 0 \to E)0 \mathbin{[]} (\mathbf{fun}\ x \in \mathbb{Z} : x = 0 \to E)1$$

Function application with deterministic arguments is governed by the substitution rule, giving

$$0 = 0 \to E \mathbin{[]} 1 = 0 \to E$$

which, according to our equivalence laws, is just E. So, we could say that the evaluation of expression (7.1) *looks ahead* to determine which choice, if any, gives a total result. Similarly, we expect the expression

$$(\mathbf{fun}\ x \in \mathbb{Z} : x = 0 \to E)([]/\mathbb{Z})$$

to behave in the same way. This could prove to be a very useful property of the application of partial functions to non-deterministic arguments. In contrast, we note that the (total) expression

$$(\mathbf{fun}\ x \in \mathbb{Z} : x = 0 \succ\!\!- E)(0 \mathbin{[]} 1)$$

will evaluate to $E \mathbin{[]} \bot$.

In this thesis we have restricted the occurrences of partial functions, in order to simplify the tasks involved in describing the calculus. A study of the unrestricted behaviour of these functions could provide some interesting results.

Non-Deterministic Functions

The choice operator of our expression language is such that function abstraction does not distribute over $[]$. This, as we have seen, results in true non-determinism [90], *i.e.*

$$(\mathbf{fun}\ x \in T : E \mathbin{[]} F) \not\equiv (\mathbf{fun}\ x \in T : E) \mathbin{[]} (\mathbf{fun}\ x \in T : F)$$

Although the function on the right is a refinement of that on the left, the two may be distinguished from each other. Not only is the function on the left proper, while that on the right is improper, but they are also distinguishable when passed as arguments to a higher-order function such as map.

We now consider the function expression

$$(\mathbf{fun}\ x \in T : E \mathbin{[\!]} F) \mathbin{[\!]} (\mathbf{fun}\ x \in T : E)$$

and compare it to

$$(\mathbf{fun}\ x \in T : E \mathbin{[\!]} F)$$

It is reasonable to consider that these two functions should be equivalent since, operationally, there is no observable difference between them. In fact, they are refinement equivalent, $\sqsupseteq\!\sqsubseteq$, and can be distinguished from each other only by using the operator Δ.

This operator is defined over $[\!]$, in chapter 2, by the axiom

$$\Delta(E \mathbin{[\!]} F) \equiv \Delta\, E \wedge \Delta\, F \wedge (E \equiv F)$$

An alternative axiom might be

$$\Delta(E \mathbin{[\!]} F) \equiv (\Delta\, E \wedge E \sqsubseteq F) \vee (\Delta\, F \wedge F \sqsubseteq E)$$

With this axiom, both of the functions in question would be proper and so impossible to distinguish from each other. In fact, it would be possible to prove equivalence, using extensionality.

If this alternative axiomatisation for Δ was to be used, the semantic definitions described in chapter 6 would require to be revised. Currently they support the axiom for Δ as included in chapter 2, and a proof of this is given in section 6.5. However, it would be useful to explore the possibilities offered by the new axiomatisation, and to find a definition in the semantic domain to support it.

Program Transformations

In this thesis we have looked at the derivation of programs from specifications, but we have not considered the issue of efficiency. It is likely that a functional program derived using this calculus will not be the most efficient of implementations. However, there are techniques for the transformation of inefficient functional programs into equivalent but efficient programs. It should be possible to prove such transformations using our equivalence laws, or to describe the transformation techniques using our syntax and use the semantic definitions to prove them.

Data Refinement

The specification language of chapter 2 contains a rich set of data types, which are not present, or not easily implementable, in a pure functional programming language. The whole point of using the model-oriented approach for specification is to model some concept using these rich, but well-understood, types. However, it is not usually possible to use these types in the implementation.

Although we can refine expressions using our calculus, refinements are always between expressions of the same type. In order to change the type of an expression, data refinement methods are needed [60, 61, 58], as described in section 1.2.1. We anticipate that the same methods as are used for data refinement of imperative style specifications could be applied to functional style specifications. Bunkenburg outlines such an approach in his thesis [18].

Module Refinement

It is possible to give a formal syntax for modules using ideas drawn from algebraic specifications, object-oriented programming, type theory *etc.* [28, 36, 49, 51]. Although we did not take such an approach, because we found it was not necessary to achieve our goals, there are a number of reasons for a more formal approach. Modularisation of a large system (of specifications or implementations) has the commonly associated benefits of seperation of concerns and re-use of components.

A formal module syntax would provide the basis of a formal module calculus. Operations over modules, such as module inclusion, union and difference could be formally defined and investigated (see [10]). We could imagine the usefulness of building a hierarchy of modules, and employing the concepts of inheritance and specialisation, moving towards an object-oriented approach. More interesting might be the consideration of parameterisation of a module, with respect to values, types and even other modules (see [80]). Finally, and importantly in a refinement calculus, we could consider the possibility of one module refining another, using both expression and data refinement. It is likely that such refinement of a module would be with respect to some notion of an interface, containing invariants and other necessary information.

Mechanisation

Tools for the refinement of specifications based on the refinement calculus for imperative programs are currently being developed, for example, the work of Grundy [33, 34] using the

HOL theorem prover. An interesting exercise would be to attempt to build such a tool for our calculus for expressions. The embedding of the semantics of the language would be a huge task. However, if we were to incorporate the methods for expression refinement into the imperative refinement calculus, as suggested in section 7.2.2, the framework provided by the theorem prover could be of enormous benefit.

7.5 Final Remarks

This thesis has investigated an approach to deriving executable expressions from specifications using a refinement calculus, in the same manner as the refinement calculus for imperative programs. In this way, the calculus could be used to extend the refinement calculus to allow the refinement of non-deterministic expressions in specifications. It could also be used to form the basis of a refinement calculus for functional programs, or to derive imperative style programs from functional specifications. The calculus consists of: a specification language of expressions based on a general expression language; a refinement relation with properties to allow the stepwise and piecewise refinement of expressions; and a set of laws which can be used in the manipulation of a specification, the derivation of a program, or in the proof of a property of a specification. We consider the main contributions to the area, as well as the calculus itself, to be the approach taken to constructing large specifications using partial expressions and functions, and the denotational semantics which is based on the idea of sets of possible evaluations.

Appendix A

Theorems of the Logic

In this appendix we list some theorems of the logic as described in chapter 2.

A.1 Theorems of Propositional Logic

Distribution of $\vee$ Disjunction distributes over itself.

$$P \vee (Q \vee R) \equiv (P \vee Q) \vee (P \vee R)$$

Involution Negation is an involution.

$$\neg\neg P \equiv P$$

Properties of Δ An equivalence is always proper.

$$\Delta(E \equiv F)$$

$$\Delta\Delta P$$

De Morgan Conjunction and disjunction satisfy de Morgan's laws.

$$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$$

$$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$$

Conjunction Conjunction satisfies the usual properties.

$$P \wedge Q \equiv Q \wedge P$$

$$P \wedge (Q \wedge R) \equiv (P \wedge Q) \wedge R$$

$$P \wedge P \equiv P$$

$$P \wedge (Q \wedge R) \equiv (P \wedge Q) \wedge (P \wedge R)$$

Absorption The absorption laws for conjunction and disjunction.

$$P \wedge (P \vee Q) \equiv P$$

$$P \vee (P \wedge Q) \equiv P$$

Identities *True* is an identity for conjunction, and *False* is an identity for disjunction.

$$P \wedge \mathit{True} \equiv P$$

$$P \vee \mathit{False} \equiv P$$

Properties of $\Rightarrow$ Implication is reflexive and trichotomous. *False* is least with respect to the implication ordering, and *True* is greatest.

$$P \Rightarrow P$$

$$(P \Rightarrow Q) \vee (Q \Rightarrow P)$$

$$\mathit{False} \Rightarrow Q$$

$$P \Rightarrow \mathit{True}$$

Substitution The substitution rule for conjunction and for implication.

$$(P \equiv Q) \wedge E(P) \equiv (P \equiv Q) \wedge E(Q)$$

$$(P \equiv Q) \Rightarrow E(P) \equiv (P \equiv Q) \Rightarrow E(Q)$$

Modus Ponens

$$P \wedge (P \Rightarrow Q) \Rightarrow Q$$

Conjunction and Implication Conjunction is a greatest lower bound with respect to implication.

$$(P \Rightarrow Q) \wedge (P \Rightarrow R) \equiv (P \Rightarrow Q \wedge R)$$

Absorption We have two further absorption laws, concerning implication.

$$P \Rightarrow P \vee Q$$

$$P \wedge Q \Rightarrow P$$

Shunting The shunting law holds.

$$P \wedge Q \Rightarrow R \equiv P \Rightarrow (Q \Rightarrow R)$$

Transitivity and Monotonicity Implication is transitive. It is monotonic in its second argument, and antimonotonic in its first (wrt implication).

$$(P \Rightarrow Q) \wedge (Q \Rightarrow R) \Rightarrow (P \Rightarrow R)$$

$$(P \Rightarrow Q) \Rightarrow ((R \Rightarrow P) \Rightarrow (R \Rightarrow Q))$$

$$(P \Rightarrow Q) \Rightarrow ((Q \Rightarrow R) \Rightarrow (P \Rightarrow R))$$

Conjunction Mono wrt Implication Conjunction is monotonic with respect to implication.

$$(P \Rightarrow Q) \Rightarrow ((P \wedge R) \Rightarrow (Q \wedge R))$$

Disjunction and Implication Disjunction is a least upper bound with respect to implication, and satisfies certain monotonicity properties.

$$(P \Rightarrow R) \wedge (Q \Rightarrow R) \equiv (P \vee Q \Rightarrow R)$$

$$(P \Rightarrow Q) \Rightarrow ((P \vee R) \Rightarrow (Q \vee R))$$

Distribution of Implication Implication left-distributes over disjunction, over equivalence, and over itself.

$$P \Rightarrow Q \vee R \equiv (P \Rightarrow Q) \vee (P \Rightarrow R)$$

$$P \Rightarrow (Q \Rightarrow R) \equiv (P \Rightarrow Q) \Rightarrow (P \Rightarrow R)$$

$$P \Rightarrow (Q \equiv R) \equiv (P \Rightarrow Q) \equiv (P \Rightarrow R)$$

Properties of Nonequivalence Nonequivalence is symmetric.

$$P \not\equiv Q \equiv Q \not\equiv P$$

A.1.1 Laws Depending on Proper Values

Excluded Middle If P is proper, then the law of the excluded middle holds.

$$\Delta\, P \Rightarrow (P \vee \neg P)$$

Negation and Equivalence These are related by the law

$$(\Delta\, P \wedge \Delta\, Q) \Rightarrow (\neg(P \equiv Q) \equiv (\neg P \equiv Q))$$

Associativity of Equivalence Equivalence is associative for proper boolean terms.

$$(\Delta\, P \wedge \Delta\, Q \wedge \Delta\, R) \Rightarrow (((P \equiv Q) \equiv R) \equiv (P \equiv (Q \equiv R)))$$

Distribution over Equivalence

$$\Delta\, P \Rightarrow (P \vee (Q \equiv R) \equiv ((P \vee Q) \equiv (P \vee R)))$$

$$\Delta\, P \Rightarrow (P \Rightarrow (Q \equiv R) \equiv ((P \wedge Q) \equiv (P \wedge R)))$$

$$\Delta\, P \Rightarrow (P \wedge (Q \not\equiv R) \equiv ((P \wedge Q) \not\equiv (P \wedge R)))$$

Golden Implication

$$(\Delta\, P \wedge \Delta\, Q) \Rightarrow (P \Rightarrow Q \equiv (P \wedge Q \equiv P))$$

Bi-Implication

$$(\Delta\, P \wedge \Delta\, Q) \Rightarrow ((P \Rightarrow Q) \wedge (Q \Rightarrow P) \equiv (P \equiv Q))$$

Conjunction Absorption We can simplify the following conjunctions.

$$\Delta\, P \Rightarrow (P \wedge (P \Rightarrow Q) \equiv P \wedge Q)$$

$$(\Delta\, P \wedge \Delta\, Q) \Rightarrow (P \wedge (P \equiv Q) \equiv P \wedge Q)$$

$$\Delta\, P \Rightarrow (P \wedge (\neg P \vee Q) \equiv P \wedge Q)$$

Exchange Laws

$$(\Delta P \wedge \Delta Q) \Rightarrow (P \Rightarrow \neg Q \equiv Q \Rightarrow \neg P)$$

$$(\Delta P \wedge \Delta Q) \Rightarrow (\neg P \Rightarrow Q \equiv \neg Q \Rightarrow P)$$

Contrapositive

$$(\Delta P \wedge \Delta Q) \Rightarrow ((P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P))$$

$$(\Delta P \wedge \Delta Q) \Rightarrow (P \Rightarrow Q \equiv \neg Q \Rightarrow \neg P)$$

Associativity of Nonequivalence Nonequivalence is associative for well-defined terms, and equivalence and nonequivalence are mutually associative.

$$(\Delta P \wedge \Delta Q \wedge \Delta R) \Rightarrow (((P \not\equiv Q) \not\equiv R) \equiv (P \not\equiv (Q \not\equiv R)))$$

$$(\Delta P \wedge \Delta Q \wedge \Delta R) \Rightarrow (((P \not\equiv Q) \equiv R) \equiv (P \not\equiv (Q \equiv R)))$$

$$(\Delta P \wedge \Delta Q \wedge \Delta R) \Rightarrow (((P \equiv Q) \not\equiv R) \equiv (P \equiv (Q \not\equiv R)))$$

A.2 Theorems of Predicate Logic

Trading Theorems

$$\Delta P \Rightarrow ((\forall x : T \mid P \bullet Q) \equiv (\forall x : T \mid \bullet \neg P \vee Q))$$

$$(\forall x : T \mid P \wedge R \bullet Q) \equiv (\forall x : T \mid P \bullet R \Rightarrow Q)$$

$$\Delta R \Rightarrow ((\forall x : T \mid P \wedge R \bullet Q) \equiv (\forall x : T \mid P \bullet \neg R \vee Q))$$

Further Distribution Provided x is not free in Q,

$$\Delta P \Rightarrow ((\forall x : T \mid P \bullet Q) \equiv Q \vee (\forall x : T \mid \bullet \neg P))$$

Distribution Provided x is not free in Q, and $\neg(\forall x : T \mid \bullet \neg P)$

$$\Delta P \Rightarrow ((\forall x : T \mid P \bullet Q \wedge R) \equiv Q \wedge (\forall x : T \mid P \bullet R))$$

Additional Theorems

$$(\forall x : T \mid P \bullet \mathit{True})$$

$$(\forall x : T \mid P \bullet Q \equiv R) \equiv ((\forall x : T \mid P \bullet Q) \equiv (\forall x : T \mid P \bullet R))$$

Weakening, Strengthening and Monotonicity

$$(\forall x : T \mid P \vee Q \bullet R) \Rightarrow (\forall x : T \mid P \bullet R)$$

$$(\forall x : T \mid P \bullet Q \wedge R) \Rightarrow (\forall x : T \mid P \bullet Q)$$

$$(\forall x : T \mid P \bullet Q \Rightarrow R) \Rightarrow ((\forall x : T \mid P \bullet Q) \Rightarrow (\forall x : T \mid P \bullet R))$$

Instantiation For any c in T

$$(\forall x : T \mid \bullet P) \Rightarrow P[c/x]$$

Generalised DeMorgan

$$\neg(\exists x : T \mid P \bullet \neg Q) \equiv (\forall x : T \mid P \bullet Q)$$

$$\neg(\exists x : T \mid P \bullet Q) \equiv (\forall x : T \mid P \bullet \neg Q)$$

$$(\exists x : T \mid P \bullet \neg Q) \equiv \neg(\forall x : T \mid P \bullet Q)$$

Trading

$$(\exists x : T \mid P \bullet Q) \equiv (\exists x : T \mid \bullet \neg(P \Rightarrow \neg Q))$$

$$(\exists x : T \mid P \wedge R \bullet Q) \equiv (\exists x : T \mid P \bullet \neg(R \Rightarrow \neg Q)$$

Distribution Provided x is not free in Q,

$$(\exists x : T \mid P \bullet Q \wedge R) \equiv Q \wedge (\exists x : T \mid P \bullet R)$$

$$\Delta P \Rightarrow ((\exists x : T \mid P \bullet Q) \equiv Q \wedge (\exists x : T \mid \bullet P))$$

Provided x is not free in Q, and $(\exists x : T \mid \bullet P)$

$$(\exists x : T \mid P \bullet Q \vee R) \equiv Q \vee (\exists x : T \mid P \bullet R)$$

Additional Theorem

$$\neg(\exists x : T \mid P \bullet \mathit{False})$$

Weakening, Strengthening and Monotonicity

$$(\exists x : T \mid Q \bullet R) \Rightarrow (\exists x : T \mid P \vee Q \bullet R)$$

$$(\exists x : T \mid P \bullet Q) \Rightarrow (\exists x : T \mid P \bullet Q \vee R)$$

$$(\exists x : T \mid P \bullet Q \Rightarrow R) \Rightarrow ((\exists x : T \mid P \bullet Q) \Rightarrow (\exists x : T \mid P \bullet R))$$

Introduction and Exchange For any c in T

$$P[c/x] \Rightarrow (\exists x : T \mid \bullet P)$$

Provided x is not free in Q, and y is not free in P,

$$(\exists x : T \mid P \bullet (\forall y : T' \mid Q \bullet R)) \Rightarrow (\forall y : T' \mid Q \bullet (\exists x : T \mid P \bullet R))$$

Appendix B

The Printer Control Specification

Here we give an outline of how the final printer control specification looks.

Given Sets

$[\textsc{Person}]$, $[\textsc{Page}]$

Initial Definitions

$$\begin{array}{ll}
\textsc{JobId} & \hat{=}\ \mathbb{N} \\
\textsc{File} & \hat{=}\ \mathit{Seq}\textsc{Page} \\
\textsc{Priority} & \hat{=}\ \mathbb{N} \\
\textsc{Buffer} & \hat{=}\ \textsc{Page}
\end{array}$$

Definitions for State

$$\begin{array}{l}
\mathit{inf} : \textsc{Jobs} \mathrel{\hat{=}} [\mathrm{KnownJobs} \in \mathbb{P}\,\textsc{JobId} \\
\qquad\qquad\quad \mathrm{FileOf} \in \mathrm{KnownJobs} \nrightarrow_t \textsc{File}, \\
\qquad\qquad\quad \mathrm{OwnerOf} \in \mathrm{KnownJobs} \nrightarrow_t \textsc{Person}, \\
\qquad\qquad\quad \mathrm{PriorityOf} \in \mathrm{KnownJobs} \nrightarrow_t \textsc{Priority}] \\
\mathit{inf} : \textsc{Jobs} \vdash \mathrm{SizeOf} \mathrel{\hat{=}} \# \circ \mathrm{FileOf}
\end{array}$$

$$c : \textsc{CurrentJob} \mathrel{\hat{=}} [\mathrm{CurrentId} \in \textsc{JobId}, \mathrm{PagesPrinted} \in \mathbb{N}]$$

$$
\begin{aligned}
&\textsc{PrintQueue} \mathrel{\hat{=}} \text{ISeq}(\textsc{JobId}\setminus\{0\})\\
&q : \textsc{PrintQueue} \vdash \text{JobsWaiting} \mathrel{\hat{=}} \operatorname{ran} q,\\
&\qquad\qquad\qquad\quad \text{RemQueue} \mathrel{\hat{=}} (\textbf{fun } id \in \textsc{JobId} : \text{Remove}(q, id))
\end{aligned}
$$

$$
\begin{aligned}
&q : \textsc{PrintQueue}, c : \textsc{CurrentJob} \vdash \text{JobsInQueue} \mathrel{\hat{=}} \text{JobsWaiting} \cup \text{CurrentId},\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{EmptyQueue} \mathrel{\hat{=}} (\text{CurrentId} = 0)
\end{aligned}
$$

$$
\begin{aligned}
u : \textsc{Users} \mathrel{\hat{=}} [&\text{KnownUsers} \in \mathbb{P}\,\textsc{Person},\\
&\text{QuotaOf} \in \text{KnownUsers} \nrightarrow_t \mathbb{N},\\
&\text{PagesUsedBy} \in \text{KnownUsers} \nrightarrow_t \mathbb{N}] :\\
&(\forall p \in \textsc{Person}.\text{QuotaOf}\, p \geqslant \text{PagesUsedBy}\, p)
\end{aligned}
$$

$$
\begin{aligned}
\sigma : \Sigma \mathrel{\hat{=}} [&q \in \textsc{PrintQueue}, c \in \textsc{CurrentJob}, b \in \textsc{Buffer}, \mathit{inf} \in \textsc{Jobs}, u \in \textsc{Users}] :\\
&\quad(\text{PagesPrinted} \leqslant \text{SizeOf} \circ \text{CurrentId}\\
&\wedge\ \text{KnownJobs} = \text{JobsInQueue}\\
&\wedge\ \text{KnownUsers} \supseteq \text{OwnerOf} * \text{JobsInQueue}\\
&\wedge\ \text{CurrentId} \notin \text{JobsWaiting}\\
&\wedge\ (\text{CurrentId} = 0 \Rightarrow q = \langle\rangle)
\end{aligned}
$$

Operations over the State

Adding a Print Job

$$
\begin{aligned}
\sigma : \Sigma \vdash \text{AddOk} \mathrel{\hat{=}} (&\textbf{fun } p \in \textsc{Person}, f \in \textsc{File}, n \in \textsc{Priority} :\\
&p \in \text{KnownUsers} \rightarrow\\
&\qquad\qquad \textbf{let } \mathit{newId} = [\,]/(\mathbb{N}\setminus(\{0\} \cup \text{KnownJobs}))\\
&\qquad\qquad \&\quad \mathit{newq} = (\neg\text{EmptyQueue} \rightarrow q \frown \langle \mathit{newId} \rangle \overset{\leftarrow}{[\,]}\ q)\\
&\qquad\qquad \&\quad \mathit{newc} = (\neg\text{EmptyQueue} \rightarrow c \overset{\leftarrow}{[\,]}\ (\mathit{newId}, 0))\\
&\qquad\qquad \&\quad \mathit{newInf} = (\text{FileOf} \oplus \{\mathit{newId} \mapsto f\},\\
&\qquad\qquad\qquad\qquad\qquad \text{OwnerOf} \oplus \{\mathit{newId} \rightarrow p\},\\
&\qquad\qquad\qquad\qquad\qquad \text{PriorityOf} \oplus \{\mathit{newId} \mapsto n\})\\
&\qquad\qquad \textbf{in } (\mathit{newq}, \mathit{newc}, b, \mathit{newInf}, u))
\end{aligned}
$$

$$
\begin{aligned}
\sigma : \Sigma \vdash \text{AddError} \mathrel{\hat{=}} (&\textbf{fun } p \in \textsc{Person}, f \in \textsc{File}, n \in \textsc{Priority} :\\
&\qquad\qquad\qquad\qquad \textsc{Unknown\_User\_Error})
\end{aligned}
$$

$$
\sigma : \Sigma \vdash \text{Add} \mathrel{\hat{=}} \text{AddOk} \overset{\leftarrow}{\cup} \text{AddError}
$$

Allocating Quotas

```
$\sigma : \Sigma \vdash \text{Alloc} \mathrel{\hat{=}} (\textbf{fun}\ p \in \textsc{Person}, q \in \mathbb{N} :$
        $\textbf{let}\ newu = (\text{QuotaOf} \oplus \{p \mapsto q\},$
                        $\text{PagesUsedBy} \oplus \{p \mapsto 0\})$
        $\textbf{in}\ (q, c, b, inf, newu))$
```

Returning the Active Job

```
$\sigma : \Sigma \vdash \text{Active} \mathrel{\hat{=}} (\neg\text{EmptyQueue} \rightarrow \textbf{let}\ id = \text{CurrentId} \parallel n = \text{PagesPrinted}$
                                    $\&\ \ size = \text{SizeOf}\ id$
                                    $\textbf{in}\ (id, n, size - n)$
        $\overset{\leftarrow}{[\,]}\ \ \textsc{Queue\_Empty\_Error})$
```

Printing a Page

```
$q : \textsc{PrintQueue}, inf : \textsc{Jobs} \vdash \text{GetNextId} \mathrel{\hat{=}} (q \neq \langle\rangle \rightarrow$
        $\textbf{let}\ pr = (\textbf{fun}\ i \in \mathbb{N} : \text{PriorityOf}\ q[i])$
        $\textbf{in}\ \sqcap/(maxWRT\ pr\{0..\#q - 1\}))$
    $\overset{\leftarrow}{[\,]}\ \ 0)$
```

```
$\sigma : \Sigma \vdash \text{PrintOk} \mathrel{\hat{=}} (\neg\text{EmptyQueue} \rightarrow$
    $\textbf{let}\ id = \text{CurrentId} \parallel n = \text{PagesPrinted}$
    $\&\ \ p = \text{OwnerOf}\ id \parallel f = \text{FileOf}\ id$
    $\&\ \ quota = \text{QuotaOf}\ p \parallel pages = \text{PagesUsedBy}\ p\ \textbf{in}$
    $quota > pages \rightarrow$
        $\textbf{let}\ newb = f[n]$
        $\&\ \ newu = \text{ChangeUser}(quota, pages + 1)\ \textbf{in}$
        $(n < \text{SizeOf}\ id \rightarrow$
            $\textbf{let}\ newc = (id, n + 1)$
            $\textbf{in}\ (q, newc, inf, newu, newb)$
        $\overset{\leftarrow}{[\,]}$  $\textbf{let}\ newid = \text{GetNextId}$
            $\&\ \ newc = (newid, 0)$
            $\&\ \ newq = remove\ newid$
            $\&\ \ newInf = \text{RemInf}\ id$
            $\textbf{in}\ (newq, newc, newinf, newu, newb)))$
```

$\sigma : \Sigma \vdash \text{QuotaError} \mathrel{\hat{=}} (\neg\text{EmptyQueue} \rightarrow \textsc{Quota\_Error})$

$\sigma : \Sigma \vdash \text{QEmpty} \mathrel{\hat{=}} \textsc{Error\_Queue\_Empty}$

$\sigma : \Sigma \vdash \text{Printpage} \mathrel{\hat{=}} \text{Printok} \overset{\leftarrow}{[]} \text{QuotaError} \overset{\leftarrow}{[]} \text{QEmpty}$

Removing a Print Job

$$\begin{aligned}
\sigma : \Sigma \vdash \text{RemoveOk} \mathrel{\hat{=}} (\textbf{fun}\ id \in \textsc{JobId} :\ & \\
& id \in \text{JobsInQueue} \wedge id \neq \text{CurrentId} \rightarrow \\
& \quad \textbf{let}\ newq = \text{RemQueue}\ id \\
& \quad \&\ \ newinf = (\text{FileOf} \backslash id, \\
& \qquad\qquad\qquad \text{OwnerOf} \backslash id, \\
& \qquad\qquad\qquad \text{PriorityOf} \backslash id) \\
& \quad \textbf{in}\ (newq, c, b, newinf, u))
\end{aligned}$$

$$\begin{aligned}
\sigma : \Sigma \vdash \text{RemoveCurrent} \mathrel{\hat{=}} (\textbf{fun}\ id \in \textsc{JobId} :\ & \\
& id = \text{CurrentId} \rightarrow \textsc{Current\_Job\_Error})
\end{aligned}$$

$\sigma : \Sigma \vdash \text{RemoveFail} \mathrel{\hat{=}} (\textbf{fun}\ id \in \textsc{JobId} : \textsc{Job\_not\_in\_Queue\_Error})$

$\sigma : \Sigma \vdash \text{RemoveJob} \mathrel{\hat{=}} \text{RemoveOk} \overset{\leftarrow}{\cup} \text{RemoveCurrent} \overset{\leftarrow}{\cup} \text{RemoveFail}$

Bibliography

[1] A. Avron. Foundations and Proof Theory of 3-valued Logics. Technical Report ECS-LFCS-88-48, Department of Computer Science, University of Edinburgh, U.K., 1988.

[2] R.J. Back and M. Butler. Exploring Summation and Product Operators in the Refinement Calculus. In B. Möller, editor, *Mathematics of Program Construction : Proceedings of the Third International Conference, MPC'95, Kloster Irsee, Germany*, number 947 in LNCS, pages 128–158. Springer-Verlag, 1995.

[3] R.J.R. Back. Correctness Preserving Program Refinements: Proof Theory and Applications. Tract 131, Mathematisch Centrum, Amsterdam, 1980.

[4] R.J.R. Back. A Calculus of Refinements for Program Derivations. *Acta Informatica*, 25:593–624, 1988.

[5] R. Backhouse. An Exploration of the Bird-Meertens Formalism. In *Proceedings of the International Summer School on Constructive Algorithmics, Hollum-Ameland*, 1989.

[6] R. Backhouse, P.Chisholm, G. Malcolm, and E. Saaman. Do-it-yourself Type Theory. *Formal Aspects of Computing*, 1:19–84, 1989.

[7] H.P. Barendregt. Introduction to Generalized Type Systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[8] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall, 1990.

[9] H. Barringer, J.H. Chen, and C.B. Jones. A Logic Covering Undefinedness in Program Proofs. *Acta Informatica*, 21:251–269, 1984.

[10] J.A. Bergstra, J. Heering, and P. Klint. Module Algebra. *Journal of the Association for Computing Machinery*, 2:335–372, 1990.

[11] J.C. Bicarregui, J.S. Fitzgerald, P.A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, 1994.

[12] R. Bird and P. Wadler. *Introduction to Functional Programming.* Series in Computer Science. Prentice Hall International, 1988.

[13] R.S. Bird. An Introduction to the Theory of Lists. In M. Broy, editor, *Logic of Programming and Calculi of Descrete Design*, volume 36 of *NATO Series F*, pages 3–42. Springer-Verlag, 1986.

[14] R.S. Bird. A Calculus of Functions for Program Derivation. In D.A. Turner, editor, *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series, pages 287–308. Addison-Wesley, 1990.

[15] R.S. Bird. The Algebra of Programming Principles. In *Proceedings of the International Summer School on Deductive Program Design, Marktoberdorf*, 1994.

[16] R.S. Bird. Functional Algorithm Design. In B. Möller, editor, *Mathematics of Program Construction : Proceedings of the Third International Conference, MPC'95, Kloster Irsee, Germany*, number 947 in LNCS, pages 2–17. Springer-Verlag, 1995.

[17] J.E. Bresenham. An algorithm for computer control of a digital plotter. *IBM Systems Journal*, 1(4):25–30, 1965.

[18] A. Bunkenburg. *Expression Refinement: Imperative Programs.* Phd thesis, Department of Computing Science, University of Glasgow, 1997.

[19] A. Bunkenburg and S. Flynn. Expression Refinement: Deriving Bresenham's Algorithm. In K. Hammond, D.N. Turner, and P.M. Sansom, editors, *Functional Programming, Glasgow 1994*, Workshops in Computing, pages 1–17. BCS, Springer-Verlag, 1994.

[20] L. Cardelli. Typeful Programming. Technical Report 45, Digital Equipment Corporation Systems Research Center, May 1989.

[21] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):472–552, 1985.

[22] J.H. Cheng and C.B. Jones. On the Usability of Logics which handle Partial Functions. In Carroll Morgan and J.C.P. Woodcock, editors, *3rd Refinement Workshop*, pages 51–69. Springer-Verlag, 1990.

[23] J. de Bakker. *Mathematical Theory of Program Correctness.* Prentice Hall International, London, 1980.

[24] E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall, N.J., 1976.

[25] E.W. Dijkstra. A Computing Scientist's Approach to a Once-Deep Theorem of Sylvester's. In *Proceedings of the International Summer School on Deductive Program Design, Marktoberdorf*, 1994. EWD1016, 1988.

[26] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[27] A. Diller. *Z - An Introduction to Formal Methods*. Wiley, second edition, 1994.

[28] J.S. Fitzgerald and C.B. Jones. Modularizing the Formal Description of a database system. In *VDM'90 Conference, Kiel*, 1990.

[29] M.M. Fokkinga. Programming Language Concepts — The Lambda Calculus Approach. In P.R.J. Asveld and A. Nijholt, editors, *Essays on Concepts, Formalism, and Tools*, volume 42 of *CWI Tract*, pages 129–162. CWI, Amsterdam, 1987.

[30] M. Frappier, A. Mili, and J. Desharnais. Program Construction by Parts. In B. Möller, editor, *Mathematics of Program Construction : Proceedings of the Third International Conference, MPC'95, Kloster Irsee, Germany*, number 947 in LNCS, pages 258–281. Springer-Verlag, 1995.

[31] D. Gries. *The Science of Programming*. Springer, New, York, 1981.

[32] D. Gries and F.B. Schneider. *A Logical Approach to Discrete Math*. Texts and Monographs in Computer Science. Springer-Verlag, 1994.

[33] J. Grundy. A Window Inference Tool for Refinement. In C.B. Jones, B.T. Denvir, and R.C.F. Shaw, editors, *Proceedings of the 5th Refinement Workshop*, Workshops in Computing, pages 230–254. BCS FACS, Springer-Verlag, 1992.

[34] J. Grundy. *A Method of Program Refinement*. PhD thesis, University of Cambridge, Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, November 1993. Technical Report 318.

[35] C.A. Gunter and D.S. Scott. Semantic Domains. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 633–674. Elsevier Science Publishers B.V., Amsterdam, 1990.

[36] R. Harper, R.Milner, and M.Tofte. A Type Discipline for Program Modules. Technical Report ECS-LFCS-87-28, Department of Computer Science, University of Edinburgh, 1987.

[37] R. Heckmann. Power Domains Supporting Recursion and Failure. In J.-C. Raoult, editor, *Prodeedings of the 17th Colloquium on Trees in Algebra and Programming, Rennes, France*, number 581 in LNCS, pages 165–181. Springer-Verlag, 1992.

[38] E.C.R. Hehner. *The Logic of Programming.* Prentice Hall International, 1984.

[39] E.C.R. Hehner. *A Practical Theory of Programming.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[40] C.B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, U.K., second edition, 1990.

[41] C.B. Jones. Partial functions and logics: A warning. *Information Processing Letters*, 54(2):65–67, 1995.

[42] C.B. Jones. TANSTAAFL (with partial functions). Work in Progress, 1996.

[43] C.B. Jones, K.D. Jones, P.A. Lindsay, and R. Moore. *mural: A Formal Development Support System.* Springer-Verlag, 1991.

[44] J.Woodcock and J.Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall, U.K., 1996.

[45] A. Kaldewaij. *Programming – The Derivation of Algorithms.* Prentice Hall, 1990.

[46] D.J. King and P. Wadler. Combining Monads. In J. Launchbury and P.M. Sansom, editors, *Functional Programming, Glasgow 1992*, Workshops in Computing. Springer-Verlag, 1993.

[47] J.-L. Lassez, V.L. Nguyen, and E.A. Sonenberg. Fixed Point Theorems and Semantics: A Folk Tale. *Information Processing Letters*, 14(3):112–116, 1982.

[48] J. Launchbury and S.L. Peyton Jones. Lazy Functional State Threads. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation, Orlando*, June 1994.

[49] D. MacQueen. Using Dependent Types to Express Modular Structure. Technical report, Department of Computer Science, University of Edinburgh, 1986.

[50] Z. Manna and A. Shamir. The Theoretical Aspects of the Optimal Fixedpoint. *SIAM Journal of Computing*, 5(3):414–426, 1976.

[51] B. Meyer. Eiffel: Programming for Reusability and Extendibility. *SIGPLAN Notices*, 22(2):85–94, 1987.

[52] E. Moggi. An Abstract View of Programming Languages. Technical Report ECS-LFCS-90-113, Department of Computer Science, University of Edinburgh, 1989. Lecture Notes for course CS 359, Stanford University.

[53] E. Moggi. Computational Lambda-Calculus and Monads. In *Symposium on Logic in Computer Science*, 1989.

[54] E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93(1), 1991.

[55] C. Morgan. The Specification Statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[56] C. Morgan. *Programming from Specifications.* Prentice Hall, U.K., 1990.

[57] C. Morgan. The Refinement Calculus. In *Proceedings of the International Summer School on Program Design Calculi, Marktoberdorf*, 1992.

[58] C. Morgan and P.H.B. Gardiner. Data Refinement by Calculation. *Acta Informatica*, 26:481–503, 1990.

[59] J.M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9:287–306, 1987.

[60] J.M Morris. Laws of Data Refinement. *Acta Informatica*, 26:287–308, 1989.

[61] J.M. Morris. Piecewise Data Refinement. In Edsger W. Dijkstra, editor, *Formal Development of Programs and Proofs*, University of Texas at Austin Year of Programming Series, chapter 10, pages 117–137. Addison-Wesley, 1989.

[62] J.M. Morris. Programs from Specifications. In Edsger W. Dijkstra, editor, *Formal Development of Programs and Proofs*, University of Texas at Austin Year of Programming Series, chapter 9, pages 81–115. Addison-Wesley, 1989.

[63] J.M. Morris. Nondeterministic Expressions and Functional Imperative Programming. To appear, 1994.

[64] J.M. Morris. Reasoning Equationally in the Presence of Undefinedness. Submitted for Publication, 1996.

[65] J.M. Morris. Undefinedness and Nondeterminacy in Program Proofs. To appear, 1996.

[66] P.D. Mosses. Denotational Semantics. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 575–631. Elsevier Science Publishers B.V., Amsterdam, 1990.

[67] G. Nelson. A generalization of Dijkstra's Calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.

[68] T.S. Norvell and E.C.R. Hehner. Logical Specifications for Functional Programs. In R. Bird, C. Morgan, and J. Woodcock, editors, *Mathematics of Program Construction 1992*, number 669 in LNCS, pages 269–290. Springer-Verlag, 1993.

[69] J.T. O'Donnell and G. Rünger. A Case Study in Parallel Program Derivation: The Heat Equation Algorithm. In K. Hammond, D.N. Turner, and P.M. Sansom, editors, *Functional Programming, Glasgow 1994*, Workshops in Computing, pages 167 – 183. BCS, Springer-Verlag, 1994.

[70] H.A. Partsch. *Specification and Transformation of Programs. A formal approach to software development.* Springer-Verlag, 1990.

[71] L.C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[72] S.L. Peyton Jones and P. Wadler. Imperative Functional Programming. In *20th ACM Symposium on Principles of Programming Languages*, pages 71–84. ACM Press, 1993.

[73] G. Plotkin. A Powerdomain Construction. *SIAM Journal of Computing*, 5(3):452–487, 1976.

[74] G. Plotkin. Domains. Lecture Notes, Department of Computer Science, University of Edinburgh, 1983.

[75] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z.* Prentice Hall, U.K., 1991.

[76] J.C. Reynolds. Three Approaches to Type Structure. In E.A. Ehring, editor, *Mathematical Foundations of Software Development*, pages 97–138. Springer, Berlin, 1985.

[77] R. Salmon and M. Slater. *Computer Graphics, Systems and Concepts.* Addison-Wesley, 1987.

[78] D. Sannella. Formal Specification of ML Programs. Technical Report ECS-LFCS-86-15, Department of Computer Science, University of Edinburgh, 1986.

[79] D. Sannella. Formal program development in Extended ML for the working programmer. In Carroll Morgan and J.C.P. Woodcock, editors, *Proceedings of the 3rd Refinement Workshop*, pages 99–130. Springer-Verlag, 1990.

[80] D. Sannella, S. Sokolowski, and A. Tarlecki. Towards formal development of programs from algebraic specifications : parameterisation revisited. Technical report, Department of Computer Science, University of Edinburgh, 1990.

[81] D. Sannella and A. Tarlecki. Toward Formal Development of Programs from Algebraic Specifications: Implementations Revisited. *Acta Informatica*, 25:233–281, 1988.

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

[83] M. Smyth. Power domains. *Journal of Computer and System Sciences*, 16(1):23–36, 1978.

[84] H. Sondergaard and P. Sestoft. Non-determinism in Functional Languages. *The Computer Journal*, 35:514–523, 1992.

[85] R.F. Sproull. Using Program Transformations to Derive Line-Drawing Algorithms. *ACM Transations on Graphics*, 1(4):259–273, 1982.

[86] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* Series in Computer Science. The MIT Press, 1977.

[87] S. Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[88] P. Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[89] P. Wadler. Monads for Functional Programming. In *Proceedings of the International Summer School on Program Design Calculi, Marktoberdorf*, volume 118 of *NATO ASI Series F : Computer and systems sciences.* Springer-Verlag, 1992.

[90] N. Ward. *A Refinement Calculus for Nondeterministic Expressions.* Phd thesis, The University of Queensland, February 1994.