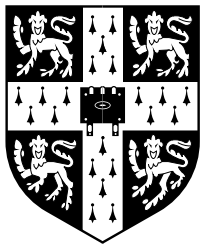
A Mechanised Theory of Refinement

Mark Staples

Gonville and Caius College

A dissertation submitted for the degree of
Doctor of Philosophy at the University of Cambridge



Second edition 1999.

Typeset by the author using the LaTeX document preparation system.
Printed in England at the University of Cambridge, Computer Laboratory.

Foreword

This is the second edition of my PhD dissertation which contains minor corrections and improvements to the version submitted in November 1998. Most of the changes reflect corrections required or suggested after the careful reading of my dissertation by my examiners, Joakim von Wright and Larry Paulson. I have also incorporated some minor changes reflecting suggestions about the presentation of this material by the anonymous reviewers of a conference paper:

> *Representing WP Semantics in Isabelle/ZF* Staples, M. (1999) In Dowek, G., Paulin, C., and Bertot, Y. (Eds.) TPHOLs: The 12th International Conference on Theorem Proving in Higher-Order Logics. LNCS, Springer-Verlag

Mark Staples, 13 July 1999

Preface

This dissertation presents new ideas in the design of program refinement tools. Program refinement is a formal method for stepwise program development. Tool support for program refinement can increase our confidence in the soundness of our refinement theory and in the ultimate correctness of our program derivations. A refinement tool must be expressive enough to represent all of the commands in our refinement language, must give ready access to the standard results in classical mathematics, should support the process of our refinement methodology, and should facilitate the valid realisation of completely developed programs. This dissertation presents new ideas and techniques which endeavour to address these requirements.

Several refinement tools use a style of reasoning called window inference. Window inference supports the transformation of terms under preorder relations, such as procedural refinement. We describe a generalised version of window inference which works with arbitrary composable relations, and thus allows the transformation of programs using data-refinement. Our flexible window inference can also support more interaction schemes in the development of a proof by allowing multiple windows to open simultaneously on the top-level term.

We then describe a new representation for the semantics of a program refinement language. Our theory is mechanised in Isabelle/ZF, a theorem prover for untyped set theory. This very expressive logic allows us to represent a wider variety of commands than earlier similar refinement tools. We treat program states as dependently typed functions from variable names to values. We show how to encode contextual information in the syntax of our language so as to support our refinement methodology. The uniform and explicit nature of our representation increases the perspicuity of our definitions and theorems. We use several devices to ameliorate the additional notational burden which would otherwise accompany our expressive framework. Finally, we demonstrate the utility of our mechanised theory in a case study: the implementation of a propositional tautology checker.

Acknowledgements

I am thankful for the the financial support provided by the Packer scholarship scheme, the Cambridge Commonwealth Trust, the Overseas Research Studentship scheme, the Cambridge Philosophical Society, and my parents, without which my study in Cambridge would not have been possible. I have also received valuable financial support for travel from the Cambridge Philosophical Society, Data Connection, Glasgow University, Gonville and Caius College, and the NATO Science Committee.

I would like to thank my supervisor Mike Gordon, for providing guidance, support, and an environment of freedom for my research. I would also like to thank Larry Paulson, who has gone beyond the call of duty in both supporting Isabelle and welcoming me in the Isabelle group.

The past and present members of the Automated Reasoning Group have always been supportive and stimulating. It has been a pleasure to share my office with Brian, Paul, Chris, Myra, and the other two Australasian musketeers: Michael and Don. Florian, Konrad, Jim, Joakim, John, Sara, Sten, Tom and Thomas have also provided help and encouragement during my time at Cambridge.

The following people have kindly proof-read parts or all of earlier versions of this thesis: Thomas Forster, Mike Gordon, John Harrison, Michael Norrish, Helen Staples, John Staples and Jill Taylor. Their suggestions have resulted in a much improved dissertation. Any remaining errors are of course my own.

I would like to thank Ian Hayes, Helen Purchase, and others at the University of Queensland's Department of Computer Science, who set me on the road to research.

Finally, thanks to Jill for being with me in good times and in bad.

Contents

Chapter 1

Introduction

This dissertation presents new techniques for use in the design of program refinement tools. Program refinement is a formal method for stepwise program development. Formal methods is a study of design in computer science, using mathematically described theories to justify the correctness of implementations relative to their specifications. The design-time analysis of predicted behaviour is central to the design of safety-critical computer systems [79]. Calculi for program refinement were developed by Back [7], Morris [75, 77] and Morgan [71, 70] as extensions of Dijkstra's guarded command language [31]. A refinement language is a wide-spectrum language which encompasses both abstract specification statements and concrete imperative programming statements. Procedural refinement is a way of incrementally transforming specification statements into corresponding correct implementations. Data-refinement is a special kind of refinement step which allows the representation of local program variables to be changed so that more efficient implementations can be given [48, 78, 14].

Tool support for program refinement can increase our confidence in the soundness of our refinement theory and ultimately in the correctness of our program derivations. A refinement tool must be expressive enough to represent all of the statements in our refinement language, must give ready access to the standard results in classical mathematics, should support the process of our refinement methodology, and should facilitate the valid realisation of completely developed programs [26, 25]. This dissertation presents new ideas and techniques which endeavour to address these requirements. Our tool is based on a theorem prover supporting classical mathematics, and is expressive enough to deal with the difficult representational demands of the refinement calculus. We support our refinement methodology by choosing a representation of our language which allows the contextual refinement of recursion blocks and procedures, and by using an inference technique that

allows the data-refinement of programs. The uniform and explicit nature of our representation of states should facilitate the translation of our language to other common styles of semantic representation.

Our work is performed in the theorem prover Isabelle/ZF, which mechanises an expressive untyped set theory underlying a large collection of standard results in classical mathematics [85]. We use this expressive logic to represent our refinement language. Representing a semantics for the refinement calculus is more challenging than representing the semantics of an ordinary programming language. A program refinement language contains abstract specification statements, and may introduce local blocks whose variables have arbitrarily complex abstract types. We use dependently typed functions in Isabelle/ZF to represent program states as an underspecified map from variable names to their values. This treatment of states allows us to give a uniform treatment of variable assignment statements, framed specification statements, local blocks, and procedure parameterisation. Our uniform logic for representing the refinement theory and its assertion language allows us to embed meta-level contextual information within statements in our language. This helps us to support a stepwise refinement methodology for the development of recursion blocks and procedures.

Using an untyped set-theory in preference to an implicitly typed framework provides an extra overhead in formalisation. However, we use several devices to ameliorate the additional notational burden this expressive framework would otherwise entail. The definitions of our statements extract typing information from their subcomponents, and we use facilities in Isabelle's higher-order meta-logic to hide most of the remaining explicit typing in our semantic representation.

Several existing refinement tools [42, 102, 29] use a style of reasoning called window inference [90, 43]. Window inference supports the transformation of terms under preorder relations, such as procedural refinement. We describe a generalised version of window inference which works with arbitrary composable relations, and thus allows the transformation of programs using data-refinement. Our flexible window inference can also support more interaction schemes in the development of a proof by allowing multiple windows to open simultaneously on the top-level term.

We demonstrate the utility of our mechanised theory in a case study: the implementation of a propositional tautology checker. This involves the refinement of an initial specification to a program using decision trees, and its subsequent data-refinement to more restricted classes of trees, and then to a final program using reduced ordered negation trees. The program we develop uses recursive procedures with recursive calls from within local blocks with various types.

The remainder of this introduction will examine the nature of program refinement and formal methods in computer science. We will argue that formal methods is neither a scientific nor an engineering discipline, but is rather a study of design, sharing concerns with both science and engineering. We will examine the role of representation and mechanised tool support for formal methods in general, and program refinement in particular. Finally, we will describe the Isabelle theorem prover, and present an outline of this dissertation.

1.1 Science, Engineering, and the Study of Design

Computer science is concerned with the nature of computation, ultimately realised as a physical process. Computer science includes scientific and engineering work, but also includes work, such as that in formal methods, which is a different kind of investigative activity. Formal methods is concerned with the design and analysis of computer systems, but abstracts away from immediate concerns about physical realisation.

1.1.1 Science and Engineering

Science can be characterised as the investigation of the world through the use of the scientific method. Scientists create general theories about the nature of the world, from which they make specific testable predictions. Theories are evaluated by investigating the correspondence between their predictions and the observations of specific physical phenomena. (See Figure 1.1, diagram A.) This is a fairly abstract view of science which is not inconsistent with the philosophies of Popper [87] or Kuhn [58]. We can similarly characterise engineering as the construction of artifacts using an engineering method. Engineers take quantifiable requirements specifications for the performance of artifacts. These specifications inform the design of an artifact, which after construction is evaluated by testing its performance against its requirements specifications. (See Figure 1.1 diagram B.)

These scientific and engineering activities can be both placed within a space depicted in Figure 1.2. This graph shows three dimensions:

Formal vs. Real
: Formal things are syntactic or mathematical, whereas Real things are causal entities or events.

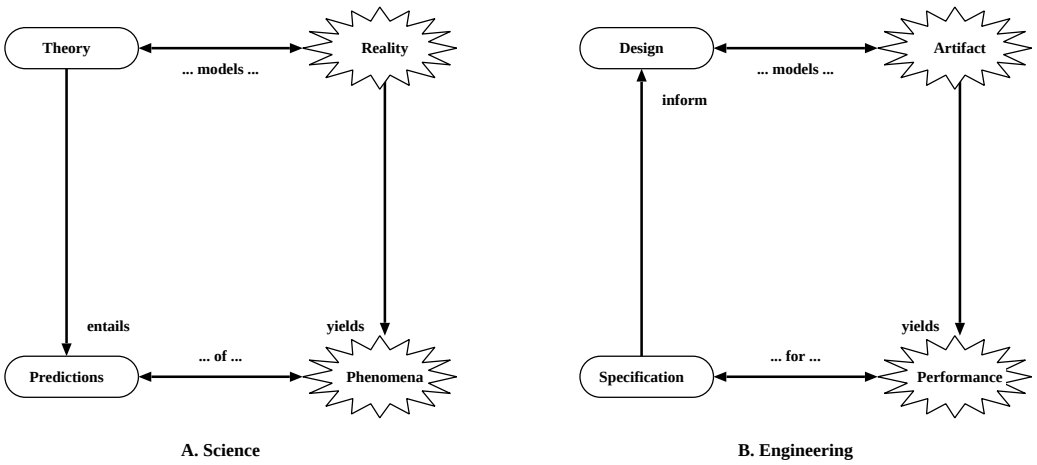


Figure 1.1: Science and Engineering

Formal things include a scientist's *theories* and consequent *predictions*, and an engineer's *specifications*, leading to *designs* for artifacts. Real things include a scientist's investigated *reality* and the observations of specific *phenomena*, and also an engineer's constructed *artifacts* and their *performance* in test and use.

Analytic vs. Synthetic

Synthesis combines pieces into a whole, whereas analysis takes apart wholes into pieces. We call synthesised things 'synthetic', and analysed things 'analytic'. (Our use of these terms is not the same as Kant's [53].) Formal synthetic things are related to formal analytic things by mathematical consequence. Real synthetic things are related to real analytic things by causation.

Analytic things include a scientist's *predictions* consequent upon their theories, and observations of specific *phenomena* in the real world, and also include an engineer's listed *specifications* for evaluating the *performance* of artifacts. Synthetic things include a scientist's encompassing *theories* of a universal *reality*, and an engineer's complete *designs* for entire *artifacts*.

Description vs. Prescription

On the descriptive plane, if there is a discrepancy between the formal and the real, the formal is wrong, whereas on the prescriptive plane, the converse holds. Correspondences between formal and real planes must be preserved at the synthetic and analytic levels.

Scientists investigate the correspondence between their *theories* and

reality, which they probe through an investigation of the correspondence between the *predictions* of the theory and *observations* of events related to those predictions. Unexpected discrepancies between predictions and observations may reduce support for the validity of the scientific theory.

Engineers produce *designs* prescribing the construction of *artifacts*, which they then test by investigating the correspondence between the *performance* of the artifact and its given *specifications*. If there is an unexpected discrepancy between performance of the artifact and its specifications, the artifact has failed its test, which may indicate that the design is bad in some way.

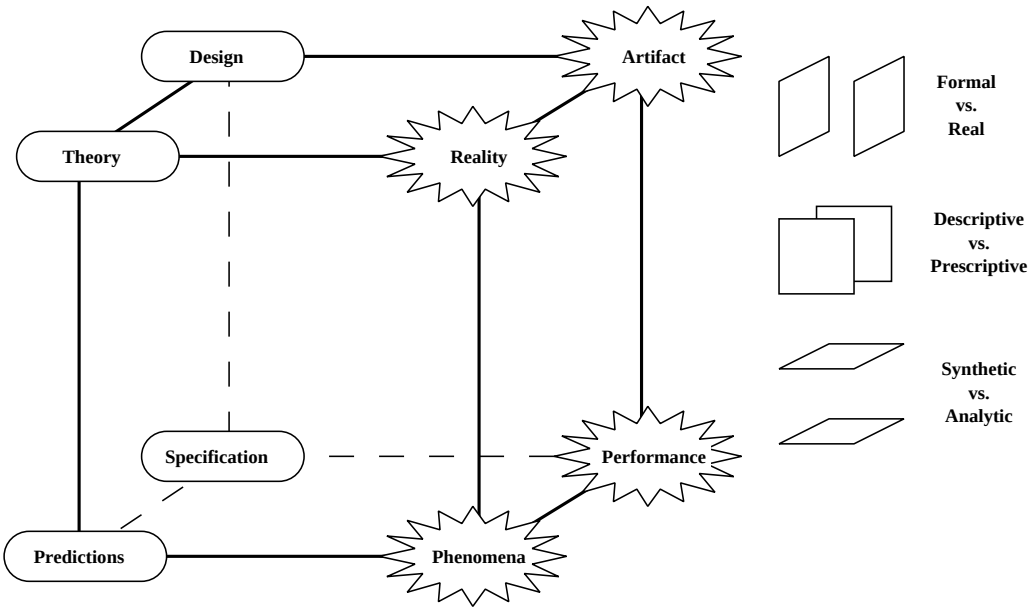


Figure 1.2: Science and Engineering Cube

These dimensions are not necessarily orthogonal—a separation between nodes does not necessarily imply a commitment to their ontological unrelatedness. For example, physical phenomena, artifacts and their performance are all a part of reality. Nonetheless, the distinctions we propose are useful for expository purposes. In this dissertation we consider the refinement calculus, which uses a wide-spectrum language operating over a continuum between abstract specifications and fully developed programs. Though, from a mathematical perspective, there is no clear difference between specifications and concrete designs, they do have very different roles in development.

The distinction between science and engineering is also not as straightforward as it might seem—engineers use scientific theories as a language for representing designs, and also perform experiments during the design and construction of artifacts. Dually, scientists construct tools and apparatus for testing their theories. Nonetheless, there is an over-arching difference of intention between science and engineering. Brooks, in his ACM Allen Newell award lecture [22, p62], says of this teleological contrast:

> That distinction [between scientific and engineering disciplines] lies not so much in the *activities* of the practitioners as in their *purposes*. [...] ...the scientist *builds in order to study*; the engineer *studies in order to build*.

1.1.2 Applied Computer Science

The popular view of computer scientists is that they are primarily interested in the design and construction of artifacts (programs or chips). This view is a largely accurate view of applied computer science. Juris Hartmanis, in his Turing award lecture [45, p40], says:

> Systems building, hardware and software, is the defining characteristic of applied and/or experimental work in computer science...

This suggests that applied computer science is an engineering discipline. However, there are two ways in which we might consider parts of applied computer science to be a scientific, descriptive activity. First, we must bear in mind the teleological subtlety discussed above: computers systems may be constructed to serve as tools in some larger descriptive investigation of computer science.

Secondly, just because something has been constructed, it isn't necessarily the case that it has been designed for the purpose for which it is used. Many computer systems have evolved over time through informal social processes. These systems nonetheless need to be understood, rather than simply dismissed as incorrect relative to some idealised specification. This is a position outlined by Milner [65, p247]:

> [thirty years ago] the main concern was to *prescribe* the behaviour of single computers or their single programs. Now that computers form just parts of larger systems, there is increasing concern to *describe* the flow of information and the interaction among the components of those larger systems.

Regarding parts of applied computer science as a science in this sense is perhaps a small departure from the normal understanding of science as an investigation of the natural world. However, applied computer science in this sense is still an investigation of the *physical* world. Milner [65, p250] says:

> To the extent that the phenomena of computing and communication are man-made, we have a substantial new 'science of the artificial' as Herbert Simon (1980) has recognised.

So although much of applied computer science is concerned with engineering practice, parts of it can be seen to be scientific: when computers are created as tools for the purpose of investigating some phenomena, or when extant computer systems are descriptively investigated, we can say that applied computer science is a scientific activity. What then about theoretical computer science?

1.1.3 Theoretical Computer Science

Theoretical computer scientists create and investigate mathematical models of idealised computation, and algorithms based on these models. Though this is a mostly mathematical investigation, it is not a completely contingent one. The standard notion of computation is to some extent limited by the Church-Turing thesis [98], which holds that all sufficiently powerful models of computation are essentially equally powerful. The limits of information theory [92] also pose constraints on investigations within theoretical computer science. Hartmanis [45, p41] sums up this view as follows:

> ...engineering in our field has different characteristics than the more classical practice of engineering. Many of the engineering problems in computer science are not constrained by physical laws.

The standard notion of computation brings with it constraints on complexity and computability of problems and algorithms. Theoretical computer science is an applied mathematics which investigates these issues.

1.1.4 Formal Methods and the Study of Design

Formal methods is usually seen as a branch of theoretical computer science, concerned with investigating formal properties of computer systems, languages and programs. We have said that computer science contains engineering and scientific activities. However, parts of computer science, especially much work in formal methods, can more usefully be considered a

'study of design'[1] dissimilar to normal science or engineering. The view of Hartmanis [45, p41] is that:

> ...computer science is concentrating more on the *how* than the *what*, which is more the focal point of physical sciences.

Similarly, Milner's position [64, p5] is that:

> ...for a physical scientist an experiment will reinforce (or undermine) his conceptual grasp of *what is true*; for a computer scientist, the experiment will reinforce (or undermine) his conceptual grasp of *how to design*.

Formal methods is based on mathematical modelling, but unlike mathematics, it is ultimately concerned with issues related to system design. Formal methods researchers investigate ways in which results based on mathematical models of computation can be used to improve the design process for computer systems. This is like working on the 'formal' face of the science and engineering cube, as depicted in Figure 1.3. There, we indicate various

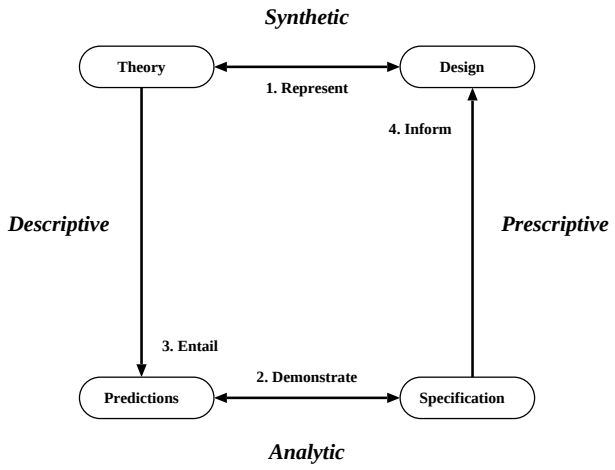


Figure 1.3: Formal Methods

kinds of formal methods research, including:

1. ways in which theoretical models can represent programs and designs for computer systems;

[1]Simon calls something like this a 'science' of design [94].

2. how predictions derived from these models can be used to demonstrate the correctness of designs (program verification is an example of this);

3. discovering classes of theoretical results which are useful when demonstrating the correctness of a system; and

4. how formal specifications can be used to inform the design and implementation of correct computer systems (program refinement includes examples of this kind of research).

Results in science and engineering are evaluated by investigating correspondences between the formal and real planes. How do we evaluate results in formal methods? Hartmanis [45, p40] considers that:

> In computer science, results of theory are judged by the insights they reveal about the mathematical nature of various models of computing and/or by their utility to the practice of computing and their ease of applicability.

Researchers in formal methods evaluate the fitness of models to support a given design methodology, and also the utility of design processes to support design under a given theoretical model.

Fetzer [32] criticises program verification as being impossible in principle. His position is that as all work in program verification is done at a formal level, no correspondence with computation in the real world is ever established, and hence program verification will never establish that real-world programs will function correctly. In one sense Fetzer's criticisms are valid: program verification does just work with formal entities, and not with real-world programs or computers. However, this is not a problem if other parts of computer science deal with the correspondence between the formal and real in a suitable way. The 'very idea of program verification' is based on an understanding that an engineer is taking care (to an appropriate level of detail) of the prescriptive correspondence between the program verifier's model of computation and some physical computing device.

1.2 Representation and Mechanisation

Formal methods uses logics and mathematics to represent systems or languages.[2] Formal methods researchers are mostly interested in representing

[2]Languages, systems, machines and logics are all potentially representing and representable, so for our purposes are essentially equivalent. We will usually say 'logic' for the representing thing, and 'language' for the represented thing. Some may prefer 'theory' instead of 'logic'.

the semantics and abstract syntax of a language, rather than being too concerned with issues of concrete syntax—parsing is a well understood technology, and does not concern us here. Different formal methods provide different logics and techniques for representing and reasoning about formal languages. The style of representation impacts on both its expressive power and efficacy for representing systems. There is often a trade-off between these two. An overview of styles of representation is presented below in Section 1.2.1.

Giving a formal specification of a system is requires more precision than giving an informal description, but this very difficulty encourages a clearer analysis of the system. Mechanised logics cannot rely upon any 'hand-waving' over matters of syntax or semantics. Formal methods tools are constructed by using either a theorem prover or an ad-hoc tool as a logic to represent languages or systems. The use of tools should facilitate the practice of formal methods for the same reasons that computers are useful; computers provide high accuracy in complex systematisable tasks. The correctness proofs typically found in formal methods are not necessarily mathematically deep, but they are prone to simple errors whose effects can propagate throughout the development. Formal methods tools should manage this detail, allowing larger systems to be addressed, and automatisation should help to bridge the gap between the prevailing and required level of mathematical sophistication for formal methods. Section 1.2.2 gives an overview of the mechanisation of formal methods.

1.2.1 Formal Embeddings of Languages and Systems

Languages are represented in a formal methods tool by 'embedding' the language in the logic underlying the tool. Two broad styles of embedding can be identified: deep and shallow [20]. They are depicted in Figure 1.4.

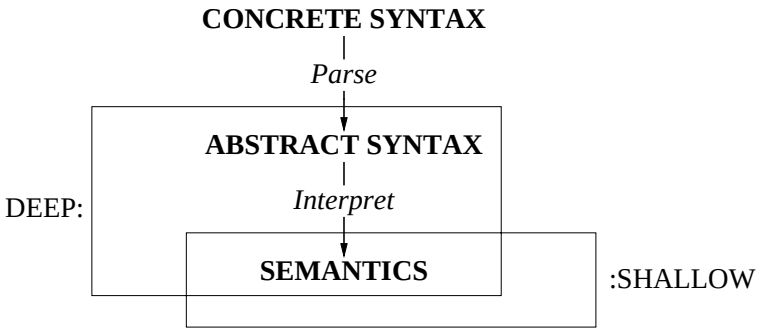


Figure 1.4: Deep and Shallow Embeddings

In a deep embedding, both the abstract syntax and the semantics of the language are represented in the logic, whereas in a shallow embedding, only the semantics of the language is represented. In a deep embedding, the relationship between syntax and semantics (the 'meaning' or 'interpretation' relationship) is explicitly defined and reasoned about, whereas in a shallow embedding the interpretation of abstract syntax occurs extra-logically. A tiny example demonstrating both styles of embedding is shown in Figure 1.5.

In the choice between deep and shallow embedding, there is a trade-off between expressiveness and ease of use [5]. Generally, it is easier to work with the semantics of shallowly embedded languages, as one can work on the semantics directly, rather than being directed to it through the syntax. It is also typically easier to incrementally extend shallowly embedded languages. For example, consider adding a multiplication expression to the example in Figure 1.5. In the deep embedding we would need to redefine expressions and expression interpretation, and re-prove all old results in the new language. In the shallow embedding, we need merely define a new semantic abbreviation, and prove old results just for the new expression.

However, a deep embedding may allow more meta-results to be stated and proved about a language. Because the syntax of the language is represented within the logic, we can state theorems involving syntactic conditions, and we can prove theorems relying upon properties arising out of an interpretation function limited by the fixed syntax of the language. The situation can become slightly blurred if it is possible to semantically characterise a class of interesting syntactic structures. Then, a shallow embedding can rely on this semantic restriction in place of a syntactic one. An example of this is found at the end of Figure 1.5, where the semantic side-condition 'a is monotonically increasing' stands in place of the syntactic side-condition 'a is a MINUS-free expression'.

If a deep embedding is used to represent a language, the question remains as to how much of the syntax is to be represented in the logic. Often, a language consists of separable sub-languages, each of which may be able to be represented in either a deep or shallow style. If the 'depth' of an embedding refers to whether the syntax is represented in the logic, this distinction might be well named the 'breadth' of an embedding. For example, in the deep embedding of an imperative programming language, one may represent only the syntax of the statement language, but leave the expression language as a shallow embedding [36]. Alternatively, one might also define the syntax and interpretation of the expression language [84]. One may even go as far as to embed an assertion language for a reasoning system about the language [49]. The more of a language that is deeply embedded, the more meta-results are possible, but the more unwieldy it is likely to be.

Example Representation Problem

Say we must represent and prove facts about simple arithmetic expressions involving integer-valued variables. Two approaches are outlined below.

Deep Embedding

We define the expression language *exp*:

$$exp \quad ::= \quad \texttt{VAR}\ v \mid exp\ \texttt{PLUS}\ exp \mid exp\ \texttt{MINUS}\ exp$$

and the meaning $[\![\_]\!]\_ : (exp \times (\mathbb{V} \to \mathbb{Z})) \to \mathbb{Z}$ of expressions under variable valuation s:

$$\begin{array}{lcl} [\![\texttt{VAR}\ v]\!]_s & \widehat{=} & s(v) \\ [\![a\ \texttt{PLUS}\ b]\!]_s & \widehat{=} & [\![a]\!]_s + [\![b]\!]_s \\ [\![a\ \texttt{MINUS}\ b]\!]_s & \widehat{=} & [\![a]\!]_s - [\![b]\!]_s \end{array}$$

Shallow Embedding

Here, we *identify* expressions with their semantics.

$$\begin{array}{lcl} \mathsf{VAR}\ v & \widehat{=} & \lambda\, s.\ s(v) \\ a\ \mathsf{PLUS}\ b & \widehat{=} & \lambda\, s.\ a(s) + b(s) \\ a\ \mathsf{MINUS}\ b & \widehat{=} & \lambda\, s.\ a(s) - b(s) \end{array}$$

Comparison

The embeddings are similar, and it is possible to prove, e.g. that:

$$[\![a\ \texttt{PLUS}\ b]\!]_s = (a\ \mathsf{PLUS}\ b)(s)$$

For the deep embedding we can state and prove theorems such as:

all MINUS-free expressions are monotonically increasing

However, under the shallow embedding, there is no semantic analogue to the syntactic condition 'MINUS-free expression', and so the theorem is impossible to state, let alone prove. The situation is not always hopeless; for example, here it is possible to state and prove theorems such as:

if a and b are monotonically increasing, then so is $a\ \mathsf{PLUS}\ b$

Theorems like this cannot be stated for the language as a whole, but can be repeated for each applicable syntactic category.

Figure 1.5: Example: Styles of Embedding for Arithmetic Expressions

Finally, most studies of semantics in computer science distinguish between styles of semantics, such as operational semantics and denotational semantics. This 'style of semantics' is mostly independent of our 'style of embedding'; there are examples in the literature of deep and shallow embeddings for most major styles of semantics. See Table 1.1 for citations to representative work. There is a gap in this table: there have been no investigations of shallow embeddings for operational semantics. It is in principle possible to have a shallow embedding for an operational semantics, as a syntax is not strictly necessary to close a set of relations. Alternatively, a language might be defined by abbreviations of syntax in a core abstract machine which itself had a normal operational definition.

	Deep	Shallow
Operational	[36, 84]	
Denotational	[36]	[3]
Predicate Transformer	[97]	[102]

Table 1.1: Examples of research using deep and shallow embeddings for various styles of semantics.

1.2.2 Mechanisation of Formal Methods

There is a range of activities in formal methods which can be supported by tools, and there is a spectrum of solutions to support the logical activities in a formal methods tool.

The principal activities of formal methods are logical ones such as system and property description, proof-obligation generation, and property proof. Tools may also provide support for meta-logical activities such as adding new methodological components, logical axioms and rules, or proof tactics. As with any computer-aided activity, tools can support administrative activities such as managing cooperative work, browsing or replaying completed developments, managing component libraries and version control.

There is a spectrum of approaches which can be used for formal methods tools. The principal activities of formal methods tools are logical ones, and so may be best supported by component sub-systems which are primarily devoted to supporting sound mathematics. Ultimately, soundness is critical for formal methods tools: their *raison d'être* is to help their users avoid reasoning errors. Theorem provers are such systems, and can be used in formal methods in one of two ways: as external or same-system components. External theorem provers are used as components outside the main part of the formal

methods tool. They are typically used to perform particular logical tasks for the tool such as proving side-conditions. Same-system theorem provers are integral to the tool, and share a common framework for representing the language. They have the advantage of not being subject to errors arising out of interaction between mutually inconsistent logical systems.

Of course, a formal methods tool might not make use of a theorem prover at all, but rather provide a hand-crafted ad-hoc system for dealing with the logical parts of the tool. Other lightweight formal methods tools might provide no support for proof, but merely aim to help users write down the description of systems and their properties.

1.3 Mechanising Program Refinement

Program refinement is a branch of formal methods which uses a 'wide-spectrum language' to represent both executable programming statements and non-executable specification constructs. This wide-spectrum language allows a seamless gradation between abstract system specifications and concrete implementations. A brief overview of program refinement and existing program refinement tools appears below.

1.3.1 Program Refinement

Program refinement was invented by Back [7] and independently discovered and popularised by Morris [75] and Morgan [71, 72]. It is based on a generalisation of Dijkstra's guarded command language [31]. The guarded command language is a simple imperative programming language with constructs such as sequential composition, assignment, alternation, and while loops. Dijkstra defined four 'healthiness conditions' which his programming language satisfied. These conditions made the language potentially implementable and also facilitated reasoning about the language. The development of the refinement calculus came about through an investigation of the guarded command language where some of these restrictions were relaxed or removed. Hence the refinement calculus is a wide-spectrum language which can represent both imperative programming statements and non-executable specification constructs. A refinement relation on statements in the language can be defined which preserves functional correctness and decreases nondeterminism. System specifications abstract implementation details and so are typically highly nondeterministic. The refinement relation provides a gradation between abstract system specifications and their concrete implementations.

The refinement calculus is best known as a formal stepwise program development method, as promulgated by Morgan[72]. Under this approach, a developer starts with a specification which characterises a set of allowable behaviours for a program. The developer then applies 'refinement laws' to incrementally transform the original specification statement into a concrete program. This approach can be methodologically contrasted with the 'invent-and-verify' program development methods based on the refinement calculus, such as VDM [51] or the B method [1]. Here, development again starts with an initial specification, and the result is executable code. However, the transformations here are rather larger, usually involving the data-refinements of whole modules, rather than the procedural refinement of individual statements.

The refinement calculus can be used in other ways. King and Arthan's compliance notation tool [6, 54] utilises much the same refinement laws as Morgan's calculus. However, instead of seeing the refinement calculus as the basis of program development, King and Arthan use it as a way of structuring the presentation of the verification of code, i.e. demonstrating the compliance of code with its specification. Another application which uses a similar semantic framework to the refinement calculus is Digital's experimental Extended Static Checking system [61]. This system is not meant for the derivation of programs satisfying characteristic specifications, but rather for automatically checking that programs satisfy specific kinds of simple properties.

1.3.2 Mechanisations of Program Refinement

The use of refinement for the development of large programs has been limited, partly because of a lack of suitable tools. This situation may be changing, as a number of experimental refinement tools are now being developed. All refinement tools provide mechanisms for system and property description; and most refinement tools provide at least ad-hoc support for the generation and proof of proof-obligations arising out of the application of refinement laws.

In the refinement calculus, formulae and programs are not separable; specification statements (which are programs) contain formulae, and data-refinement rules have proof-obligations involving the refinement of programs. Refinement tools must provide at least some support for manipulating both programs and formulae, and so there is great appeal in tools using same-system theorem provers for proof obligation generation and proof.

There are various extant refinement tools which support a methodology of progressive incremental transformation. The RED tool [50] was one of

the earliest refinement tools. It had ad-hoc generation of proof obligations, and almost no support for proof. Tredoux's [97] mechanisation of a theory of predicate transformers was a deep embedding in the HOL theorem prover. The RRE tool [80] was another early system which introduced refinement tactics and scheme variables in program refinement. It had ad-hoc generation and proof of proof obligations. The UQPRT project [28, 29, 27] uses the Ergo theorem prover [99] for same-system proof obligation generation and proof. It has a shallow embedding of its refinement language, with its semantics given axiomatically. The Refinement Calculator project [24] is based on the HOL theorem prover [37] for same-system proof obligation generation and proof. It uses a shallow embedding of its refinement language, with its semantics given definitionally. Tools for the B method [1] include Pratten's AMN tool [88], the B Toolkit [59]. Tools for VDM-related methods include μral [52], RAISE [89], and IFAD's VDM-SL Toolbox. Most of these systems use ad-hoc proof obligation generation, and satisfy proof obligations using external theorem provers. Finally, Grundy's refinement tool [42] uses an unusual semantics based on a three-valued logic, but is notable in demonstrating the utility of window inference [90] for refinement tools. Window inference is now used in the Refinement Calculator, the UQPRT, and in this dissertation.

1.4 The Isabelle Theorem Prover

Isabelle [85] is a generic theorem prover in the LCF [35] family of theorem provers implemented in the ML language [66]. It is generic in the sense of being a logical framework which allows the axiomatisation of various object logics. Isabelle also provides a collection of extensible proof tools which are instantiable for specific object logics. Isabelle's object logics include Isabelle/ZF, which is the main logic used in this dissertation. A brief overview of LCF theorem provers, Isabelle's meta-logic, and the object logic Isabelle/ZF is provided below. Aspects of Isabelle not discussed here include its theory definition interface, theory management, tactics and tacticals, generic proof tools, its goal-directed proof interface, advanced parsing and pretty-printing support, and its growing collection of generic decision procedures.

Isabelle/ZF was useful for work in this dissertation because:

- it has a well developed collection of mathematical results;
- it has a collection of powerful and extensible tactics and proof tools, which allow us to develop tactic-driven refinement tools [40];

- as an LCF theorem prover, it is a safe framework for developing new inference systems, such as that described in Chapter 2; and

- it is untyped, which is important for work introduced in Chapter 5.

1.4.1 LCF Theorem Provers

Theorem provers in the LCF family represent theorems of a logic by values of an abstract datatype in their implementation language. The only way to construct values of this type is by appeal to functions in the datatype which correspond to axioms and primitive inferences rules of a logic. The implementation of this abstract datatype forms the 'core' of an LCF theorem prover. A small core can be readily inspected for correctness. Then, the soundness of the theorem prover relies only upon the type safety of the implementation language. A strong type discipline will prohibit invalid 'theorems' from arising. LCF theorem provers have traditionally been implemented in variants of the ML language, which is one of the few languages with a formalised semantics and a demonstrably sound type system.

LCF theorem provers provide a safe foundation for extensible theorem prover development. Extensions to the theorem prover need not be overly concerned with soundness, as this will be guaranteed by the core. LCF theorem provers use this approach in the construction of reasoning environments, tactic languages, and other proof tools.

1.4.2 Isabelle's Meta-Logic

As a generic theorem prover, Isabelle can represent a wide variety of logics. Isabelle represents both the syntax of its meta-logic and the syntax of its object logics in a term language which is a datatype in ML. The language provides constants, application, lambda abstraction, and bound, free and scheme variables. Scheme variables are logically equivalent to free variables, but may be instantiated during unification. These are readily utilised in the development of prototype refinement tools supporting meta-variables [81].

Isabelle's meta-logic is an intuitionistic polymorphic higher-order logic. The core datatype of `thm` implements the axiom schemes and rules of this meta-logic. Isabelle provides the following constants in its meta-logic which are used to represent the rules and axioms of object logics:

Meta-level equality $A \mathrel{\widehat{=}} B$, is used to represent definitions of an object logic.

Meta-level implication $A \Longrightarrow B$, is used to represent rules of an object logic. Multiple hypotheses $A \Longrightarrow (B \Longrightarrow ...(C \Longrightarrow H)...)$ are sometimes represented using syntactic sugar $[\![A;\ \ B;\ \ ...\ C]\!] \Longrightarrow H$. In this dissertation we will sometimes write single or multiple hypothesis implications as follows:

$$\frac{A}{B} \qquad \frac{A \quad B \quad ... \quad C}{H}$$

Nested meta-level implications are possible: in alternation they correspond to either assumptions or obligations.

Meta-level universal quantification $\bigwedge x.\ P(x)$ is used to represent variable-capture side-conditions in the statement of rules or axiom-schemes in an object logic.

Meta-level function application $F(x)$ is used to represent the application of an operator F to an operand x.

Meta-level function abstraction $\lambda x.\, F(x)$ is used to represent the construction of parametric operators.

An Isabelle object logic is specified by declaring constants and their syntax, and by stating the axioms and inference rules for the logic. Isabelle implements a core datatype `thy` of theories, which is used to manage theory extension, and version control after the redefinition of theories. Definitional or axiomatic extensions to theories are possible. In Isabelle, object logics and theories are treated in the same manner. Object logics differ from most theories in providing non-definitional axioms and rules.

1.4.3 Isabelle/ZF

Isabelle/ZF is an Isabelle object logic for untyped set theory. It is based on an axiomatisation of standard set theory in first order logic. The defined meta-level type of sets i is distinct from the meta-level type of first-order logic propositions o. Families of sets can be defined as meta-level functions from index sets to result sets, and operators can be defined as meta-level functions from argument sets to result propositions. From this basis, a large collection of constructs are defined and theorems about them proved. Isabelle/ZF also provides a mechanism for defining recursive datatypes and inductively-defined relations.

Some Isabelle/ZF notation which is frequently used in this thesis includes: $a : A$ or $a \in A$ for set membership, $\mathbb{P}\, A$ for the powerset operator, $A \subseteq B$ for

subset, and $\{x : A \mid P(x)\}$ for set comprehension of elements of A satisfying P. Appendix A lists all the mathematical notation from Isabelle/ZF used in this thesis.

1.5 Outline of this Dissertation

Program refinement proceeds by transformation, and Chapter 2 presents a flexible framework for transformational proofs. This framework is a version of window inference [90] which is more general in allowing transformation under non-preorder relations and 'window opening' at multiple points.

Chapter 3 describes the mechanisation, in Isabelle/ZF, of a set-transformer presentation of a weakest precondition semantics for the refinement calculus. Chapter 4 describes how we can use Isabelle's meta-logic to lift this set transformer semantics to a predicate transformer semantics which has the expressive power of untyped set theory, and much of the clarity of simple type theory.

The work in Chapters 3 and 4 is done with a completely general state-type. Chapter 5 specialises this state-type to represent named program variables. This allows us to define variable assignment, framed specification statements, local blocks and parameterisation. This representation makes use of the expressive power of Isabelle/ZF. Chapter 6 echoes Chapter 4, lifting these set transformer statements to predicate transformers, but taking advantage of the additional structure on our state types. We also describe our method for handling procedures and recursive procedures.

Chapter 7 describes definitions and rules for data-refinement. Chapter 8 provides a case-study of the use of the mechanised refinement theory in the development of a program which tests the validity of formulae in propositional logic. Chapter 9 concludes with a summary of the dissertation, arguments for the main theses, and an indication of future work.

Finally, all of the theorems and rules stated in this thesis have been proved in Isabelle/ZF, using theories containing only definitional extensions to the standard logic. The proofs of these theorems and rules have not been given in this dissertation, as they are generally straightforward; usually by appeal to definitions, earlier results, and completely standard mathematical techniques. However, they do appear on the web at:

`http://www.cl.cam.ac.uk/users/ms204/settrans/`

In this thesis, the point of interest is not in the proofs of theorems and rules, but rather in their statement, and especially how elements of the refinement calculus are represented in our theory.

Chapter 2

Contextual Transformational Reasoning

Window inference is a style of reasoning which provides the problem decomposition of natural deduction and the sequent calculus, and the intuitive transformational style of equational reasoning. We give an overview of early versions of window inference, and lead into a generalised form of window inference which allows the transformation of windows under non-preorder relations, side-conditions on window opening and window transformation, and window opening at multiple locations. The form of window inference rules discussed here motivates many of the refinement theorems given later in this dissertation.

Natural deduction and the sequent calculus are two styles of reasoning familiar to logicians. They are analytic, providing mechanisms for problem decomposition and easy access to logical context. They have been central to proof theory, and have a role in taught courses on logic. However, although they were meant to capture the essence of the use of logic by mathematicians, in practice they are not used for proof. They both encourage a reductionist style of proof in which it is easy to lose sight of 'where sub-goals come from'.

Equational reasoning is an alternative style of reasoning, involving the progressive transformation of terms or formulae under equivalence relations [38]. It is an arguably more natural style of reasoning, familiar to students from proofs in secondary-school algebra. However, equational reasoning suffers by not providing any mechanism for problem decomposition nor any easy way to access context when transforming sub-terms.

Window inference [90] has many of the benefits of natural deduction/the sequent calculus, and equational reasoning. It is a hierarchical transformational style of reasoning which provides mechanisms both for progressive

transformation of terms and contextual problem decomposition. Window inference was initially intended to support equational reasoning [90], but was later generalised by Grundy to admit preorder relations [41, 43]. This generalisation allowed window inference to be used to reason about the program refinement relation. I have previously proposed further generalisations to window inference, to admit arbitrary composable relations and allow window opening at multiple points [96]. We re-present that work here. These generalisations were later independently reported by von Wright, along with several other extensions to window inference [103].

Grundy's refinement tool [42] demonstrated the utility of window inference for refinement tools. Other current refinement tools [102, 29] now also use window inference. The window inference rules discussed in this chapter motivate the form of the refinement rules we present in later chapters. Underlying window inference are relation composition theorems for combining a sequence of transformations. For program refinement this is the transitivity of refinement as seen in Section 4.2, and for data-refinement this is the composition of data-refinement with procedural refinement, as seen in Section 7.3 The main step in a window inference proof is to apply a window transformation rule, or in our terminology below, to transform at a locus. For program refinement, these transformation rules motivate the form of the statement introduction rules we list in Section 4.2 and Section 6.4. Window opening rules (or loci) are used to decompose the structure of a problem. For program refinement, these rules motivate the form of the refinement monotonicity theorems we have listed in Appendix C.

2.1 Window Inference

Window inference maintains a hierarchy of problem decomposition during a proof—while a problem can be readily decomposed into smaller problems, the information present in the original problem isn't lost. Sub-problems (represented as 'windows') are treated one at a time, and maintained in a window stack. The transformation of sub-problems may give rise to auxiliary side-conditions. Window inference is in some ways similar to conditional contextual rewriting, but is intended to support interactive proof by humans, and supports transformation under non-equivalence relations.

2.1.1 Window Inference as Natural Deduction

Like natural deduction or the sequent calculus, window inference is a generic style of inference applicable across a wide variety of logics. There are several

ways of presenting logics for window inference. The original presentation [90] used logical rules acting over values describing windows. A notion of window equivalence was given, and used to justify the soundness of the window inference rules. Grundy's presentation [43] instead described window inference rules by translation to natural deduction. This provided an easy soundness argument by appeal to the soundness of natural deduction, and also aided in (and was motivated by) the implementation of window inference in theorem provers supporting natural deduction.

Grundy's presentation is based on forward reasoning, with windows represented by theorems. Another presentation of window inference as natural deduction uses backward reasoning, with windows represented by proof-goals containing instantiable scheme variables for the final term [95]. This is in many ways closer to the initial presentation of window inference than Grundy's: in the backward version, 'opening rules' actually apply at opening, whereas in Grundy's forward version, 'opening rules' apply at window closing.

We will now review window inference by translation to natural deduction. This presentation uses forward reasoning, but differs from Grundy's in allowing side-conditions on window opening rules. Our presentation leads into the formal description of flexible window inference in Section 2.2.

Windows

The central element of window inference is the *window*, which contains a term of interest called the *focus*, a relation under which the focus is being transformed, and contextual information which can be used in the transformation of the focus. A window is represented by a theorem of the form:

$$\Gamma \Longrightarrow R(F_0, F_n)$$

where R is a preorder relation between an initial focus F_0 and our current focus F_n. Γ are our global and contextual assumptions.

Initial Window

When starting a window inference proof, we supply an initial focus F_0, a preorder relation R, and any global assumptions Γ. The initial window is represented by the reflexivity of R:

$$\Gamma \Longrightarrow R(F_0, F_0)$$

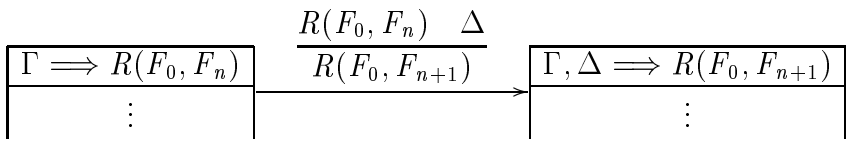


Figure 2.1: Window Transformation

Window Transformation

A window $\Gamma \Longrightarrow R(F_0, F_n)$ can be transformed by transforming its focus F_n to a new focus F_{n+1}. A transformation theorem:

$$\frac{\Delta}{R(F_n, F_{n+1})}$$

is used in conjunction with the transitivity of R to construct a rule:

$$\frac{R(F_0, F_n) \quad \Delta}{R(F_0, F_{n+1})}$$

Resolving the main antecedent of this rule against the theorem for the original window results in a transformed window represented by a theorem $R(F_0, F_{n+1})$ with new side-conditions Δ. We depict window transformation in Figure 2.1.

Window Stacks

A *window stack* records the current problem decomposition hierarchy. The top window of the stack describes the current problem, and the bottom window of the stack describes the initial problem. In window inference, only the top window in the window stack can be changed. A window stack can be represented by a stack of theorems.

Window Opening and Closing

To further de-compose the current problem, we may open a new window on a selected sub-term of the current focus. The new window is pushed onto the window stack. Its focus will be the selected sub-term, and the new relation and any additional context are governed by the *opening rule* used. If a window represented by a theorem $\Gamma \Longrightarrow R(F_0, F[f])$ is opened at the sub-term f with chosen preorder r, a new sub-window is pushed onto the window stack. This sub-window is represented by the reflexivity of r, with both the parent's assumptions Γ, and any new contextual assumptions Φ:

$$\Gamma, \Phi \Longrightarrow r(f, f)$$

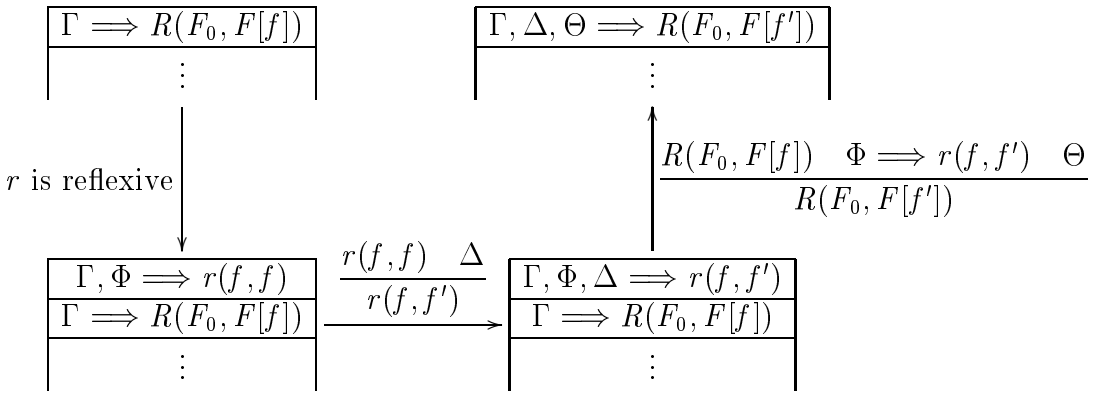


Figure 2.2: Window Opening, Transformation, and Closing

The new context Φ is determined by inspecting an opening rule of the form:

$$\frac{\Phi \Longrightarrow r(f, f') \quad \Theta}{R(F[f], F[f'])}$$

Though called an 'opening rule', it is applied to transform the parent window when the child window is closed.

In a window stack deeper than one window, the top window may be closed. The top window is popped from the window stack and its focus replaces the sub-term in the originally selected sub-term in its parent window. The opening rule is used with the transitivity of the parent relation R to construct a rule:

$$\frac{R(F_0, F[f]) \quad \Phi \Longrightarrow r(f, f') \quad \Theta}{R(F_0, F[f'])}$$

Resolving the antecedents of this rule against the parent window and the transformed sub-window results in the transformation of the parent window to $R(F_0, F[f'])$, with new side-conditions Θ. Window opening, (sub) window transformation, and window closing are depicted in Figure 2.2.

Transform Context

The context of the top window can be transformed. Given a window:

$$A, \Gamma \Longrightarrow R(F_0, F)$$

the context A can be transformed by appealing to a transformation theorem $B, \Gamma \Longrightarrow A$, resulting in a new window:

$$B, \Gamma \Longrightarrow R(F_0, F)$$

The transformation theorem can itself be constructed using window inference techniques, starting with initial focus A, initial context Γ, and transforming under the relation $\Longleftarrow$.

Opening Rule Selection and Rule Re-Use

When opening a window on a sub-term in the current focus, the opening rule used determines the relation and context for the sub-window. However, if a collection of opening rules is maintained, an appropriate opening rule can be implicitly selected, so that a user need only specify the position at which the new window should be opened. The automatic selection of window opening rules is an important part of the 'user interface' of window inference. It is not unproblematic—for any specific position in a focus there may be many applicable opening rules. Grundy [42] outlines one way of dealing with this issue: to heuristically select a single 'best' window opening rule.

Given a collection of opening rules, it is possible to implicitly derive new opening rules. Grundy [42] proposes various ways of implementing rule re-use, including using window opening composition, relation strength and lifted relations. The Ergo theorem prover's window inference facility uses relation strength, relation inverses, and also provides a wide variety of possible ways to matching against rules [99]. The most important of all of these mechanisms is the implicit composition of window opening rules in order to allow window opening at deep sub-terms. The collection of window opening rules maintained will usually all open on immediate sub-terms. The implicit composition of window opening rules will allow a user to open on a deeper sub-term, while increasing the size of the window stack.

The problem of selecting between multiple opening rules is exacerbated by derived opening rules. However, the process of derivation will often supply clues as to the relative strength of the derived rules, aiding in the heuristic solution of the 'best' rule.

2.2 Flexible Window Inference

Earlier versions of window inference required transformation relations to be preorders. This means that they could not be used to reason about data-refinement. Moreover, the problem decomposition provided by window inference only allows users to work on one sub-problem at a time. The restriction is not readily apparent in the linearised presentation of completed window inference proofs. However, in the exploratory development of a window inference proof, users may wish to survey more than one sub-problem at a time,

or to quickly alternate between working on different sub-problems. Flexible window inference [96] as re-presented here allows the transformation of terms under arbitrary composable relations, and allows the contextual survey of many sub-problems simultaneously.

The framework proposed here should not be regarded as a specific user interface for inference. Flexible window inference is a design space for hierarchical transformational inference systems. For example, simple constraints on flexible window inference can reduce it to an implementation of window stacks or window trees. Further experimentation and HCI research is required to determine versions of flexible window inference which are suitable for specific inference applications.

2.2.1 Flexible Window Inference as Natural Deduction

Just as we presented standard window inference by translation to natural deduction in Section 2.1.1, we now present flexible window inference in terms of natural deduction.

Proof State and Loci

In Section 2.1.1, the state of a window inference proof was represented as a stack of theorems implicitly joined by opening rules. Here, the state of a flexible window inference proof is represented by a main theorem, and an unstructured collection of opening rules called *loci*:

1. The main theorem $\Gamma \Longrightarrow R(F_0, F_n)$ describes the transformation, under some relation R, of the initial term F_0 to the current top-level focus F_n, with global assumptions Γ.

2. Each locus L_i provides a view upon the state of the proof, and is of the form:

$$\frac{\Phi_i \Longrightarrow r_i(f_i, f_i') \quad \Theta_i}{R_i(F_i[f_i], F_i[f_i'])}$$

   where $F_i[f_i]$ matches the current top-level focus. A locus provides access to a window, opening onto a specific position with focus f_i and providing additional context Φ_i. If this focus is transformed to f_i', the locus will transform the main focus $F_i[f_i]$ to $F_i[f_i']$ and generate new side-conditions Θ_i.

Note that something must be done if a locus is rendered ill-formed by the transformation of the state at another locus, but at this level of abstraction, no action is specified. Loci management policies may be implemented at higher levels of the user interface in order to avoid or handle this problem.

Relations are not required to be transitive, but in order to transform the top-level focus using some locus L_i, the main relation R must at least *compose* with the relation R_i to give some new top-level relation R'. A collection of relation composition theorems is maintained and will be appealed to implicitly in the course of a flexible window inference proof.

Initial Window

To start a new flexible window inference proof, the user supplies an initial focus F_0, and initial global assumptions Γ. The initial proof state is:

1. an initial top-level theorem $\Gamma \Longrightarrow F_0 \mathrel{\widehat{=}} F_0$, and

2. an empty collection of loci.

Add or Remove a Locus

A locus may be added to or removed from the collection of loci. The main theorem is unchanged.

Transform at a Locus

A user may choose to transform the focus accessed by a locus in the collection of loci. This results in a change to the main theorem, but the collection of loci is unchanged. To effect a transformation, the user supplies a transformation theorem $\Phi \Longrightarrow r(f, f')$ and nominates a locus, which will be of the form:

$$\frac{\Phi \Longrightarrow r(f, f') \quad \Theta}{R'(F[f], F[f'])}$$

An appropriate relation composition theorem:

$$\frac{R(F_0, F_n) \quad R'(F_n, F) \quad \Psi}{R''(F_0, F)}$$

is chosen to change the state $R(F_0, F[f])$ to a new state $R''(F_0, F[f'])$ with additional side-conditions Θ and Ψ.

Transform a Locus

A locus may be changed by transforming its context or side-conditions. This results in a change to one locus in the collection of loci, but the main theorem is unchanged. The locus to be transformed will be of the form:

$$\frac{A, \Phi \Longrightarrow r(f, f') \quad B, \Theta}{R(F[f], F[f'])}$$

To change the side-condition of a locus, the user supplies a transformation theorem $B', \Theta \Longrightarrow B$. This can be generated by window inference techniques by transforming an initial focus B with context Θ to a final focus B' under the $\Longleftarrow$ relation. The nominated locus is replaced by a new locus:

$$\frac{A, \Phi \Longrightarrow r(f, f') \quad B', \Theta}{R(F[f], F[f'])}$$

To change the context at a locus, the user supplies a transformation theorem $A, B, \Phi, \Theta \Longrightarrow A'$. This can be generated by window inference techniques by transforming an initial focus A with context B, Φ, Θ to a final focus A' under the $\Longrightarrow$ relation. The nominated locus is replaced by a new locus:

$$\frac{A', \Phi \Longrightarrow r(f, f') \quad B, \Theta}{R(F[f], F[f'])}$$

2.2.2 Constraints: Window Stacks and Window Trees

We can implement window stacks or window trees in this general framework by imposing constraints on the maintenance of loci. Each locus opens onto a specific position within the conclusion of the main theorem. We can use this position information to partially order the loci. The illusion of repeatedly transforming a single focus may be generated by repeatedly appealing to a single locus in order to re-compute the focus and context after each change to the main theorem.

So for a suitable definition of 'deepest locus', we will have an implementation of window stacks corresponding to existing versions of window inference if we:

1. only present the focus available through the deepest locus;
2. force all transformations to operate upon the deepest locus;
3. only allow the addition of loci at positions inside the deepest locus; and

4. only allow the removal of the deepest locus.

Similarly, for a suitable definition of 'leaf locus', we will have an implementation of window trees if we:

1. only present the focus available through leaf loci;

2. force all transformations to operate upon a leaf locus;

3. only allow the addition of loci at position inside a leaf locus; and

4. only allow the removal of leaf loci.

Window stacks or window trees may turn out to provide the basis for a human-friendly interface. Other, more flexible, approaches to accessing and maintaining loci may be required for an effective user interface application. Experimentation and usability studies are required.

2.2.3 Problems and Benefits

Flexible window inference is more general than earlier versions of window inference in several ways. It allows non-preorder relations, side-conditions on window openings, and multiple window openings. These generalisations bring with them new problems whose solution must be traded with any increase in flexibility. This is discussed below.

Non-Preorder Relations

Window inference was originally proposed to use only equivalence relations [90]. Grundy's implementation allowed for preorders [41], and Grundy reports that Robinson had suggested that reflexivity need not be a condition of relations used in window inference. Flexible window inference requires only that the relation in the main top-level theorem be composable with any relation generated by a used loci.

In Grundy's presentation of window inference, reflexivity is used to provide the initial theorem $r(f, f)$ for a sub-window. This is unnecessary even in Grundy's system, as the first transformation theorem $r(f, f')$ used on this sub-window could serve as the initial theorem for the window. If a sub-window is immediately closed after opening, then no change results to the parent window, and the uninitialised sub-window could simply be discarded. Flexible window inference does not maintain any sub-windows explicitly, and hence has no call for reflexivity of relations.

Grundy uses transitivity for two purposes: firstly at window closing, to justify the transformation of a parent window; and secondly prior to window closing, to transform each sub-window. Flexible window inference avoids this latter use of transitivity by immediately resolving all sub-term transformations to the top-level theorem. Moreover, only the top-level relation must be composable.

This extra generality exacerbates the problem of choosing between opening rules. By allowing our relations to be non-transitive, we allow them to compose with other relations in arbitrary ways, increasing the number of derived loci we can create for a position. In our presentation we have avoided this problem by sweeping it under the carpet—loci are added explicitly, rather than being chosen based on a new position.

Multiple Window Openings

The standard difficulty with opening at multiple foci independently is that it is not sound to use context from certain multiple window opening rules simultaneously. For example, consider the opening rules for transforming each side of a conjunction:

$$\frac{B \Longrightarrow A = A'}{A \wedge B = A' \wedge B} \qquad \frac{A \Longrightarrow B = B'}{A \wedge B = A \wedge B'}$$

In order to transform a focus term $A \wedge B$, we cannot both assume A to transform sub-focus B and simultaneously assume B to transform sub-focus A. Under flexible window inference, the context and focus for each subproblem is recomputed after each transformation of the main focus. So for example, if our main focus is $A \wedge B$ and our collection of loci are the above two opening rules, then after using context B to transform the sub-focus A to A', the derived context for the sub-focus on B would be A'.

Thus in this generalised setting, the effect of a transformation may be non-local, perhaps rendering other loci inapplicable. The transformation of the top-level theorem can render a locus invalid in a number of ways: by making the position in that locus not a valid position in the top-level term; by making the 'term with a hole' $F[-]$ not match the top-level term; or by making the relation R not compose with the relation in the top-level term. Stack and tree constraints such as those mentioned in Section 2.2.2 avoid or ameliorate this problem. However, other approaches may prove more suitable. For example, we could automatically removed ill-formed loci, or perhaps notify the user when a side-effecting transformation occurs.

Chapter 3

Set Transformers

We present a shallow embedding of a weakest precondition semantics for the program refinement language used in this thesis. Commands in the language are represented by set-transformers in the object logic of Isabelle/ZF. We also discuss the definitions of refinement and some healthiness conditions. We highlight how it is possible to sometimes extract 'types' from the arguments of constants defined in Isabelle/ZF. We also show how Boolean guards and other expressions in the refinement language can be represented by sets of states and object-level functions respectively. Finally, we define a novel interfaced recursion statement which supports the step-wise development of recursion blocks.

Weakest precondition semantics was first used by Dijkstra in his description of a non-deterministic programming language [31]. That language was generalised to the refinement calculus by Back [7], and later by Morris [75] and Morgan [71]. Textbook presentation of weakest precondition semantics usually represent conditions on states as formulae. However, such presentations can be unclear about the logic underlying these formulae. Mechanisations of the refinement calculus must perforce be precise, and two approaches have been taken. The most literal approach is to use a modal logic with program states forming the possible worlds. Work done in the Ergo theorem prover [99] is in this vein [28]. However, support for modal logics is currently unwieldy in most theorem provers. More importantly, using a modal logic hampers the re-use of results from existing classical mathematics.

Another approach is to represent program states explicitly in a classical logic. Conditions can then be represented as collections of states. For example, Agerholm [2] (and later the Refinement Calculator project [102] and others [86, 57]) used predicates (functions from states to Booleans) in higher-order logic to represent sets of states. Our approach uses this kind

of representation, but as we are working in untyped set theory, we represent conditions by sets of states. This chapter demonstrates that this kind of shallow embedding can be done in a first-order setting.

The refinement language presented here is expressed in terms of a completely general state type. We will postpone until Chapter 5 the discussion of statements which use program variables. That chapter will also reveal why we have chosen to work in a very expressive logic such as Isabelle/ZF.

Methodologically, our calculus is intended to support program refinement as presented in Morgan's influential textbook [73]. In particular, we provide specification statements [71] and value-binding statements which are not used in the Refinement Calculator project. We also describe how to support the stepwise refinement of recursion blocks with the use of interfaces and embedded contextual information.

3.1 Set Transformers: A Shallow Embedding

Weakest precondition semantics is usually presented by inductively defining an interpretation $wp(c, q)$ over a language of refinement language statements c and arbitrary postconditions q. However, the mechanisation described here uses a shallow embedding—statements are identified with abbreviations for their semantics. Instead of defining an interpretation operator wp, we instead define each statement as an abbreviation for a specific set transformer from a set of states representing a postcondition to a set of states representing the weakest precondition.

Henceforth, we write $\mathcal{P}_{A,B}$ for the function space of heterogeneous set transformers $\mathbb{P}\,A \to \mathbb{P}\,B$, and we write $\mathcal{P}_A$ for the homogeneous set transformers $\mathcal{P}_{A,A}$. Our 'user-level' refinement theory will ultimately be concerned only with homogeneous set transformers, but in the development of that theory we will sometimes consider heterogeneous set transformers.

3.1.1 Skip and Sequential Composition

To illustrate the shallow embedding of the statements in ZF, we define[1] the skip and sequential composition statements as follows:

$$\begin{aligned} \mathsf{Skip}_A &\mathrel{\widehat{=}} \lambda\, q{:}\mathbb{P}\,A.\, q \\ a; b &\mathrel{\widehat{=}} \lambda\, q{:}\mathrm{dom}(b).\, a`(b`q) \end{aligned}$$

[1] Remember that the back-quote operator ` is infix function application in Isabelle/ZF. Appendix A contains a summary of the notation used in this dissertation.

Note that in the definition of the skip statement we explicitly require an operand A to represent the state type. In simple type theory this could be left implicit, but as we are using ZF, we must explicitly bound the set constructed on the right hand side of the definition. However, in the definition of the sequential composition statement we haven't supplied a state type, because we can extract the state type from the structure of its sub-components. Here, the sub-statement b should be represented by a set transformer in $\mathcal{P}_{A,B}$ for some final state type A and initial state type B. In this case, the domain of b will be $\mathbb{P}\, A$ and so we have:

$$a; b \quad = \quad \lambda\, q{:}\mathbb{P}\, A.\, a`(b`q)$$

It easy to see that skip and sequential composition are set transformers. That is:

$$\mathsf{Skip}_A : \mathcal{P}_{A,A} \qquad \frac{a : \mathcal{P}_{B,C} \quad b : \mathcal{P}_{A,B}}{a; b : \mathcal{P}_{A,C}}$$

3.1.2 State Assignment and Alternation

The definition of skip and sequential composition illustrated the representation of set transformers in ZF, and demonstrated how we can sometimes extract the state type from a statement's components. We will now define the semantics of state assignment (which functionally updates the entire state), and alternation ('if-then-else') in order to illustrate the representation of expressions in our language. As it was natural to use sets of states to represent conditions on states, we similarly use sets of states to represent Boolean expressions within statements. To represent expressions of other types, we use ZF's standard representation of functions and relations as sets of pairs. State assignment and alternation are defined as follows:

$$\begin{array}{rcl} \langle F \rangle_A & \widehat{=} & \lambda\, q{:}\mathbb{P}\, A.\, \{s{:}\mathrm{dom}(F) \mid F`s \in q\} \\ \textsf{if } g \textsf{ then } a \textsf{ else } b \textsf{ fi} & \widehat{=} & \lambda\, q{:}\mathrm{dom}(a) \cup \mathrm{dom}(b). \\ & & \quad (g \cap a`q) \cup ((\bigcup(\mathrm{dom}(a) \cup \mathrm{dom}(b)) - g) \cap b`q) \end{array}$$

Note that in these definitions, we again extract the state type from the structure of sub-components. However, in the definition of state assignment, we extract the state type not from a sub-statement, but instead from the expression F, which should be a state-function from some state type B to state type A. In the definition of alternation we take the union of the domains of a and b, because the final state q may be reached through either a or b. We also extract the actual underlying state type A, using the fact that

$\bigcup(\mathbb{P}\,A) = A$. So when $F : B \to A$ and $a, b : \mathcal{P}_{A,B}$, we have the simpler equalities:

$$\begin{array}{rcl} \langle F \rangle_A & = & \lambda\, q{:}\mathbb{P}\,A.\, \{s{:}B \mid F`s \in q\} \\ \mathsf{if}\ g\ \mathsf{then}\ a\ \mathsf{else}\ b\ \mathsf{fi} & = & \lambda\, q{:}\mathbb{P}\,A.\, (g \cap a`q) \cup ((A - g) \cap b`q) \end{array}$$

It is easy to see that state assignment and alternation are set transformers:

$$\frac{F : B \to A}{\langle F \rangle_A : \mathcal{P}_{A,B}} \qquad \frac{a : \mathcal{P}_A \quad b : \mathcal{P}_A}{\mathsf{if}\ g\ \mathsf{then}\ a\ \mathsf{else}\ b\ \mathsf{fi} : \mathcal{P}_A}$$

You may wonder why I have (apparently arbitrarily) restricted alternation to be a homogeneous set transformer, but have taken the trouble of including an extra state type in the definition of state assignment, in order to make it a heterogeneous set transformer. This is because it suits our purposes: we are mainly concerned with developing a theory of refinement laws over homogeneous set transformers, and so our definition of alternation will do. However, we will need recourse to a heterogeneous state-assignment statement in Chapter 5, and so define it here.

The equalities above correspond fairly closely to standard definitions of state assignment and alternation [1, 13]. Back and von Wright's [16] alternative definition of state assignment in terms of relation image can be proved as a consequence of our definition. For a state function $F : B \to A$ we have: $\langle F \rangle_A = \lambda\, q{:}\mathbb{P}\,A.\, F^{-1}\ \text{“}\ q$.

Our definitions also receive some measure of validation by the proof of theorems about our language which we would expect to be true. For example, the skip statement is the identity state-assignment, and the sequential composition of state assignments is the assignment of their compositions. That is, for $F : A \to B$ and $G : B \to C$ we have:

$$\mathsf{Skip}_A = \langle \lambda\, s{:}A.\, s \rangle_A \qquad \langle F \rangle_B ; \langle G \rangle_C = \langle G \circ F \rangle_C$$

Similarly for alternation statements, we would expect and it is true that an alternation with identical branches is equivalent to its common branch, an alternation with a true guard (universal set) is equivalent to its first branch, and an alternation with a false guard (empty set) is equivalent to its second branch. That is, for $g : \mathbb{P}\,A$, and $a, b, c : \mathcal{P}_A$, we have:

$$\mathsf{if}\ g\ \mathsf{then}\ c\ \mathsf{else}\ c\ \mathsf{fi} = c \quad \mathsf{if}\ A\ \mathsf{then}\ a\ \mathsf{else}\ b\ \mathsf{fi} = a \quad \mathsf{if}\ \varnothing\ \mathsf{then}\ a\ \mathsf{else}\ b\ \mathsf{fi} = b$$

3.2 Refinement and Healthiness

Program refinement takes place in a wide-spectrum language which contains elements which are non-deterministic, non-terminating, and even non-executable. The refinement ordering on programs reduces both non-determinism and possibilities for non-termination. It preserves correctness, and so plays a central role in the transformation of abstract specifications into correct and executable implementations. The statements we define in this thesis are all monotonic set transformers. Monotonicity is one of Dijkstra's original 'healthiness conditions' [31]. In this section we present our definition of refinement and correctness, describe monotonicity and monotonic set transformers, and finally touch on other healthiness conditions.

3.2.1 Refinement and Total Correctness

We define refinement in a fairly standard manner [15], as follows:

$$a \sqsubseteq b \;\widehat{=}\; \forall\, q{:}\mathrm{dom}(a).\; a`q \subseteq b`q$$

That is, for $a : \mathcal{P}_{A,B}$

$$a \sqsubseteq b \;\Leftrightarrow\; \forall\, q{:}\mathbb{P}\, A.\; a`q \subseteq b`q$$

If $a \sqsubseteq b$, then the set of states in which we can execute b is larger than the set of states in which we can execute a. Our definition uses the domain of a for safety, so that b will have at least the behaviour guaranteed for anything which can be safely executed by a.

We define total correctness in a standard way [9], as follows:

$$\{p\}\ c\ \{q\} \;\widehat{=}\; p \subseteq c`q$$

That is, if $\{p\}\ c\ \{q\}$, then p is contained in the set of initial states for which we are allowed to execute c and terminate in a state in q. The refinement relation preserves total correctness. For $a, b : \mathcal{P}_{A,B}$,

$$a \sqsubseteq b \;\Leftrightarrow\; \forall\, p{:}\mathbb{P}\, A\ q{:}\mathbb{P}\, B.\, \{p\}\ a\ \{q\} \Rightarrow \{p\}\ b\ \{q\}$$

3.2.2 Monotonic Set Transformers

Monotonicity plays an important role in the development of recursion in the wide-spectrum language. The condition of monotonicity is defined as follows:

$$\mathsf{monotonic}(c) \mathrel{\widehat{=}} \forall\, a\ b{:}\mathrm{dom}(c).\; a \subseteq b \Rightarrow c`a \subseteq c`b$$

That is, for $c : \mathcal{P}_{A,B}$ we have:

$$\mathsf{monotonic}(c) = \forall\, a\ b : \mathbb{P}\, A.\ a \subseteq b \Rightarrow c`a \subseteq c`b$$

Within our ultimate 'user-level' theory of program refinement, we will operate solely within a space of monotonic set transformers. We can define the set of heterogeneous monotonic set transformers as follows:

$$\mathcal{M}_{A,B} \ \widehat{=}\ \{c : \mathcal{P}_{A,B} \mid \mathsf{monotonic}(c)\}$$

We write $\mathcal{M}_A$ for the homogeneous monotonic set transformers $\mathcal{M}_{A,A}$.

3.2.3 Other Healthiness Conditions: Refining to 'Code'

The refinement methodology described here aims to transform specification statements until only 'code' remains. However, just what is executable code? In the literature, the question of executability is usually held indeterminate, in order to allow flexibility in the sophistication of our implementation language. Dijkstra's healthiness conditions go some way in restricting predicate transformers to executable code. However even the full complement of healthiness conditions is not sufficient. For example, in general we won't be able to execute a complicated specification statement which happens to be mathematically equivalent to some executable statement.

Nonetheless, at the very least a final program will be strict, i.e. it will not recover from an aborted state. Morgan and Vickers recommended a check for strictness (or 'feasibility') at the end of a refinement development [69]. In their setting, this check corresponds to a check for type correctness. However, our semantics maintains a type-correct program throughout the development. We also now introduce the termination healthiness condition, which is dual to strictness. That is, if establishing any final state will do, then one may can safely start (and be assured of termination) from any initial state. Our standard definitions [13] of strictness and termination are as follows:

$$\mathsf{strict}(c) \ \widehat{=}\ c`\varnothing = \varnothing$$

$$\mathsf{terminating}(c) \ \widehat{=}\ c`(\bigcup \mathrm{dom}(c)) = \bigcup \mathrm{dom}(c)$$

That is, for $c : \mathcal{P}_{A,B}$

$$\mathsf{terminating}(c) \Leftrightarrow c`A = A$$

We won't discuss the other healthiness conditions, as they are not used in this thesis. For a general discussion and categorisation of healthiness conditions from a lattice-theoretic perspective, see [12].

3.3 The Refinement Language

We have seen the definitions for the skip, sequential composition, state assignment and alternation statements in Section 3.1. Appendix B contain the definitions of these and other atomic and compound statements. These statements are all (monotonic) set transformers, as indicated in Appendix C. The remainder of this section provides a brief commentary on individual definitions, and presents theorems which help to validate our definitions.

3.3.1 Atomic Statements

Abort, Magic, and Chaos

The abort and magic statements Abort_A and Magic_A are the worst and best statements respectively. Magic is an unimplementable statement which always terminates, and satisfies any required post-condition, including impossible ones such as $\varnothing$. Abort will never establish any required postcondition, except the impossible one $\varnothing$, and can be thought of as a non-terminating statement. The chaos statement Chaos_A is is just about the worst possible statement—it will always terminate, but it may change the program state in any manner whatsoever. Set transformers form a complete lattice under refinement, and magic and abort are the least and greatest refinements. For $b : \mathcal{P}_A$ we have:

$$\mathsf{Abort}_A \sqsubseteq a \qquad b \sqsubseteq \mathsf{Magic}_A$$

Magic and abort are also left-idempotent. For $a, b : \mathcal{P}_A$, we have:

$$\mathsf{Abort}_A ; a = \mathsf{Abort}_A \qquad \mathsf{Magic}_A ; b = \mathsf{Magic}_A$$

Magic and abort are right-idempotent given conditions on the left statement. For $a, b : \mathcal{P}_{A,B}$ with $\mathsf{strict}(a)$ and $\mathsf{terminating}(b)$, we have:

$$a ; \mathsf{Abort}_A = \mathsf{Abort}_A \qquad b ; \mathsf{Magic}_A = \mathsf{Magic}_A$$

Assert, Guard, and Non-Deterministic Assignment

An assertion statement $\{P\}_A$ behaves like a skip statement when it is started in a state which satisfies P, and otherwise aborts. The guard statement (which Morgan calls 'coercion') $(Q)_A{\rightarrow}$ will terminate establishing a state in Q, but will not change the state. The non-deterministic assignment $[Q]_A$ is similar to the guard statement, but may change the state. We would expect, and it is true that, a non-deterministic assignment statement is equivalent to a chaos statement preceding a guard statement:

$$[Q]_A = \mathsf{Chaos}_A ; (Q)_A{\rightarrow}$$

If we consider the special cases of these statements where the operand is either empty or total, we see that they are equal to one of the skip, chaos or magic statements. These theorems reinforce our intuition that guarding does not change the state, but that non-deterministic assignment may do:

$$\begin{array}{lll} \{\varnothing\}_A = \mathsf{Abort}_A & (\varnothing)_A{\rightarrow} = \mathsf{Magic}_A & [\varnothing]_A = \mathsf{Magic}_A \\ \{A\}_A = \mathsf{Skip}_A & (A)_A{\rightarrow} = \mathsf{Skip}_A & [A]_A = \mathsf{Chaos}_A \end{array}$$

Specification Statements

Morgan's refinement methodology [73] centres on the use of specification statements [71] which are transformed step-wise to an executable (and correct) implementation. A specification statement has some precondition P and postcondition Q. When executed in a state satisfying P it will terminate and establish a state satisfying Q, and otherwise it will abort. Morgan's specification statement includes a frame w of program variables which the statement is allowed to modify. We define a framed specification statement in Chapter 5, where we introduce our representation of program variables. For now, we consider only two specification statements: the fixed specification statement ${}^{\bot}[P,\ Q]_A$ which does not change the state, and the free specification statement ${}^{\top}[P,\ Q]_A$ which may change the state in any way at all. As might be expected, we can construct these statements out of the assert, guard and non-deterministic assignment statements. For $P : \mathbb{P}\, A$, we have:

$${}^{\bot}[P,\ Q]_A = \{P\}_A ; (Q)_A{\rightarrow} \qquad {}^{\top}[P,\ Q]_A = \{P\}_A ; [Q]_A$$

We can consider more general specification constructs: relational assertion $\{R\}_A$ and relational non-deterministic assignment $[R]_A$. When the relation in question is constructed by set comprehension, we will write these as $o =_A i.\, R(i, o)$ and $o :=_A i.\, R(i, o)$ respectively. Relational non-deterministic assignment is the main form of specification statement used in the Refinement Calculator [102]. All the atomic monotonic set transformers can be expressed in terms of these statements. For $P : \mathbb{P}\, A$, $Q : \mathbb{P}\, A$, and $F : A \rightarrow A$, we have:

$$
\begin{array}{rcl@{\qquad}rcl}
\langle F \rangle_A & = & o :=_A i.\, o = F`i & \langle F \rangle_A & = & o =_A i.\, o = F`i \\
(Q)_A{\rightarrow} & = & o :=_A i.\, o \in Q \wedge i = o & \{P\}_A & = & o =_A i.\, i \in P \wedge i = o \\
[Q]_A & = & o :=_A i.\, o \in Q & & & \\
{}^{\perp}[P,\ Q]_A & = & o :=_A i.\, i \in P \wedge o \in Q \wedge i = o & & & \\
{}^{\top}[P,\ Q]_A & = & o :=_A i.\, i \in P \wedge o \in Q & & &
\end{array}
$$

3.3.2 Choice Statements

We have already described one statement for choosing between two sub-statements: alternation. From a semantic perspective there are 'simpler' binary choice statements: demonic choice $\sqcap$ and angelic choice $\sqcup$. Demonic choice can be thought of as the operation of Murphy's Law, and will always choose the 'worst' of the possible statements. Angelic choice will always choose the 'best' one, and can be thought of as a form of backtracking, or as a user-directed choice. We can express alternation in terms of either demonic or angelic choice. For $g : \mathbb{P}\, A$ and $a, b : \mathcal{P}_A$, we have:

$$
\begin{array}{lcl}
\mathsf{if}\ g\ \mathsf{then}\ a\ \mathsf{else}\ b\ \mathsf{fi} & = & (g)_A{\rightarrow}; a \sqcap (A - g)_A{\rightarrow}; b \\
\mathsf{if}\ g\ \mathsf{then}\ a\ \mathsf{else}\ b\ \mathsf{fi} & = & \{g\}_A; a \sqcup \{A - g\}_A; b
\end{array}
$$

Rather than just considering binary choice, we may consider a generalised choice between a set of statements. We won't use generalised choice in our user-level theory of refinement, but it is useful in the comparison of various of our statements. Again, we have defined a demonic $\bigsqcap_A C$ and an angelic $\bigsqcup_A C$ form of generalised choice. Both forms of generalised choice specialise to their binary equivalents when restricted to two-element sets of statements. For singleton sets, they are equivalent to the single member statement, for empty sets they are equivalent to aborting statements, and for universal sets of (monotonic) set transformers angelic and demonic choice are equivalent to the magic and aborting statements respectively. For $a, b, c : \mathcal{P}_{A,B}$ we have:

$$
\begin{array}{lll}
\bigsqcap_A\{a, b\} = a \sqcap b & \bigsqcap_A\{c\} = c & \bigsqcap_A\varnothing = \mathsf{Abort}_A \\
\bigsqcup_A\{a, b\} = a \sqcup b & \bigsqcup_A\{c\} = c & \bigsqcup_A\varnothing = \mathsf{Abort}_A
\end{array}
$$

$$\bigsqcup_A \mathcal{P}_A = \mathsf{Magic}_A \qquad \bigsqcap_A \mathcal{P}_{A,B} = \mathsf{Abort}_A$$

$$\bigsqcup_A \mathcal{M}_A = \mathsf{Magic}_A \qquad \bigsqcap_A \mathcal{M}_{A,B} = \mathsf{Abort}_A$$

Generalised demonic choice allows us to revisit our intuition that the chaos statement is a terminating statement which may change any variable in any non-desirable manner. Chaos is equivalent to the demonic choice of all state assignment statements:

$$\mathsf{Chaos}_A = \bigsqcap_A \{\langle F \rangle_A . F : A \to A\}$$

3.3.3 Value Binding Statements

Some programming languages provide statements which bind values to variables. These variables need not be assignable program variables. For example, the ML language has patterns which bind values to non-assignable variables [66]. Morgan's refinement calculus makes heavy use of value binding statements, especially the logical constant statement [73]. His logical constant statements bind values to program variables, and are in fact dual to his local block statements. We present our quite different treatment of local program variables in Chapter 5. Nonetheless, in the spirit of Morgan's refinement calculus, we consider angelic and demonic value binding statements, which we call logical constant statements and logical variable statements respectively. These statements differ from those in Morgan's calculus, because they do not introduce bindings for program variables. Nonetheless, we will put our logical constants to the same use as Morgan's [73]: to 'remember' the initial value of program variables and the previous values of variant functions for recursion and loops. At the end of this section we discuss our store statement, which remembers the value of a state. We will make use of the store statement in defining variable localisation in Chapter 5.

We have unbounded and bounded versions of the logical constant statement: $\mathsf{con}\ x.\ c(x)$ and $\mathsf{con}_A\ x : T.\ c(x)$; and the logical variable statement: $\mathsf{var}\ x.\ c(x)$ and $\mathsf{var}_A\ x : T.\ c(x)$. In the unbounded versions, x is free to take on any value, whereas in the bounded versions, x must take on some value in the set of values T. As we are using a shallow embedding, we can appeal to the binding facilities available in our underlying logic—we will identify value binding statements in the refinement language with value-binding terms in Isabelle/ZF.

For compound statements such as sequential composition, we didn't have

to explicitly supply the state type, as we could 'extract' it given assumptions about sub-statements. It would be nice to do the same here, but instead of the sub-component being a statement, it is an indexed family of statements. For unbounded quantification, instead of using the ordinary domain operator, we can define an indexed domain operator, $\mathrm{Dom}(c)$ and use that to extract the state type from the family of statements. In order to define bounded value quantifiers, we could try to similarly define a bounded lifted domain operator. However, this would extract a sensible state type only when the bounding set T is non-empty. This would introduce somewhat artificial side-conditions for theorems about bounded value quantifiers. So, for bounded value quantifiers, we explicitly indicate the state type.

Bounded and unbounded value quantifiers are similar. If $c(x) : \mathcal{P}_A$ for any x, then we have:

$$\mathsf{con}\ x.\ \{\!\{\{s{:}A \mid\ x : T\}\}\!\}_A ; c(x) = \mathsf{con}_A\ x{:}T.\ c(x)$$
$$\mathsf{var}\ x.\ (\{s{:}A \mid\ x : T\})_A{\rightarrow}; c(x) = \mathsf{var}_A\ x{:}T.\ c(x)$$

Our definitions for bounded value quantifiers are equivalent to Pratten's demonic and angelic choice statements [88]. Bounded value quantifiers are equivalent to the generalised angelic or demonic choice of an indexed family of statements. If $c(x) : \mathcal{P}_{A,B}$ for any $x : T$, then we have:

$$\mathsf{con}_A\ x{:}T.\ c(x) = \bigsqcup_A \{c(x).\, x{:}T\}$$

$$\mathsf{var}_A\ x{:}T.\ c(x) = \bigsqcap_A \{c(x).\, x{:}T\}$$

We can use logical constants to relate the fixed and free specification statements. Here the logical constant fixes the state to be the same in the pre-condition and the post-condition of the free specification statement:

$${}^{\perp}[P,\ Q]_A \quad = \quad \mathsf{con}\ x.\ {}^{\top}[\{s{:}A \mid\ s = x \wedge s \in P\},\ \{s{:}A \mid\ s = x \wedge s \in Q\}]_A$$

We can also define both relational assertion and non-deterministic assignment in terms of logical constants and specification statements, as follows:

$$i =_A o.\, R(i,o) \quad = \quad \mathsf{con}\ x.\ {}^{\top}[\{i{:}A \mid\ x : A \wedge R(i,x)\},\ \{o{:}A \mid\ o = x\}]_A$$
$$i :=_A o.\, R(i,o) \quad = \quad \mathsf{con}\ x.\ {}^{\top}[\{i{:}A \mid\ x = i\},\ \{o{:}A \mid\ R(x,o)\}]_A$$

Finally, we introduce the definition of our store statement, which remembers the value of the state in which it is executed. It can be defined in two equivalent ways, as follows:

$$\mathsf{store}_A\ x.\ c(x) \quad = \quad \mathsf{con}\ x.\ \{\!\{\{s{:}A \mid\ s = x\}\}\!\}_A ; c(x)$$
$$\mathsf{store}_A\ x.\ c(x) \quad = \quad \mathsf{var}\ x.\ (\{s{:}A \mid\ s = x\})_A{\rightarrow}; c(x)$$

As we would expect, a skip statement is just like a program which remembers the initial state, executes any normal statement, and then restores the initial state. Formally, for a strict and terminating statement $c : \mathcal{P}_A$, we have:

$$\mathsf{store}_A\ x.\ c; \langle \lambda\, s : A.\, x \rangle_A = \mathsf{Skip}_A$$

3.3.4 Recursion and Loops

Recursion and looping constructs play an important role in any programming language. The most primitive recursion construct in the refinement calculus is the recursion block $\mathsf{re}_A\ N.\ F(N)\,\mathsf{er}$, which executes the statement F, recursing on N. Recursion blocks are not usually seen as statements in programming languages. However, we will use them in the analysis of recursive procedures in Chapter 6, and here we use them in the definition of while-do loops. Our definitions and theory of recursion follow Agerholm [2] and the Refinement Calculator project [102].

We can show that recursion blocks are least fixed points. That is, for $\mathsf{regular}_A(F)$ we have: $F(\mathsf{re}_A(F)\,\mathsf{er}) = \mathsf{re}_A(F)\,\mathsf{er}$. Regular statements are those whose bodies are well-typed and preserve refinement. Regularity is are formally defined in Appendix B. All atomic monotonic set transformers are regular, and all of our compound statements preserve regularity. The fixed point theorem leads to the proof of a recursion theorem using well-founded induction. For $\mathsf{regular}_A(F)$, $c : \mathcal{M}_A$, variant function V well-typed given invariant I, i.e. $\forall\, s : A.\, I(s) \Rightarrow V(s) : T$, and relation $R : \mathbb{P}(T \times T)$ well-founded on T, we have:

$$\frac{\forall\, x : T.\, \{\!\{ s : A \mid I(s) \wedge V(s) = x \}\!\}_A; c \sqsubseteq F(\{\!\{ s : A \mid I(s) \wedge R(V(s), x) \}\!\}_A; c)}{\forall\, x : T.\, \{\!\{ s : A \mid I(s) \wedge V(s) = x \}\!\}_A; c \sqsubseteq \mathsf{re}_A(F)\,\mathsf{er}}$$

This recursion theorem leads directly to the recursion introduction rule described below, and refinement monotonicity theorems for recursion blocks presented in Appendix C.

We can define while-do statements in terms of recursion in the usual way, leaving us with the following equality. When $c : \mathcal{P}_A$,

$$\mathsf{while}\ g\ \mathsf{do}\ c\ \mathsf{od} \quad = \quad \mathsf{re}_A\ N.\ \mathsf{if}\ g\ \mathsf{then}\ c; N\ \mathsf{else}\ \mathsf{Skip}_A\ \mathsf{fi}\,\mathsf{er}$$

We can prove that while-do loops are a 'lifted' form of Isabelle/ZF's standard least fixed point operator. For $g : \mathbb{P}\, A$ and $c : \mathcal{M}_A$, we have:

$$\mathsf{while}\ g\ \mathsf{do}\ c\ \mathsf{od} = \lambda\, q : \mathbb{P}\, A.\, \mathrm{lfp}_A\, N.\, (g \cap c`N) \cup ((A - g) \cap q))$$

As we would expect, a loop with a false guard won't start, and is equal to the skip statement. Also, a loop with a true guard won't terminate, and hence is equivalent to an aborting program. That is, for $a, b : \mathcal{M}_A$ and $\mathsf{strict}(b)$, we have:

$$\mathsf{while}\ \varnothing\ \mathsf{do}\ a\ \mathsf{od} = \mathsf{Skip}_A \qquad \mathsf{while}\ A\ \mathsf{do}\ b\ \mathsf{od} = \mathsf{Abort}_A$$

A general form of loop introduction can be expressed in terms of a Hoare correctness triple [47], and proved using well-founded induction. For $c : \mathcal{M}_A$, variant function V well typed given invariant I, i.e. $\forall\, s : A.\, I(s) \Rightarrow V(s) : T$, and relation $R : \mathbb{P}(T \times T)$ well-founded on T we have:

$$\frac{\forall\, x : X.\, \{\{s : A \mid\ I(s) \wedge G(s) \wedge V(s) = x\}\}\ c\ \{\{s : A \mid\ I(s) \wedge R(V(s), x)\}\}}{\{\{s : A \mid\ I(s)\}\}\ \mathsf{while}\ \{s : A \mid\ G(s)\}\ \mathsf{do}\ c\ \mathsf{od}\ \{\{s : A \mid\ I(s) \wedge \neg\, G(s)\}\}}$$

This theorem is used to prove all of the while-do loop introduction refinement law described in Chapter 4, and the refinement monotonicity theorem listed in Appendix C.

Recursion Interfaces

The raw recursion block captures the semantics of recursion, but it is not a suitable basis for program development. To apply the recursion theorem we would need to introduce the final code of the recursion in one fell swoop. The raw recursion block does allow us to refine a specification statement to a recursion block containing essentially the same specification statement. However, the refinement monotonicity theorem for raw recursion blocks does not let us know that suitable instances of the original specification statement can be refined to a recursive call.

However, we can use assertion statements to remember this information. The refinement theory and its assertion language are expressed in a uniform logic, and so mixing them presents us with no meta-logical difficulties. For example, we can prove the following general recursion introduction theorem. For $b : \mathcal{M}_A$, variant function V well-typed under invariant I, i.e. $\forall\, s : A.\, I(s) \Rightarrow V(s) : T$ over a relation R well-founded on T, we have:

$$\begin{array}{l} \{\!\{s : A \mid\ I(s)\}\!\}_A ; b \\ \sqsubseteq \\ \mathsf{re}_A N.\ \mathsf{con}_A\ x : T. \\ \quad \{\!\{\_ : A \mid\ \{\!\{s : A \mid\ R(V(s), x) \wedge I(s)\}\!\}_A ; b \ \sqsubseteq\ N \wedge V(s) = x\}\!\}_A ; b \\ \mathsf{er} \end{array}$$

That is, the assertion statement inside the logical constant statement 'remembers' that we can refine instances of our interface which reduce the variant function V into recursive calls N. The underscore in the context-dependent assertion statement indicates that it is asserting a proposition which is independent of the value of the state at that point.

This leads us to define the following interfaced recursion block, which encapsulates this contextual information.

$$\begin{array}{l}\mathsf{re}_A\ x\!:\!T, N \sqsupseteq I(x).\ c(x,N)\,\mathsf{er} \ \widehat{=} \\ \quad \mathsf{re}_A\ N.\ \mathsf{con}_A\ x\!:\!T.\ \{\!\{\_\!:\!A \mid\ I(x) \sqsubseteq N\}\!\}_A; c(x,N)\,\mathsf{er}\end{array}$$

The following refinement monotonicity theorem for this construct lets us do step-wise refinement on the body of the recursion block by providing us with extra context letting us refine the interface to a recursive call.

$$\frac{\forall\, x\!:\!T\ N\!:\!\mathcal{M}_A.\ I(x) \sqsubseteq N \Rightarrow a(x,N) \sqsubseteq b(x,N)}{\mathsf{re}_A\ x\!:\!T, N \sqsupseteq I(x).\ a(x,N)\,\mathsf{er} \ \sqsubseteq\ \mathsf{re}_A\ x\!:\!T, N \sqsupseteq I(x).\ b(x,N)\,\mathsf{er}}$$

Chapter 4

Lifting Sets to Predicates

We define a 'lifted' version of the language presented in the previous chapter. We lift sets representing expressions to functions in the meta-logic, and have statements pass abstracted state types to their subcomponents. These lifting techniques allow us to preserve the underlying expressiveness of untyped set theory, but provide us with many of the benefits of an implicitly typed logic. We also list a collection of refinement laws at the level of the lifted language.

The shallow embedding of the weakest precondition semantics by Agerholm [2] and the Refinement Calculator project [102] is done in higher-order logic. Our work is done in the untyped set theory of Isabelle/ZF for reasons that will be outlined in Chapter 5. Working in untyped set theory provides an extra overhead above working in an implicitly typed theory such as higher-order logic. In untyped set theory we must in effect provide explicit type information to defined operators. This chapter describes a technique to ameliorate this explicit typing burden by appealing to a higher-order meta-logic, such as that provided by Isabelle/ZF.

4.1 Predicate Transformers: Lift to Meta-Logic

We use three mechanisms to ameliorate the added burden of explicit typing. The first has been described already in Chapter 3: when defining statements, extract the state type from the structure of sub-components. The other two mechanisms are described below, and essentially depend on creating a new 'lifted' language. We will push as much as possible of the representation of set transformer statements into the meta-logical structure of the lifted language. Although ZF does not admit higher-order functions, we can define higher-order operators within Isabelle/ZF's meta-logic. This allows us to define

statements as meta-level type-passing operators, and also to use predicates on states to represent conditions.

4.1.1 Abstracting the State Type

For statements in the set transformer language, we had to explicitly denote the state type. This burden increases with the composition of statements: we must repeatedly mention the state type for every atomic sub-statement. We can reduce this burden by using a technique somewhat similar to the monadic style of functional programming. We define a language of lifted statements as higher-order operators in Isabelle's meta-logic. The state type is mentioned once at the top of a lifted statement, and is implicitly passed down to its sub-statements.

The definitions of lifted skip and sequential composition statements are as follows:

$$\begin{aligned} \mathsf{Skip} &\mathrel{\widehat{=}} \lambda A.\, \mathsf{Skip}_A \\ a; b &\mathrel{\widehat{=}} \lambda A.\, a(A); b(A) \end{aligned}$$

We engage in a slight abuse of notation here by overloading the names of conventional and lifted statements; context will normally distinguish them. The lambda in these definitions is Isabelle's meta-logical abstraction. The lifted skip statement takes a set which will form the state type of the conventional skip statement, and the sequential composition statement takes a set which it passes to its sub-statements. We have thus restricted our attention to homogeneous predicate transformers, but that suits our purposes of providing a simple 'user-level' theory of refinement.

We usually introduce the state type once at the top of an expression. Lifted equality and refinement is defined as follows:

$$\begin{aligned} a =_A b &\mathrel{\widehat{=}} a(A) = b(A) \\ a \sqsubseteq_A b &\mathrel{\widehat{=}} a(A) \sqsubseteq b(A) \end{aligned}$$

For example, compare $\mathsf{Skip} =_A \mathsf{Skip}; \mathsf{Skip}$ to its equivalent in set transformer statement: $\mathsf{Skip}_A = \mathsf{Skip}_A; \mathsf{Skip}_A$. This mechanism provides a great economy of notation in the presentation of complex programs, and recovers most of the benefits of implicit typing as seen in higher-order logic.

4.1.2 Lifting Sets to Functions and Predicates

Many statements defined in this chapter have operands which we intuitively think of as expressions or conditions, but which are represented as sets. For

example, we use sets of pairs of states to represent the state-valued expression in the state assignment statement, and we use sets of states to represent the Boolean expression in the guard of the alternation statement.

We can define 'lifted' versions of these statements which instead of representing these operands as sets, use meta-logical predicates or functions. To use predicates on states instead of sets of states, we can appeal to Isabelle/ZF's set comprehension $\{x : A \mid P(x)\}$ to construct the subset of A whose elements satisfy the characteristic function P. Similarly, to use meta-level functions instead of object-level functions we can appeal to Isabelle/ZF's bounded lambda quantification $\lambda\, x : A.\, F(x)$ to construct a set of pairs $\langle x, F(x)\rangle$ for x in A.

Thus our definitions of lifted state-assignment and alternation are as follows:

$$\begin{array}{rcl} \langle F \rangle & \widehat{=} & \lambda\, A.\, \langle \lambda\, x : A.\, F(x) \rangle_A \\ \mathsf{if}\ g\ \mathsf{then}\ a\ \mathsf{else}\ b\ \mathsf{fi} & \widehat{=} & \lambda\, A.\, \mathsf{if}\ \{x : A \mid g(x)\}\ \mathsf{then}\ a(A)\ \mathsf{else}\ b(A)\ \mathsf{fi} \end{array}$$

In these definitions we also abstract the state type, as was seen for the lifted skip and sequential composition statements.

4.1.3 The Lifted Language

The definitions of our lifted language follow the general scheme presented above, and appear in Appendix B. We do not lift fixed specification statements, as they are not useful for our refinement methodology—we want to derive programs that will effect *some* change in the program state! Given this, we write lifted free specification statements as $[P,\ Q]$.

4.2 Refinement Laws

We present a set of 'refinement laws' using this lifted language. Refinement laws are theorems which support a specific style of refinement methodology. Here we support a style of refinement similar to that described by Morgan [73]. To use this approach, we begin with a specification statement, which we transform into a compound statement whose sub-statements are specification statements. These are in turn progressively transformed until final code is reached. The refinement laws described here allow us to use the window inference style of reasoning described in Chapter 2.

4.2.1 Refinement as Transformation

In order to transform a statement using window inference, we need to know that refinement is a transitive relation. Refinement is a partial order on predicate transformers. For $a, b : \boldsymbol{\mathcal{M}}_A$ we have:

$$c \sqsubseteq_A c \qquad \frac{a \sqsubseteq_A b \quad b \sqsubseteq_A c}{a \sqsubseteq_A c} \qquad \frac{a \sqsubseteq_A b \quad b \sqsubseteq_A a}{a = b}$$

4.2.2 Refinement in Context

Given that we have refined a specification statement to a compound statement containing new specification statements, we want to be able to refine the new sub-statements in context. Theorems which show that compound statements are refinement-monotonic justify this. These refinement-monotonicity theorems correspond to window opening rules, and are shown in Appendix C.

4.2.3 Refinement of Specifications

A specification statement can be refined either into an atomic statement, or into a compound statement containing specification statements. These refinement laws correspond to window transformation rules, and are listed below.

Introduce Skip

We can transform a specification statement to a skip statement if the pre-condition implies the post-condition:

$$\frac{\forall\, s : A.\, P(s) \Rightarrow Q(s)}{[P,\ Q] \sqsubseteq_A \mathsf{Skip}}$$

Introduce Sequential Composition

We can transform a specification statement to the sequential composition of two specification statements which share an arbitrary mid-condition:

$$[P,\ R] \sqsubseteq_A [P,\ Q]; [Q,\ R]$$

Weaken Pre-Condition, Strengthen Post-Condition

We can transform a specification statement into another specification statement, by either weakening the pre-condition or strengthening the post-condition. We are left with the proof obligation that the new condition is implied by, or implies (respectively), the old condition:

$$\frac{\forall\, s : A.\ P(s) \Rightarrow P'(s)}{[P,\ Q] \sqsubseteq_A [P',\ Q]} \qquad \frac{(\exists\, s.\ P(s)) \Rightarrow (\forall\, s : A.\ Q'(s) \Rightarrow Q(s))}{[P,\ Q] \sqsubseteq_A [P,\ Q']}$$

Introduce State Assignment

We can transform a specification statement into a state assignment statement, if the the state function will give us a state satisfying the post-condition:

$$\frac{\forall\, s : A.\ P(s) \Rightarrow Q(F(s))}{[P,\ Q] \sqsubseteq_A \langle F \rangle}$$

Introduce Alternation

We can transform a specification statement to an alternation statement whose branches are specification statements. The guard is preserved in the pre-condition of the first branch, and the negation of the guard is preserved in the pre-condition of the second branch:

$$\begin{array}{lll} [P,\ Q] & \sqsubseteq_A & \mathsf{if}\ G\ \mathsf{then} \\ & & \quad [\lambda\, s.\ G(s) \wedge P(s),\ Q] \\ & & \mathsf{else} \\ & & \quad [\lambda\, s.\ \neg\ G(s) \wedge P(s),\ Q] \\ & & \mathsf{fi} \end{array}$$

Introduce While-Do Loop

We can transform a specification statement to a while-do loop whose body is a specification statement. The general rule for arbitrary well-founded relations R on X is as follows:

$$\begin{array}{c} \forall\, s : A.\ P(s) \Rightarrow I(s) \\ \forall\, s : A.\ I(s) \wedge \neg\ G(s) \Rightarrow Q(s) \\ \forall\, s : A.\ I(s) \Rightarrow V(s) : X \\ \hline [P,\ Q] \sqsubseteq_A \mathsf{while}\ G\ \mathsf{do} \\ \qquad \mathsf{con}\ x : X.\ [\lambda\, s.\ G(s) \wedge I(s) \wedge V(s) = x,\ \lambda\, s.\ I(s) \wedge R(V(s), x)] \\ \quad \mathsf{od} \end{array}$$

For the special case of the natural numbers, the rule is as follows:

$$\frac{\begin{array}{c}\forall\, s : A.\ P(s) \Rightarrow I(s) \\ \forall\, s : A.\ I(s) \wedge \neg\ G(s) \Rightarrow Q(s) \\ \forall\, s : A.\ I(s) \Rightarrow V(s) : \mathbb{N}\end{array}}{\begin{array}{l}[P,\ Q] \sqsubseteq_A \ \mathsf{while}\ G\ \mathsf{do} \\ \qquad \mathsf{con}\ x : \mathbb{N}.\ [\lambda\, s.\ G(s) \wedge I(s) \wedge V(s) = x,\ \lambda\, s.\ I(s) \wedge V(s) < x)] \\ \quad \mathsf{od}\end{array}}$$

Introduce Interfaced Recursion Block

We can transform a specification statement to a recursion statement whose body is a (essentially the same) specification statement:

The general rule for an arbitrary well-founded relation R on T, with a variant function well-typed (on $s : A$ satisfying $P(s)$) $V(s) : T$ is as follows:

$$[P,\ Q] \sqsubseteq_A \ \mathsf{re}\ x : T, N \sqsupseteq [\lambda\, s.\ R(V(s), x) \wedge P(s),\ Q].\ [\lambda\, s.\ V(s) = x \wedge P(s),\ Q]\ \mathsf{er}$$

The rule for a natural number-typed variant function (again, for any $s : A$ satisfying $P(s)$ we must have $V(s) : \mathbb{N}$) is as follows:

$$[P,\ Q] \sqsubseteq_A \ \mathsf{re}\ x : \mathbb{N}, N \sqsupseteq [\lambda\, s.\ V(s) < x \wedge P(s),\ Q].\ [\lambda\, s.\ V(s) = x \wedge P(s),\ Q]\ \mathsf{er}$$

Remove Assertion

At the end of a development we can remove any remaining assertion statements. We can refine assertion statements to skip as follows. As skip is a left and right identity of sequential composition, it can itself be reduced:

$$\{P\} \sqsubseteq_A \mathsf{Skip}$$

Remove Logical Constant

At the end of a development we can remove any logical constant statements which no longer bind variables in the program. For $c : \boldsymbol{\mathcal{M}}_A$ (c not dependent on v) we have:

$$\mathsf{con}\ v.\ c =_A c$$

Reduce Interfaced Recursion Block

When an interfaced recursion block does not contain any instances of the logical constant which fixes the variant function, then we can transform it to a standard recursion block. In effect, we refine away the assertion and the free logical constant statement. i.e. if $F(x) : \boldsymbol{\mathcal{M}}_A$ for any $x : \mathcal{M}_A$, then we have:

$$\mathsf{re}\ x : T, N \sqsupseteq c(x).\ F(N)\ \mathsf{er} \quad \sqsubseteq_A \quad \mathsf{re}\ N.\ F(N)\ \mathsf{er}$$

Chapter 5

Representing Program Variables

We extend our mechanisation of the refinement calculus to include statements which use program variables. The refinement calculus presented in Chapter 3 was defined in terms of a completely general state type. In this chapter we specialise our state type to allow us to represent program variables and their values. We can re-use the general framework established in Chapter 3 in this new setting.

Variable names are central to any programming language. Constructs common in programming languages include variable assignment for updating the value of a program variable, local blocks for declaring a new locally-scoped program variable of a particular type, and formal parameter declarations for procedures. Refinement languages often include other constructs, such as the framed specification statement [71], where the frame is a set of program variables which are allowed to change. It is more difficult to represent the semantics of refinement languages than programming languages. During program refinement, a developer may want to introduce a local variable whose values will come from some new mathematical set of values. In advance of performing a refinement we may not know which abstract types we will use, and so our type universe can't be fixed ahead of time.

5.1 State Representation

This section reviews progressively more sophisticated approaches for representing state types, and considers their suitability for representing state types in a mechanisation of the refinement calculus.

5.1.1 Review of State Representations

Variable Names to Numbers

The classic way to represent the states of toy imperative programming languages is by functions of the form $\mathcal{V} \to \mathbb{N}$ [36]. One can use either partial functions where the domain of the state is exactly the set of declared variables in scope, or total functions where we under-specify the values of variables which are out of scope.

The main problem with this approach is that even programming languages usually have a richer type universe than the natural numbers! In principle we could use Gödel numbering to represent any finitely specifiable value, but doing that would neither be perspicuous nor readily mechanisable. Moreover program refinement languages may use infinitely large abstract values, and so this approach is not suitable for our purposes.

Variable Names to Universal Type

It would be appealing to represent states as functions of the form $\mathcal{V} \to U$, where U is the universal type. However, most widely accepted logics do not admit such types!

Variable Names to Type Sum

To have a more expressive type universe, we might construct a range type consisting of the sum of types which we might use. For example, we might define a fixed type 'universe' τ for a state type $\mathcal{V} \to \tau$ as follows:

$$\tau ::= Nat\langle\langle \mathbb{N} \rangle\rangle \mid NatList\langle\langle \mathbb{N}\text{seq} \rangle\rangle \mid NatFun\langle\langle \mathbb{N} \to \mathbb{N} \rangle\rangle$$

This style of state representation, while fine for programming languages, is not entirely suitable for refinement languages, because we don't know ahead of time all of the types which we might use in a refinement.

Variable Names to a Progressively Defined Sum Type

The previous approach is unsuitable only because in the course of performing a refinement, we might choose to introduce a type which we have not already included in our type sum. One possible fix for this problem would be to redefine the type sum to include any new type, and then to either lift or replay the development with the new type sum. With a certain amount of 'hand-waving', this approach has some appeal in principle. However it would be cumbersome to deal with this hand-waving explicitly in a practical mechanisation.

Constructors of a Progressively Defined Data Type

A technique similar to the one above has been descried, based in the HOL theorem prover [18]. Here the state type is not represented by a map from variables to values, but rather as a datatype whose constructors represent program variables. This requires that the datatype be redefined upon entering every local block, regardless of whether the new local type has been seen before. Hence it is not a suitable basis for a practical refinement tool.

Polymorphism and Polymorphic Products

Although most type theories do not provide a universal type, many type theories do provide polymorphic types, which can be instantiated to any particular type. A polymorphic product represents the pairing of two values of an arbitrary type. Nesting these pairings allows us to represent finite vectors of arbitrarily typed values. So, one could represent a state type with three variables by a three-tuple with type $(\alpha \times (\beta \times \gamma))$. This approach to representing states was first mechanised by Agerholm [2] for use in program verification,

This mechanism allows us to represent state types with arbitrarily typed values. However, it does not allow us to represent variable names! More precisely, in this scheme there is no type of all variable names. 'Variables' here are projection functions from the current state type to an individual value. Just as there is no type of all types, so is there no type of all projection functions. Tuple positions, as bound variables, may have 'names' in the logic, but these do not distinguish between alpha-equivalent terms. For example, in the Refinement Calculator, the following two assignment statements are equal, even though the 'name' of the state variable varies:

$$(\mathsf{assign}(\lambda\, a.\ 1)) = (\mathsf{assign}(\lambda\, b.\ 1))$$

This mechanism does not allow us to represent normal properties of variables within our logic because it lacks variable names. For example, when a local block introduces a new variable, it defines the internal type by prepending the new local type to the external type sum. However, because program variables are identified with tuple positions rather than names, block statements don't hide variables declared in higher scopes. This fails to capture the normal programming language abstraction mechanism of variable name scoping. To represent single program variables, tricks can be played with let-expressions to bind specific projection functions to scoped let-bound variables. However, this still does not let us represent sets of variables, and so for example we can't give a uniform definition for framed specification

statements. Another limitation with this scheme is seen with parameterised procedure calls: for a given procedure, every differently typed calling context requires a different parameterisation mapping. This means we can't develop general procedure call rules, but must, as is seen in [101], re-prove them for every procedure and each of its calling state types.

Variable Names to Dependent Type

The final approach which we consider (and the one which we will adopt) is to represent state types as dependently typed partial or total functions of the form $\Pi_{v:V}.\,\tau(v)$. The family of types τ is a total meta-level function. Using dependent functions has the benefit of allowing us to represent variable names within the logic, and also allows us a free choice of types for program variables. This style of representation is available in expressive type theories, and in untyped logics such as Isabelle/ZF. Partial dependent functions will under-specify the typing τ for program variables not in the domain, and total dependent functions under-specify not only the typing τ but also the value of program variables not under consideration.

Kleymann [56] reports a similar representation in the context of machine-checked proofs of soundness and completeness results for program verification. Both Pratten [88] and von Wright [100] use a seemingly similar representation. They consider states a function from program variables x to a typing function D_x. However, they take D to be fixed. This is not appropriate when, for example, local blocks can change the type of a variable in scope. Our statements are defined over state types. This allows us to change the state type under consideration, for example when we enter a local block, or when we call a parameterised procedure.

5.1.2 Fixing Variables for a State Representation

In this thesis, we represent the state type as a total dependent function $\Pi_{v:V}.\,\tau(v)$ where τ is a (meta-level) variable-name indexed family of types.

5.2 Statements

This section presents the definitions of the set-transformer semantics for statements which use program variables. Again, we list theorems which provide a measure of validation to support our definitions. The definitions for the set-transformer semantics of statements using program variables appears in Appendix B. Theorems stating that these definitions are indeed (monotonic) set-transformers appear in Appendix C.

5.2.1 Atomic Statements

The atomic statements which we present are variable choice, single-variable assignment, multiple-variable assignment, and framed specification statements. They are related to the chaos, state assignment and specification statements seen in Chapter 3, but use program variables to give a finer grain of control over changes to the state.

Choose: Variable Chaos

The choose statement acts like chaos on a set of variables, leaving other variables unchanged. This can be seen by expressing the choose statement as relational non-deterministic assignment as follows[1]:

$$\mathsf{Choose}(w)_{\Pi_V T} = o :=_{\Pi_V T} i.\ o \text{ dsub } w = i \text{ dsub } w$$

For the special cases where the set of variables is empty or universal, the choose statement acts like skip or chaos respectively:

$$\mathsf{Choose}(\varnothing)_A = \mathsf{Skip}_A \qquad \mathsf{Choose}(V)_{\Pi_V T} = \mathsf{Chaos}_{\Pi_V T}$$

We won't use the choose statement within our user-level theory of refinement. However, it is an invaluable component in our construction of local blocks.

Single-Variable Assignment

Single-variable assignment updates the value of a single program variable to the value of an expression computed in the initial state. Our definition of single-variable assignment strongly resembles our definition of state assignment. Single-variable assignment can be expressed in terms of state assignment, i.e. (for $E : A \to X$) we have:

$$v :=_A E = \langle \lambda\, s : A.\ s[E`s/v] \rangle_A$$

That single-variable assignment updates only v can be seen by re-expressing it in terms of relational non-deterministic assignment. For $E : \Pi_V T \to T(v)$ and $v : V$, we have:

$$v :=_{\Pi_V T} E \ = \ o :=_{\Pi_V T} i.\ o`v = E`i \wedge o \text{ dsub } \{v\} = i \text{ dsub } \{v\}$$

[1] The operator dsub is relational domain subtraction, as indicated in Appendix A.

Multiple-Variable Assignment

Multiple-variable assignment is like single-variable assignment, but can assign values to many variables in parallel. A multiple-variable assignment $\langle \overrightarrow{M} \rangle_{A,B}$ contains an expression M which takes a state and returns a new set of variable/value pairs which will override variables in the initial state. It resembles state assignment, and specialises to it when it assigns to every variable. When $M : \Pi_V X \to \Pi_V Y$, we have the following:

$$\langle \overrightarrow{M} \rangle_{\Pi_V Y, \Pi_V X} = \langle M \rangle_{\Pi_V Y}$$

When a multiple-variable assignment affects only a singleton variable, it is equivalent to the single-variable assignment statement. When E represents an expression from a state type B, we have:

$$\langle \overrightarrow{\overline{\lambda\, s : B.\, \{\langle v, E\text{`}s \rangle\}}} \rangle_{A,B} = v :=_A E$$

Framed Specifications

In Chapter 3, we defined fixed and free specification statements, which (respectively) did not change the state, and could change the state in any manner whatsoever. Where states are functions from variables to values, this means fixed specification statements may not change the value of any program variable, and free specification statements may change the value of any program variable. We can now define a framed specification statement $w : [P,\ \ Q]_A$ whose frame w contains a set of program variables which are allowed to be modified. When the frame is either empty or universal, a framed specification statement is equivalent to (respectively) the fixed or free specification statement. For example, we have (in the latter case when $Q : \mathbb{P}\, \Pi_V \tau$):

$$\varnothing : [P,\ \ Q]_A = {}^{\perp}[P,\ Q]_A \qquad V : [P,\ \ Q]_{\Pi_V \tau} = {}^{\top}[P,\ Q]_{\Pi_V \tau}$$

We can use a logical constant to 'hold steady' the value of a program variable in the frame of a framed specification statement. Alternatively, instead of holding steady one program variable, we can fix the entire state, thus expressing framed specification in terms of free specification. For a v in w a subset of V, we have:

$$\begin{aligned} w - \{v\}\ &: [P,\ \ Q]_{\Pi_V \tau} = \mathsf{con}\ e.\ w : [\{s : P \mid e = s\text{`}v\},\ \{s : Q \mid e = s\text{`}v\}]_{\Pi_V \tau} \\ w\ &: [P,\ \ Q]_{\Pi_V \tau} = \mathsf{con}\ x.\ {}^{\top}[\{s : \Pi_V \tau \mid s \in P \wedge x = s \text{ dsub } w\}, \\ &\qquad\qquad \{s : \Pi_V \tau \mid s \in Q \wedge x = s \text{ dsub } w\}]_{\Pi_V \tau} \end{aligned}$$

Just as we could express chaos in terms of free specification, so can we express the choose statement in terms of framed specification. Similarly, we

can express framed specification in terms of assertion and guarding interposed by the choose statement. Formally, we have:

$$\begin{array}{rcl} \mathsf{Choose}(w)_A & = & w : [A, \quad A]_A \\ w : [P, \quad Q]_A & = & \{P\}_A ; \mathsf{Choose}(w)_A ; (Q)_A \rightarrow \end{array}$$

5.2.2 Compound Statements: Localisation

In this section we describe compound statements that make use of program variables. We will first consider single and multiple variable declaration blocks. We also describe parameterisation, which substitutes for program variables used as formal parameters. However, we postpone our description of parameter declaration until Chapter 6. These statements are defined in a similar way:

1. store the initial state,

2. initialise the new values for the localised variable(s),

3. execute the localised sub-statement, and

4. finally restore the original values of the localised variable(s).

Our variable localisation statements are homogeneous set transformers which contain a differently-typed homogeneous set transformer. Our initialisation and finalisation stages are therefore heterogeneous set transformers. Our development of variable localisation is, to some extent, based on Pratten's unmechanised formalisation of local variables [88].

Single-Variable Blocks

A single-variable block $\mathsf{begin}_A\ v.\ c\ \mathsf{end}$ executes the statement c in an initial state which has an unknown value for the program variable v. We model this initialisation by demonically setting the initial value of v using $\mathsf{Choose}(\{v\})_{B,A}$. The finalisation simply restores the original value of v with a state assignment statement. We have defined blocks such that when $c : \mathcal{P}_B$, we have the following equality:

$$\mathsf{begin}_A\ v.\ c\ \mathsf{end} \quad = \quad \mathsf{store}_A\ i.\ \mathsf{Choose}(\{v\})_{B,A} ; c ; \langle \lambda\, x : B.\, x[{}^{i`v}/_{v}] \rangle_A$$

Multiple-Variable Blocks

Multiple-variable blocks are a straight-forward generalisation of single-variable blocks above where we declare a set of program variables instead of a single program variable. When $c : \mathcal{P}_B$, we have the following equality:

$$\mathsf{begin}_A\ \overrightarrow{w}.\ c\ \mathsf{end} \quad = \quad \mathsf{store}_A\ i.\ \mathsf{Choose}(w)_{B,A}; c; \langle \lambda\, x : B.\, x \oplus (i \operatorname{dres} w) \rangle_A$$

Trivially, multiple-variable blocks which declare a single variable are the same as single-variable blocks:

$$\mathsf{begin}_A\ \overrightarrow{\{v\}}.\ c\ \mathsf{end} \quad = \quad \mathsf{begin}_A\ v.\ c\ \mathsf{end}$$

Variable Parameterisation

It is standard in theories of the refinement calculus to separate the treatment of parameterisation and procedure call [70]. We consider procedure calls and formal parameter declaration in Chapter 6 but here describe the semantics underlying parameterisation. Presentations of the refinement calculus typically consider three different kinds of parameter declaration: value, result, and value-result [73, 88]. Procedures with multiple parameters must be modelled with a composition of these individual rules. However, this is a semantically unacceptable treatment: the evaluation of each parameter argument should be done in the same calling state, rather than being progressively updated by each parameter evaluation.

We define a generic parameterisation statement $\mathsf{Param}_A(c, I, F)$ which represents a parameterised call to c from a calling type A. Like a multiple-variable assignment statement, parameterisation updates the value of many variables in parallel. The variables to be updated are contained in the structure of the initialisation and finalisation arguments I and F. These components represent the interpretation of the actual parameters in the context of some formal parameter declaration. The following equality holds. When $c : \mathcal{P}_B$,

$$\begin{array}{l} \mathsf{Param}_A(c, I, F) = \\ \quad \mathsf{store}_A\ i.\ \mathsf{Chaos}_{B,A}; \langle \lambda\, s : B.\, s \oplus I(i) \rangle_B; c; \langle \lambda\, s : B.\, i \oplus F(s) \rangle_A \end{array}$$

Parameterisation resembles the multiple-variable block statement, except for two important differences. First, parameterisation uses the chaos statement to help establish the initial state. This prevents us from using global variables within procedure bodies. This restriction was introduced to allow us to represent procedure calls from various state types. The localised statement

c is defined only at the declaration state type B, but may be called from any external state type A. Additionally, references to variables outside the parameter list would be dynamically bound to variable names in scope at run-time, rather than statically bound to variables in scope at declaration time. This would be inconsistent with the semantics of most imperative programming languages. Second, at finalisation we don't simply restore values of localised variables. We restore the entire initial state except for variables carried in the finalisation expression F. Thus we will be able to change the value of variables given as actual parameters to result or value-result parameter declarations.

Chapter 6

Lifting with Program Variables

In the previous chapter we fixed the state type in the set transformer semantics so that we could represent program variables. We can now redefine our lifted language, in order to encapsulate more of this state structure. We describe our representation of parameter declaration, and interfaced procedures and recursive procedures. Finally, we list refinement laws for the statements introduced in this chapter.

6.1 Using State Structure in Lifting

In Chapter 4, we developed a lifted language of type-passing predicate transformers in order to reduce the burden of explicitly annotating set transformers with their state-types. In the previous chapter we specialised the state-type to be a dependent function type with a fixed domain. We can revisit our approach to abstracting the state-type to take advantage of this extra structure. Instead of using an arbitrary set of variable names V, we fix them to a constant non-empty set of variable names $\mathcal{V}$. For a typing τ, the state type $\mathcal{S}_\tau$ is defined as follows:

$$\mathcal{S}_\tau \rightleftharpoons \Pi_\mathcal{V} \tau$$

Our type of variable names $\mathcal{V}$ is a constant, and so instead of passing sets $\Pi_\mathcal{V}\tau$ representing the state-type, we will pass only the typings τ. Although τ is not a set, defining a lifted language over a family of types presents no problem for Isabelle/ZF's higher-order meta-logic. In a manner similar to our previous lifted language, we introduce typings once at the top of an expression, and implicitly pass them to any component sub-statements. For

example, we can define lifted equality and refinement as follows:

$$\begin{array}{rcl} a =_\tau b & \widehat{=} & a(\tau) = b(\tau) \\ a \sqsubseteq_\tau b & \widehat{=} & a(\tau) \sqsubseteq b(\tau) \end{array}$$

We can define the lifted skip, sequential composition, state assignment and alternation statements as follows. Note that we can reconstruct the state type $\mathcal{S}_\tau$ from the typing τ:

$$\begin{array}{rcl} \mathsf{Skip} & \widehat{=} & \lambda\, \tau.\, \mathsf{Skip}_{\mathcal{S}_\tau} \\ a; b & \widehat{=} & \lambda\, \tau.\, a(\tau); b(\tau) \\ \langle F \rangle & \widehat{=} & \lambda\, \tau.\, \langle \lambda\, S : \mathcal{S}_\tau .\, F(s) \rangle_{\mathcal{S}_\tau} \\ \mathsf{if}\ g\ \mathsf{then}\ a\ \mathsf{else}\ b\ \mathsf{fi} & \widehat{=} & \lambda\, \tau.\, \mathsf{if}\ \{s : \mathcal{S}_\tau \mid g(s)\}\ \mathsf{then}\ a(\tau)\ \mathsf{else}\ b(\tau)\ \mathsf{fi} \end{array}$$

Other lifted statements are defined similarly. The definitions of all these lifted statements appear in Appendix B. We will take as read the replay, in this modified setting, of work presented in Chapter 4.

6.2 Lifting Blocks

There are no expressions or conditions in our block statements, so lifting them will mostly be a matter of abstracting the typing. However, the state type inside a block will in general be different to the state type outside the block. Hence, we cannot simply pass the external typing to the body of the block. For each local variable we will require a type declaration. We will use these type declarations to modify the typing which we implicitly pass to the body of the block.

6.2.1 Lifting Single-Variable Blocks

Given an external typing τ and a single-variable block which declares a local variable v of type T, we will pass a modified typing $\tau[T/v]$ to the body of the block, as follows:

$$\mathsf{begin}\ v : T.\ c\ \mathsf{end} \ \widehat{=}\ \lambda\, \tau.\, \mathsf{begin}_{\mathcal{S}_\tau}\ v.\ c(\tau[T/v])\ \mathsf{end}$$

Typing substitution is defined as follows:

$$\tau[T/v] \ \widehat{=}\ \lambda\, w.\, \mathsf{if}(w = v, T, \tau(w))$$

As we would expect, single-variable blocks are monotonic predicate transformers:

$$\frac{c : \boldsymbol{\mathcal{M}}_{\tau[T/v]}}{\mathsf{begin}\ v : T.\ c\ \mathsf{end} : \boldsymbol{\mathcal{M}}_{\tau}}$$

The effect of the modified typing environment can be seen in the following refinement monotonicity theorem:

$$\frac{a : \boldsymbol{\mathcal{M}}_{\tau[X/v]} \quad b : \boldsymbol{\mathcal{M}}_{\tau[X/v]} \quad a \sqsubseteq_{\tau[X/v]} b}{\mathsf{begin}\ v : X.\ a\ \mathsf{end} \sqsubseteq_{\tau} \mathsf{begin}\ v : X.\ b\ \mathsf{end}}$$

Whenever we refine a block, we use these theorems to calculate the internal typing for the block.

6.2.2 Lifting Multiple-Variable Blocks

For multiple-variable blocks, we can declare a set of variables and their new types. We will model these declarations as a function from the local variables to their types. Hence the domain of this set will be the set of names of the local variables. Our definition of lifted multiple-variable blocks is as follows:

$$\mathsf{begin}\ D.\ c\ \mathsf{end} \quad \widehat{=} \quad \lambda\, \tau.\, \mathsf{begin}_{\mathcal{S}_\tau}\ \overrightarrow{\mathrm{dom}(D)}.\ c(\tau \boxplus D)\ \mathsf{end}$$

Our syntax here for lifted multiple-variable blocks is for expository purposes overloaded with the syntax for lifted single-variable blocks.

The typing overriding operator $\boxplus$ is defined as follows:

$$\tau \boxplus S \quad \widehat{=} \quad \lambda\, w.\, \mathsf{if}(w \in \mathrm{dom}(S), S`w, \tau(w))$$

Although we define multiple-variable blocks in terms of this typing overriding, we will always resolve concrete instances to typing substitutions, using the theorems below. (For the second theorem we need to know that $\{\langle v, T\rangle\} \cup S$ is a function, i.e. that the variable names are distinct.)

$$\tau \boxplus \varnothing = \tau \qquad \tau \boxplus (\{\langle v, T\rangle\} \cup S) = (\tau \boxplus S)[T/v]$$

As we would expect, multiple-variable blocks are monotonic predicate transformers:

$$\frac{c : \boldsymbol{\mathcal{M}}_{\tau \boxplus D}}{\mathsf{begin}\ D.\ c\ \mathsf{end} : \boldsymbol{\mathcal{M}}_{\tau}}$$

The effect of the modified typing environment can be seen in the following refinement monotonicity theorems:

$$\frac{a : \boldsymbol{\mathcal{M}}_{\tau \boxplus D} \quad b : \boldsymbol{\mathcal{M}}_{\tau \boxplus D} \quad a \sqsubseteq_{\tau \boxplus D} b}{\mathsf{begin}\ D.\ a\ \mathsf{end} \sqsubseteq_{\tau} \mathsf{begin}\ D.\ b\ \mathsf{end}}$$

Whenever we refine a block, we use these theorems to calculate the internal typing for the block.

6.3 Parameterised Procedures

Our representation of parameterised procedures will bring together many of the strands of work in this thesis. As is common in the refinement literature [70], we develop procedures separately from parameterisation. Parameterised procedures are lifted set transformers on a state type representing program variables. Like the recursion blocks discussed in Chapter 4, we will associate interfaces with our procedures, so that we can support a step-wise refinement methodology. Also, like the variable blocks discussed earlier in this chapter, the body of a procedure has a different state type to its calling type. A procedure body is defined at only one state type, but the procedure can be called from many different state types. We will use parameter declaration information to create a declaration typing to pass to calls of a parameterised statement. This declaration typing is fixed per procedure, and ignores the calling type. This restriction means that we do not model global variables. An extension to the language presented in this thesis could support global variables by introducing a mechanism for explicitly listing global variables used in a procedure.

6.3.1 Lifting Parameterisation

Lifting the parameterisation statement is slightly different to lifting other statements. Because a procedure body is defined at its declaration type and not its calling type, we must pass the declaration type to a parameterised statement. So, we define the lifted parameterisation statement as follows:

$$\mathsf{PARAM}(c, S, I, F) \;\widehat{=}\; \lambda\, \tau.\, \mathsf{Param}_{\mathcal{S}_\tau}(c(S), I, F)$$

6.3.2 Parameter Declarations

The $\mathsf{PARAM}(c, S, I, F)$ statement is a general form of parameterisation. It uses initialisation and finalisation expressions embodying parameter declarations which have already been resolved. We turn now to consider how these expressions are constructed.

Formal and Actual Argument Syntax

We consider three types of parameterisation common in the refinement literature: value, result, and value-result [70]. We define a collection of uninterpreted constants to use as syntactic abbreviations for declaring each kind of formal parameter declaration: $\mathsf{value}\ v : T$, $\mathsf{result}\ v : T$ and $\mathsf{valueresult}\ v : T$.

The formal arguments of a procedure are represented by a list of these formal parameters.

An actual parameter is represented by the operator $\mathsf{Arg}\ a$, where a is a meta-level function representing an expression. For value parameters, this will be a normal expression, and for result and value-result parameters, this will be a constant expression returning a program variable. The actual arguments to a procedure are represented by a list of these actual parameters.

Interpreting Parameter Declarations

We define three operators for interpreting lists of parameter declarations D given a typing at declaration τ and actual argument list a: a declaration typing operator $\mathbb{T}_{D,\tau}$, and initialisation and finalisation operators $\mathbb{I}_{D,a}$ and $\mathbb{F}_{D,a}$. Parameterised statements c are then interpreted as $\mathsf{PARAM}(c, \mathbb{T}_{D,\tau}, \mathbb{I}_{D,a}, \mathbb{F}_{D,a})$.

$\mathbb{T}_{D,\tau}$ determines the typing internal to the declaration. Where *decl* is any of our three forms of formal parameter declaration, we have:

$$\begin{aligned} \mathbb{T}_{[],\tau} &\mathrel{\widehat{=}} \tau \\ \mathbb{T}_{(decl\ \ v:T)::l,\tau} &\mathrel{\widehat{=}} \mathbb{T}_{l,\tau[T/v]} \end{aligned}$$

$\mathbb{I}_{D,a}$ determines the initialisation expression. It takes the state s outside the procedure call, so that actual arguments can be evaluated. Value parameters evaluate an expression which is substituted for the formal parameter, and value-result parameters take the value of variable given as the actual argument and substitute it for the formal parameter:

$$\begin{aligned} \mathbb{I}_{[],[]} &\mathrel{\widehat{=}} \lambda\, s.\, \varnothing \\ \mathbb{I}_{(\mathsf{value}\ v:T::)dl,(\mathsf{Arg}\ a)::al} &\mathrel{\widehat{=}} \lambda\, s.\, \{\langle v, a(s)\rangle\} \cup \mathbb{I}_{dl,al}(s) \\ \mathbb{I}_{(\mathsf{result}\ v:T::)dl,(\mathsf{Arg}\ a)::al} &\mathrel{\widehat{=}} \mathbb{I}_{dl,al} \\ \mathbb{I}_{(\mathsf{valueresult}\ v:T::)dl,(\mathsf{Arg}\ a)::al} &\mathrel{\widehat{=}} \lambda\, s.\, \{\langle v, s`a(s)\rangle\} \cup \mathbb{I}_{dl,al}(s) \end{aligned}$$

$\mathbb{F}_{D,a}$ determines the finalisation expression. It takes a state s outside the procedure call, so that actual variable arguments can be updated. Result and value-result parameters update the value of the variable given as the actual argument with the value of the formal parameter:

$$\begin{aligned} \mathbb{F}_{[],[]} &\mathrel{\widehat{=}} \lambda\, s.\, \varnothing \\ \mathbb{F}_{(\mathsf{value}\ v:Ty::)dl,(\mathsf{Arg}\ a)::al} &\mathrel{\widehat{=}} \mathbb{F}_{dl,al} \\ \mathbb{F}_{(\mathsf{result}\ v:Ty::)dl,(\mathsf{Arg}\ a)::al} &\mathrel{\widehat{=}} \lambda\, s.\, \{\langle a(s), s`v\rangle\} \cup \mathbb{F}_{dl,al}(s) \\ \mathbb{F}_{(\mathsf{valueresult}\ v:Ty::)dl,(\mathsf{Arg}\ a)::al} &\mathrel{\widehat{=}} \lambda\, s.\, \{\langle a(s), s`v\rangle\} \cup \mathbb{F}_{dl,al}(s) \end{aligned}$$

For example, assume we have a simple procedure declared (at a typing τ) as follows:

$$\mathsf{procedure}\ P([\mathsf{value}\ a : A,\ \mathsf{result}\ b : B,\ \mathsf{valueresult}\ c : C]) \mathrel{\widehat{=}} Body$$

Here a, b and c are variables in $\mathcal{V}$. Now, consider the following procedure call:

$$P([\mathsf{Arg}\ \ \lambda\, s.\, \alpha(s),\ \mathsf{Arg}\ \ \lambda\, s.\, \beta,\ \mathsf{Arg}\ \ \lambda\, s.\, \gamma])$$

Here α is an expression of type A, and β, and γ are program variables in $\mathcal{V}$, with $s`\beta : B$ and $s`\gamma : C$. This procedure call can be expanded to the following parameterised statement:

$$\mathsf{PARAM}(Body, \tau[C/c][B/b][A/a], \lambda\, s.\, \langle a, \alpha(s)\rangle, \langle c, s`\gamma\rangle, \lambda\, s.\, \langle \beta, s`b\rangle, \langle \gamma, s`c\rangle)$$

Note that the value parameters a and c are updated in the initialisation expression, and the result arguments β and γ are updated in the finalisation expression. In an actual program refinement this expanded form of procedure call need not be seen by the developer, as it is built into the definition of procedure declaration.

6.3.3 Interfaced Procedures

A non-recursive procedure P with an implementation B which is declared for a calling context $C(P)$ is like the let binding $\mathbf{let}\, P = B\, \mathbf{in}\, C(P)$. In our refinement methodology, we want to introduce procedures by refining some statement C to $\mathbf{let}\, P = I\, \mathbf{in}\, C$. Then when refining C, we can refine instances of the procedure interface I to calls to P. However, at some time, we might refine I to B. When later refining C, we would still like to be able to refine instances of the interface I into calls to P. However, a simple let-binding is not sufficient to support this methodology. This is similar to the limitations of the raw recursion block discussed in Chapter 3. We solve this problem in a similar way: we add an interface I to the syntax of our procedures and remember (with an assertion statement) that B is an implementation of I. Thus, our procedures are more like: $\mathbf{let}\, P = B\, \mathbf{in}\, \{\, \lambda\_.\, I \sqsubseteq B\}; C(P)$. This would allow us, when refining C, to replace instances of I with calls to P. The underscore $\_$ above highlights the fact that the asserted context is not dependent on the value of the state. This allows us to extract this condition as a purely logical fact in the context of $C(P)$.

Our procedures are actually slightly more complicated, because they are parameterised, and because we must implicitly pass the declaration typing

$\mathbb{T}_{D,\tau}$ to $C(P)$, regardless of whatever the state type may be at any call to P in $C(P)$. Our definition of interfaced parameterised procedures is as follows:

$$\begin{array}{l} \quad \mathsf{proc}\ P(D)\,\mathsf{int} = I\,\mathsf{imp} = B\,\mathsf{in}\ C(P) \\ \widehat{=} \\ \quad \mathbf{let}\ P = \lambda\,\tau\ a.\,\mathsf{PARAM}(B, \mathbb{T}_{D,\tau}, \mathbb{I}_{D,a}, \mathbb{F}_{D,a}) \\ \quad \mathbf{in}\ \lambda\,\tau.\,(\{\,\lambda\_.\, I\ \sqsubseteq_{\mathbb{T}_{D,\tau}}\ B\,\};\, C(P(\tau)))(\tau) \end{array}$$

As we would expect, interfaced procedures are monotonic predicate transformers, given that their implementations are predicate transformers at the declaration type, and given that the calling context is a predicate transformer of the external type:

$$\frac{B : \boldsymbol{\mathcal{M}}_{\mathbb{T}_{D,\tau}} \quad (\bigwedge P.\,(\bigwedge S\ a.\,P(a) : \boldsymbol{\mathcal{M}}_S) \Longrightarrow C(P) : \boldsymbol{\mathcal{M}}_\tau)}{\mathsf{proc}\ P(D)\,\mathsf{int} = I\,\mathsf{imp} = B\,\mathsf{in}\ C(P) : \boldsymbol{\mathcal{M}}_\tau}$$

In the hypothesis of the rule above, note that when proving that the calling context is a predicate transformer, we can assume that all of the procedure calls are monotonic predicate transformers for any type S and any argument a. We will often see the assumption that a procedure call P is a monotonic predicate transformer, regardless of its arguments or calling type. We will abbreviate this as follows:

$$P : \boldsymbol{\mathcal{M}} \rightleftharpoons \bigwedge S\ a.\,P(a) : \boldsymbol{\mathcal{M}}_S$$

We can introduce a procedure at any point using the following theorem:

$$\frac{C : \boldsymbol{\mathcal{M}}_\tau}{C\ \sqsubseteq_\tau\ \mathsf{proc}\ P(D)\,\mathsf{int} = I\,\mathsf{imp} = I\,\mathsf{in}\ C}$$

We can refine the various components of a procedure. The following theorems are refinement monotonicity theorems used to transform the implementation, interface, and calling contexts of an interfaced procedure.

To transform an implementation B to a new implementation B', we can use the following theorem. In addition to the main hypothesis listed below, we will require that the sub-components are monotonic predicate transformers, i.e. $I, B, B' : \boldsymbol{\mathcal{M}}_{\mathbb{T}_{D,\tau}}$ and $(\bigwedge P.\,P : \boldsymbol{\mathcal{M}} \Longrightarrow C(P) : \boldsymbol{\mathcal{M}}_\tau)$, and also that the calling context is refinement-monotonic, i.e.:
$\bigwedge P\ Q.\,[\![P, Q : \boldsymbol{\mathcal{M}};\ (\bigwedge S\ a.\,P(a)\ \sqsubseteq_S\ Q(a))]\!] \Longrightarrow C(P)\ \sqsubseteq_\tau\ C(Q)$.

These extra obligations would usually be automatically proved in a mechanised interactive environment. This leaves us with the main implementation refinement-monotonicity theorem:

$$\frac{B \sqsubseteq_{\mathbb{T}_{D,\tau}} B'}{\mathsf{proc}\ P(D)\,\mathsf{int} = I\,\mathsf{imp} = B\,\mathsf{in}\ C(P) \sqsubseteq_\tau \mathsf{proc}\ P(D)\,\mathsf{int} = I\,\mathsf{imp} = B'\,\mathsf{in}\ C(P)}$$

To transform an interface I to a new interface I', we can use the following theorem. Again, we require that sub-components are monotonic predicate transformers, i.e. $I, I', B : \boldsymbol{\mathcal{M}}_{\mathbb{T}_{D,\tau}}$ and also $(\bigwedge P.\, P : \boldsymbol{\mathcal{M}} \implies C(P) : \boldsymbol{\mathcal{M}}_\tau)$. These obligations would usually be automatically proved in a mechanised interactive environment. This leaves us with the main interface refinement–anti-monotonicity theorem:

$$\frac{I' \sqsubseteq_{\mathbb{T}_{D,\tau}} I}{\mathsf{proc}\ P(D)\,\mathsf{int} = I\,\mathsf{imp} = B\,\mathsf{in}\ C(P) \sqsubseteq_\tau \mathsf{proc}\ P(D)\,\mathsf{int} = I'\,\mathsf{imp} = B\,\mathsf{in}\ C(P)}$$

To transform a calling context C to a new one C', we can use the following theorem. We require that the sub-components are monotonic, i.e. $B, I : \boldsymbol{\mathcal{M}}_{\mathbb{T}_{D,\tau}}$ and $(\bigwedge P.\, P : \boldsymbol{\mathcal{M}} \implies C(P), C'(P) : \boldsymbol{\mathcal{M}}_\tau)$. In a mechanised interactive proof, these will usually be automatically proved, leaving us with the main context refinement-monotonicity theorem below:

$$\frac{\left(\begin{array}{l}\bigwedge P.\, [\![P : \boldsymbol{\mathcal{M}};\\ \quad (\bigwedge S\ a.\, \mathsf{PARAM}(I, \mathbb{T}_{D,\tau}, \mathbb{I}_{D,a}, \mathbb{F}_{D,a}) \sqsubseteq_S P(a))]\!] \implies\\ \quad\quad C(P) \sqsubseteq_\tau C'(P)\end{array}\right)}{\mathsf{proc}\ P(D)\,\mathsf{int} = I\,\mathsf{imp} = B\,\mathsf{in}\ C(P) \sqsubseteq_\tau \mathsf{proc}\ P(D)\,\mathsf{int} = I\,\mathsf{imp} = B\,\mathsf{in}\ C'(P)}$$

The main hypothesis here is that the old calling context refines to the new calling context. When performing this refinement, we are allowed to assume that calls to P are monotonic predicate transformers, and that a (parameterised) instance of the interface I can be refined to a call to P.

6.3.4 Interfaced Recursive Procedures

Just as a non-recursive procedure is like the let binding of a simple statement, so a recursive procedure is like the let binding of a recursion statement $\mathbf{let}\, P = \mathsf{re}\ P.\ B(P)\,\mathsf{er}\,\mathbf{in}\, C(P)$. Here instances of let-bound P in $C(P)$ represent initial calls to the recursive procedure, and re-bound instances of P in $B(P)$ represent recursive calls. In a manner similar to non-recursive procedures above and to recursion blocks in Chapter 3, we can remember the interface I to a recursive procedure with an assertion statement. We will also use a bound logical constant $v : V$ to compare to our variant function (and its well-founded relation) in R, in order to help us demonstrate that recursion terminates. So, our recursive procedures are more like the following:

$\mathbf{let}\, P = B'\,\mathbf{in}\, \{\, \lambda\_.\, I \sqsubseteq B' \}; C(P)$
$\mathbf{where}\, B' = \mathsf{re}\ P.\ \mathsf{con}\ v : V.\ \{\, \lambda\_.\, \{\, \lambda\, s.\, R(v, s) \}; I \sqsubseteq P \}; B(v, P)\,\mathsf{er}$

Again, the recursive procedures which we consider in this thesis are more complicated than this, because our procedures are parameterised, and because we must pass the declaration typing to any instance of the procedure call. We must treat both the initial call and the recursive call in a similar way. Our definition of interfaced recursive parameterised procedures is as follows:

$$
\begin{array}{l}
\quad \mathsf{rec}\ \ P(D)\ \ \mathsf{int} = I\ \mathsf{var} = v : V, R(v)\, \mathsf{imp} = B(v,P)\, \mathsf{in}\ C(P) \\
\widehat{=} \\
\quad \mathbf{let}\, P = \lambda\, \tau\ a.\, \mathsf{PARAM}(B'(\tau), \mathbb{T}_{D,\tau}, \mathbb{I}_{D,a}, \mathbb{F}_{D,a}) \\
\quad \mathbf{in}\, \lambda\, \tau.\, (\{\, \lambda\_.\, I\ \sqsubseteq_{\mathbb{T}_{D,\tau}}\ B'(\tau)\}; C(P(\tau)))(\tau) \\
\quad \mathbf{where}\, B'(\tau) = \\
\qquad \mathsf{re}\ P.\ \mathsf{con}\ v : V. \\
\qquad\quad \{\, \lambda\_.\, \{\, \lambda\, s.\, R(v,s)\}; I\ \sqsubseteq_{\mathbb{T}_{D,\tau}}\ P\}; \\
\qquad\quad B(v, \lambda\, a.\, \mathsf{PARAM}(P, \mathbb{T}_{D,\tau}, \mathbb{I}_{D,a}, \mathbb{F}_{D,a})) \\
\qquad \mathsf{er}
\end{array}
$$

As we would expect, a recursive procedure is a monotonic predicate transformer given that its calling context is a monotonic predicate transformer:

$$
\frac{\bigwedge P.\, P : \boldsymbol{\mathcal{M}} \Longrightarrow C(P) : \boldsymbol{\mathcal{M}}_\tau}{\mathsf{rec}\ \ P(D)\ \ \mathsf{int} = I\ \mathsf{var} = v : V, R(v)\, \mathsf{imp} = B(v,P)\, \mathsf{in}\ C(P) : \boldsymbol{\mathcal{M}}_\tau}
$$

We can introduce a recursive procedure at any point using the following theorem. The side-conditions are all typing obligations. We only require our well-founded variant function to be well-typed when the interface holds:

$$
\frac{\begin{array}{c} I : \boldsymbol{\mathcal{M}}_{\mathbb{T}_{D,\tau}} \qquad C : \boldsymbol{\mathcal{M}}_\tau \qquad R : \mathbb{P}(X \times X) \qquad \mathrm{wf}_X\, R \\ \forall\, x : \mathcal{S}_{\mathbb{T}_{D,\tau}}.\, (\exists\, q : \mathbb{P}\, \mathcal{S}_{\mathbb{T}_{D,\tau}}.\, x : I(\mathbb{T}_{D,\tau})\text{`}q) \Rightarrow V(x) : X \end{array}}{\begin{array}{l} C\ \sqsubseteq_\tau\ \mathsf{rec}\ P(D)\, \mathsf{int} = I\, \mathsf{var} = x : X, (\lambda\, s.\, R(V(s), x)) \\ \qquad\qquad\qquad\qquad \mathsf{imp} = \{\, \lambda\, s.\, V(s) = x\}; I \\ \qquad\quad \mathsf{in}\ C \end{array}}
$$

When the interface to a recursive procedure is a specification statement, we can prove simplified versions of the theorem above which merge the assertion statement into the precondition of the initial implementation. These simplified laws are shown later in this chapter along with the other main refinement laws.

We can transform the interface I of a recursive procedure into a new interface I' by using the following theorem. We will require that our subcomponents are monotonic predicate transformers, i.e. $I, I' : \boldsymbol{\mathcal{M}}_{\mathbb{T}_{D,\tau}}$, $(\bigwedge P\ v.\, [\![v : V;\ P : \boldsymbol{\mathcal{M}}]\!] \Longrightarrow B(v,P) : \boldsymbol{\mathcal{M}}_{\mathbb{T}_{D,\tau}})$, and

$(\bigwedge P.\, P : \boldsymbol{\mathcal{M}} \implies C(P) : \boldsymbol{\mathcal{M}}_\tau)$, and that our calling context is refinement-monotonic, i.e.
$(\bigwedge P\ Q.[\![P, Q : \boldsymbol{\mathcal{M}};\ (\bigwedge S\ a.\, P(a) \sqsubseteq_S Q(a))]\!] \implies C(P) \sqsubseteq_\tau C(Q))$.

In a mechanised interactive proof, all of these will usually be automatically proved, leaving us with the main interface refinement–anti-monotonicity theorem below:

$$\frac{I' \sqsubseteq_{\mathbb{T}_{D,\tau}} I}{\begin{array}{r}\mathsf{rec}\ P(D)\ \ \mathsf{int} = I\ \mathsf{var} = v : V, R(v)\ \mathsf{imp} = B(v,P)\ \mathsf{in}\ C(P) \\ \sqsubseteq_\tau \mathsf{rec}\ P(D)\ \ \mathsf{int} = I'\ \mathsf{var} = v : V, R(v)\ \mathsf{imp} = B(v,P)\ \mathsf{in}\ C(P)\end{array}}$$

We can transform the calling context $C(P)$ of a recursive procedure to a new calling context $C'(P)$ by using the theorem below. Again, we will require that the sub-components are monotonic predicate transformers, i.e. $I : \boldsymbol{\mathcal{M}}_{\mathbb{T}_{D,\tau}}$, and $(\bigwedge P.\, P : \boldsymbol{\mathcal{M}} \implies C(P), C'(P) : \boldsymbol{\mathcal{M}}_\tau)$. These can all usually be automatically proven, leaving us with the main calling context refinement monotonicity theorem below:

$$\frac{\left(\begin{array}{l}\bigwedge P.[\![P : \boldsymbol{\mathcal{M}}; \\ \quad (\bigwedge S\ a.\, \mathsf{PARAM}(I, \mathbb{T}_{D,\tau}, \mathbb{I}_{D,a}, \mathbb{F}_{D,a}) \sqsubseteq_S P(a))]\!] \implies \\ \quad\quad C(P) \sqsubseteq_\tau C'(P)\end{array}\right)}{\begin{array}{r}\mathsf{rec}\ P(D)\ \ \mathsf{int} = I\ \mathsf{var} = v : V, R(v)\ \mathsf{imp} = B(v,P)\ \mathsf{in}\ C(P) \\ \sqsubseteq_\tau \mathsf{rec}\ P(D)\ \ \mathsf{int} = I\ \mathsf{var} = v : V, R(v)\ \mathsf{imp} = B(v,P)\ \mathsf{in}\ C'(P)\end{array}}$$

We can transform the recursive call $B(v,P)$ of a recursive procedure to a new recursive call $B'(v,P)$ by using the theorem below. We will require that the sub-components are monotonic predicate transformers, i.e. $I : \boldsymbol{\mathcal{M}}_{\mathbb{T}_{D,\tau}}$, $(\bigwedge P.\, P : \boldsymbol{\mathcal{M}} \implies C(P) : \boldsymbol{\mathcal{M}}_\tau)$, and
$(\bigwedge P\ v.[\![v : V;\ P : \boldsymbol{\mathcal{M}}]\!] \implies B(v,P), B'(v,P) : \boldsymbol{\mathcal{M}}_{\mathbb{T}_{D,\tau}})$ and that the calling context is refinement-monotonic, i.e.
$(\bigwedge P\ Q.[\![P, Q : \boldsymbol{\mathcal{M}};\ (\bigwedge S\ a.\, P(a) \sqsubseteq_S Q(a))]\!] \implies C(P) \sqsubseteq_\tau C(Q))$.

These conditions can usually be automatically proved, leaving us with the main recursive-call refinement-monotonicity theorem:

$$\frac{\left(\begin{array}{l}\bigwedge P\ v.[\![v : V;\ P : \boldsymbol{\mathcal{M}}; \\ \quad (\bigwedge S\ a.\, \mathsf{PARAM}(\{\lambda s.\, R(v,s)\}; I, \mathbb{T}_{D,\tau}, \mathbb{I}_{D,a}, \mathbb{F}_{D,a}) \sqsubseteq_S P(a)) \\ \quad]\!] \implies B(v,P) \sqsubseteq_\tau B'(v,P)\end{array}\right)}{\begin{array}{r}\mathsf{rec}\ P(D)\ \ \mathsf{int} = I\ \mathsf{var} = v : V, R(v)\ \mathsf{imp} = B(v,P)\ \mathsf{in}\ C(P) \\ \sqsubseteq_\tau \mathsf{rec}\ P(D)\ \ \mathsf{int} = I\ \mathsf{var} = v : V, R(v)\ \mathsf{imp} = B'(v,P)\ \mathsf{in}\ C(P)\end{array}}$$

Where the interface I is a specification statement, this theorem can be specialised by merging the parameterised assertion statement with the pre-condition of the specification statement. We do not show these specialised theorems here.

6.4 Refinement Laws

The refinement laws and refinement-monotonicity theorems for the state-type independent statements given in Chapter 4 carry over under the modified lifting given in this chapter. We will take that as read, and here discuss refinement laws for statements making use of program variables. We have already seen the refinement-monotonicity theorems for local blocks and our procedure statements, and so we will focus on refinement transformation laws. We will first present the refinement laws for transforming free specification statements into statements involving program variables, and then we will present the refinement laws for transforming framed specification statements.

6.4.1 Refining Free Specifications

Introduce Framed Specification Statement

A form of frame contraction is to transform a free specification statement to a framed specification statement with the same pre and post conditions:

$$[P,\ Q] \quad \sqsubseteq_\tau \quad w : [P,\quad Q]$$

Introduce Single-Variable Assignment

We can transform a free specification statement to a single-variable assignment statement where the assignment updates a program variable $v : \mathcal{V}$. We can assume that the precondition is true when proving that the assignment establishes the postcondition and is well-typed:

$$\frac{\forall\, s : \mathcal{S}_\tau .\, P(s) \Rightarrow Q(s[E(s)/v]) \wedge E(s) : \tau(v)}{[P,\ Q] \sqsubseteq_\tau\ v := E}$$

Introduce Multiple-Variable Assignment

We can transform a free specification statement to a multiple-variable assignment statement where the assignment expression F updates a set of program variables v. We can assume that the precondition holds when proving that the assignment establishes the postcondition and is well-typed:

$$\frac{(\forall\, s : \mathcal{S}_\tau .\, P(s) \Rightarrow Q(s \oplus F(s)) \wedge F(s) : \Pi_v \tau)}{[P,\ Q] \sqsubseteq_\tau\ \langle \overrightarrow{F} \rangle}$$

Introduce Single-Variable Block

We can transform a free specification statement to a local block which contains essentially the same specification statement, but with the value of the localised program variable existentially bound in the pre-condition and universally bound in the post-condition. That is, the internal specification statement must work for any initial value of the local variable and is allowed to produce any final value for the local variable. For v in $\mathcal{V}$ and non-empty type T and state-space $\mathcal{S}_\tau$, we have:

$$[P,\ Q] \quad \sqsubseteq_\tau \quad \mathsf{begin}\ v : T. \\ [\lambda\, s.\, \exists\, e : \tau(v).\, P(s[e/v]),\ \lambda\, s.\, \forall\, e : \tau(v).\, Q(s[e/v])] \\ \mathsf{end}$$

This rule reduces to Morgan's block introduction rule [73] when P and Q don't depend on v. For example, if $P(s)$ contains $s`u$, then $P(s[e/v])$ will instead contain $s[e/v]`u$. This will simplify to $s`u$ for $u \neq v$. Thus, if P or Q don't depend on v, e will not be bound, and so the quantifiers can be simplified away, leaving Morgan's rule.

Introduce Multiple-Variable Block

Given that our new declaration D is a function from some set of variables V, we have the following multiple-variable block introduction rule:

$$[P,\ Q] \sqsubseteq_\tau \mathsf{begin}\ D.\ [\lambda\, s.\, \exists f : \Pi_V \tau.\, P(s \oplus f),\ \lambda\, s.\, \forall f : \Pi_V \tau.\, Q(s \oplus f)]\ \mathsf{end}$$

Again, when P and Q don't depend on V (the variables in the domain of D), then the quantifiers can be simplified away.

Introduce Procedure

At any time we can introduce a non-recursive procedure whose implementation is the same as its interface, by using the general rule given earlier in Section 6.3.3.

Introduce Recursive Procedure

We can introduce a recursive procedure whose interface is a free specification statement, and whose implementation is the same specification statement, given that the calling context is well-typed, i.e. $C : \boldsymbol{\mathcal{M}}_\tau$, and that the variant function V is well-typed on the well-founded R, i.e.
$\forall\, x : \mathcal{S}_{\mathbb{T}_{D,\tau}}.\, p(x) \Rightarrow V(x) : X.$

$$C \sqsubseteq_\tau \mathsf{rec}\ P(D)\ \mathsf{int} = [p,\ q]\ \mathsf{var} = x : X, (\lambda\, s.\, R(V(s), x))\ \mathsf{imp} = [\lambda\, s.\, x = V(s) \wedge p(s),\ q]\ \mathsf{in}\, C$$

Introduce Procedure Call

The refinement-monotonicity theorems for procedures and recursive procedures allow us to transform parameterised instances of the procedure interface into calls to the procedure. This rule determines if a free specification statement is a suitable instance of an interface which is itself a free specification statement. We can assume that the calling precondition P is true when proving that the parameterisation initialisation I is well-typed and satisfies the interface precondition p. We can also assume that the calling precondition P is true when proving that the finalisation F maps states satisfying the interface postcondition q to states satisfying the calling postcondition Q:

$$\frac{\forall\, i : \mathcal{S}_\tau\ x : \mathcal{S}_S.\ P(i) \Rightarrow x \oplus I(i) : \{s : \mathcal{S}_S \mid p(s)\} \quad \forall\, i : \mathcal{S}_\tau\ x : \mathcal{S}_S.\ P(i) \wedge q(x) \Rightarrow i \oplus F(x) : \{s : \mathcal{S}_\tau \mid Q(s)\}}{[P,\ Q] \sqsubseteq_\tau \mathsf{PARAM}([p,\ q], S, I, F)}$$

6.4.2 Refining Framed Specifications

Contract Frame

We can transform a framed specification statement with a frame w of program variables to the same specification statement, except with a smaller frame w':

$$w : [P,\quad Q] \quad \sqsubseteq_\tau \quad w' : [P,\quad Q]$$

Expand Frame

We can expand the frame of a framed specification statement if we nonetheless introduce a logical variable to fix the value of the program variable. For $v : \mathcal{V}$ we have:

$$w : [P,\quad Q] \quad \sqsubseteq_\tau \quad \mathsf{con}\ v_0.\ v, w : [\lambda\, s.\, P(s) \wedge s`v = v_0,\quad \lambda\, s.\, Q(s) \wedge s`v = v_0]$$

Weaken Precondition, Strengthen Postcondition

We can transform the precondition and postcondition of a framed specification statement by appealing to the following theorems. When proving that a new postcondition will establish the old postcondition, we can assume that

the precondition originally held, and that variables not in the frame are the same in the initial and final states:

$$\frac{\forall\, s : \mathcal{S}_\tau .\, P(s) \Rightarrow P'(s)}{w : [P, \quad Q] \sqsubseteq_\tau \; w : [P', \quad Q]}$$

$$\frac{\forall\, i : \mathcal{S}_\tau .\, P(i) \Rightarrow (\forall\, o : \mathcal{S}_\tau .\, i \,\mathrm{dsub}\, w = o \,\mathrm{dsub}\, w \wedge Q'(o) \Rightarrow Q(o))}{w : [P, \quad Q] \sqsubseteq_\tau \; w : [P, \quad Q']}$$

Introduce Skip

We can refine a framed specification statement to a skip statement when a state satisfying the precondition satisfies the postcondition:

$$\frac{\forall\, s : \mathcal{S}_T .\, P(s) \Rightarrow Q(s)}{w : [P, \quad Q] \sqsubseteq_\tau \; \mathsf{Skip}}$$

Introduce Sequential Composition

We can introduce an arbitrary mid-condition when we introduce the sequential composition of framed specification statements:

$$w : [P, \quad R] \sqsubseteq_\tau \; w : [P, \quad Q]; w : [Q, \quad R]$$

Introduce Alternation

We can transform a framed specification statement to an alternation statement with an arbitrary guarding expression. The guard's context is carried into the precondition of framed specification statements in the branches of the alternation statement:

$$\begin{array}{l} w : [P, \quad Q] \sqsubseteq_\tau \; \mathsf{if}\; G\; \mathsf{then} \\ \qquad\qquad w : [\lambda\, s .\, G(s) \wedge P(s), \quad Q] \\ \qquad \mathsf{else} \\ \qquad\qquad w : [\lambda\, s .\, \neg\, G(s) \wedge P(s), \quad Q] \\ \qquad \mathsf{fi} \end{array}$$

Introduce While-Do Loop

We can transform a framed specification statement into a while-do loop. The body of the loop will be a framed specification statement embodying the invariant of the loop and information about the variant showing termination of the loop. The general rule for variant functions decreasing on a well-founded relation R on X is as follows:

$$\frac{\begin{array}{c}\forall\, s : \mathcal{S}_\tau .\, I(s) \Rightarrow V(s) : X \\ \forall\, s : \mathcal{S}_\tau .\, P(s) \Rightarrow I(s) \\ \forall\, s : \mathcal{S}_\tau .\, I(s) \wedge \neg\, G(s) \Rightarrow Q(s)\end{array}}{\begin{array}{l} w : [P,\quad Q] \sqsubseteq_\tau \ \mathsf{while}\ G\ \mathsf{do} \\ \qquad \mathsf{con}\ x : X .\ w : [\lambda\, s .\, G(s) \wedge I(s) \wedge V(s) = x, \\ \qquad\qquad\qquad \lambda\, s .\, I(s) \wedge R(V(s), x)] \\ \quad \mathsf{od}\end{array}}$$

When the relation is less-than on the natural numbers, we have the following special case of the above theorem:

$$\frac{\begin{array}{c}\forall\, s : \mathcal{S}_\tau .\, I(s) \Rightarrow V(s) : \mathbb{N} \\ \forall\, s : \mathcal{S}_\tau .\, P(s) \Rightarrow I(s) \\ \forall\, s : \mathcal{S}_\tau .\, I(s) \wedge \neg\, G(s) \Rightarrow Q(s)\end{array}}{\begin{array}{l} w : [P,\quad Q] \sqsubseteq_\tau \ \mathsf{while}\ G\ \mathsf{do} \\ \qquad \mathsf{con}\ x : \mathbb{N} .\ w : [\lambda\, s .\, G(s) \wedge I(s) \wedge V(s) = x, \\ \qquad\qquad\qquad \lambda\, s .\, I(s) \wedge V(s) < x] \\ \quad \mathsf{od}\end{array}}$$

Introduce Single-Variable Assignment

We can transform a framed specification statement into a single-variable assignment statement, if the variable is in the frame and the assignment will give us a state satisfying the post-condition. For $v \in w$, we have:

$$\frac{\forall\, s : \mathcal{S}_\tau .\, P(s) \Rightarrow Q(s[E(s)/v])}{w : [P,\quad Q] \sqsubseteq_\tau \ v := E}$$

Introduce Multiple-Variable Assignment

We can transform a framed specification statement into a multiple-variable assignment statement. The assignment expression F must affect only a set of variables v contained in the frame w of program variables for the specification statement:

$$\frac{(\forall\, s : \mathcal{S}_\tau .\, P(s) \Rightarrow F(s) : \Pi_v \tau \wedge Q(s \oplus F(s))}{w : [P,\quad Q] \sqsubseteq_\tau \ \langle \overrightarrow{F} \rangle}$$

Introduce Single-Variable Block

We can transform a framed specification statement to a local block which contains essentially the same specification statement, but with the frame extended with the localised variable, and with the value of the localised

program variable existentially bound in the pre-condition and universally bound in the post-condition. That is, the internal specification statement must work for any initial value of the local variable and is allowed to produce any final value for the local variable. For a set of variables $w \subseteq \mathcal{V}$ and a local variable $v : \mathcal{V}$, with a non-empty external state type $\mathcal{S}_\tau$ and local variable type T, we have:

$$
\begin{array}{l}
\quad w : [P, \quad Q] \\
\sqsubseteq_\tau \\
\quad \mathsf{begin}\ v : T. \\
\qquad v, w : [\lambda\, s.\, \exists\, e : \tau(v).\, P(s[e/v]), \quad \lambda\, s.\, \forall\, e : \tau(v).\, Q(s[e/v])] \\
\quad \mathsf{end}
\end{array}
$$

Introduce Multiple-Variable Block

Given that our the new declarations is a function from some set of variable names v, and that the frame of the specification statement is a set of variable names w, we have:

$$
\begin{array}{l}
\quad w : [P, \quad Q] \\
\sqsubseteq_\tau \\
\quad \mathsf{begin}\ D.\ w \cup v : [\lambda\, s.\, \exists f : \Pi_v \tau.\, P(s \oplus f), \quad \lambda\, s.\, \forall f : \Pi_v \tau.\, Q(s \oplus f)]\ \mathsf{end}
\end{array}
$$

Introduce Procedure

At any time we can introduce a non-recursive procedure whose implementation is the same as its interface, by using the general rule given earlier in Section 6.3.3.

Introduce Recursive Procedure

We can introduce a recursive procedure whose implementation is its interface, given that the calling context is well-typed, i.e. $C : \boldsymbol{\mathcal{M}}_\tau$, and that the variant function V is well-typed on the well-founded R, i.e. $\forall\, x : \mathcal{S}_{\mathbb{T}_{D,\tau}}.\, p(x) \Rightarrow V(x) : X$.

$$
\begin{array}{lll}
C & \sqsubseteq_\tau & \mathsf{rec}\ P(D)\ \mathsf{int} = w : [p, \quad q]\ \mathsf{var} = x : X, (\lambda\, s.\, R(V(s), x)) \\
& & \qquad\qquad\ \ \mathsf{imp} = w : [\lambda\, s.\, x = V(s) \wedge p(s), \quad q] \\
& & \quad \mathsf{in}\, C
\end{array}
$$

Introduce Procedure Call

The refinement-monotonicity theorems for procedures and recursive procedures allow us to transform parameterised instances of the procedure interface into calls to the procedure. This rule determines if a framed specification

statement is a suitable instance of an interface which is itself a framed specification statement. We can assume that the calling precondition P is true when proving that the parameterisation initialisation I is well-typed and satisfies the interface precondition p. When proving that the finalisation F maps states satisfying the interface postcondition q to states satisfying the calling postcondition Q, and that the calling frame W is not violated, we can assume that the calling precondition P is true, and that the procedure interface respects its frame w:

$$\frac{\begin{array}{c}\forall\, i : \mathcal{S}_\tau\ x : \mathcal{S}_S.\ P(i) \Rightarrow x \oplus I(i) : \{s : \mathcal{S}_S \mid p(s)\} \\ \left(\begin{array}{c}\forall\, i : \mathcal{S}_\tau\ x : \mathcal{S}_S.\ P(i) \wedge q(x) \wedge (\exists\, y : \mathcal{S}_S.\ (y \oplus I(i)) \text{ dsub } w = x \text{ dsub } w) \\ \Rightarrow i \text{ dsub } W = (i \oplus F(x)) \text{ dsub } W \wedge i \oplus F(x) : \{s : \mathcal{S}_\tau \mid Q(s)\}\end{array}\right)\end{array}}{W : [P,\ \ Q] \sqsubseteq_\tau \mathsf{PARAM}(w : [p,\ \ q], S, I, F)}$$

6.5 A Small Example

In order to give some idea about how the refinement rules appear when applied in practice, we present a small example: the development of a non-recursive procedure which computes natural number multiplication by repeated addition. We will present the refinement as it would appear in a window inference proof, and we will note which side-conditions can be proved automatically.

We begin with our initial specification, a framed specification statement which requires program variable r be equal to the product of the values of the program variables a and b. Only r is allowed to change in the course of the program:

$$r : [\text{true},\ \ r = a * b]$$

In the course of this refinement, we assume a top-level typing environment τ, with $\tau(a) = \tau(b) = \tau(r) = \mathbb{N}$. We begin by introducing an as yet undeveloped procedure to perform the multiplication. The user must supply the formal parameters and the interface of the procedure in an application of the *Introduce Procedure Call* rule from Section 6.3.3. This leaves us with the following:

$$\begin{array}{ll}\sqsubseteq_\tau & \mathit{Mult}([\mathsf{value}\ x : \mathbb{N}, \mathsf{value}\ y : \mathbb{N}, \mathsf{result}\ z : \mathbb{N}]) \\ & \mathsf{int}\ {=}z : [\text{true},\ \ z = x * y] \\ & \mathsf{imp}\ {=}z : [\text{true},\ \ z = x * y] \\ & \mathsf{in}\ r : [\text{true},\ \ r = a * b]\end{array}$$

The use of this rule in combination with the transitivity of refinement results in side-conditions that the framed specification statement and the procedure declaration are lifted monotonic predicate transformers. These are automatically proven and not seen by the user. In this section we will not again comment on these kind of side-conditions, as they are always automatically proved and not seen by the user.

We can proceed by developing the calling part of the procedure. We open a window on the in part of the procedure declaration, using the refinement-monotonicity theorem from Section 6.3.3. This leaves us with the following focus:

$$r : [\text{true}, \quad r = a * b]$$

Using the refinement monotonicity rule also gives us two extra pieces of context in the hypotheses of this focus:

$$\bigwedge S \ a. \mathsf{PARAM}(z : [\text{true}, \quad z = x * y], \tau', \mathbb{I}_{D,a}, \mathbb{F}_{D,a}) \sqsubseteq_S \ \mathit{Mult}(a)$$
$$\bigwedge S \ a. \mathit{Mult}(a) : \boldsymbol{\mathcal{M}}_S$$

Here τ' is the declaration typing, i.e. $\tau[\mathbb{N}/x][\mathbb{N}/y][\mathbb{N}/z]$, and D are the formal parameter declarations, i.e. $[\mathsf{value}\ x : \mathbb{N}, \mathsf{value}\ y : \mathbb{N}, \mathsf{result}\ z : \mathbb{N}]$. These hypothesis will be carried forward in this sub-development, and allow us to transform any suitably parameterised instance of the procedure interface into an appropriate call to the *Mult* procedure. We can introduce the procedure at any calling type S and with any suitable actual arguments a, and know that the procedure call is a lifted monotonic predicate transformer.

We will use this context immediately, in conjunction with the *Introduce Procedure Call* rule of Section 6.3.3 to transform our framed specification statement into a call to *Mult*, as follows:

$$\sqsubseteq_\tau \quad \mathit{Mult}([a, b, r])$$

As our specification statement was a trivial instance of the procedure interface, the two main proof obligations of the *Introduce Procedure Call* rule are automatically proved. That is, the actual pre- and post-conditions are parameterised instances of the interface's pre- and post-conditions. However, a typing side-condition is presented to the user as a result of this rule:

$$a : \mathbb{N} \wedge b : \mathbb{N} \quad \Rightarrow \quad a * b : \mathbb{N}$$

We can supply a hint to quickly prove this condition. When we close the window onto the procedure call, we are presented with the top-level procedure declaration with the new calling part. We now open on the implementation of the procedure, by using the appropriate refinement-monotonicity theorem from Section 6.3.3. We are left with the following focus, at the typing τ':

$$z : [\text{true}, \quad z = x * y]$$

The user can supply a new local variable t and type $\mathbb{N}$, and by appealing to the *Introduce Multiple-Variable Block* rule from Section 6.4, transform the specification statement to the following:

$$\begin{array}{ll} \sqsubseteq_{\tau'} & \textsf{begin } t : \mathbb{N}. \\ & \quad t, z : [\exists\, x.x : \Pi_{\{t\}}\tau'', \quad (\exists\, x.x : \Pi_{\{t\}}\tau') \Rightarrow z = x * y] \\ & \textsf{end} \end{array}$$

Here $\tau'' = \tau'[\mathbb{N}/t]$. The side-condition that the block declaration is functional (i.e. that we do not declare two local variables with the same name) is automatically proved and not seen by the user. We can open on body of block, by using the refinement-monotonicity rule of Section 6.2. This leaves us with:

$$t, z : [\exists\, x.x : \Pi_{\{t\}}\tau'', \quad (\exists\, x.x : \Pi_{\{t\}}\tau') \Rightarrow z = x * y]$$

We can ignore the existential assumption in the pre-condition, but we will clean up the post-condition by using the *Strengthen Postcondition* rule of Section 6.4, to arrive at:

$$\sqsubseteq_{\tau''} \quad t, z : [\exists\, x.x : \Pi_{\{t\}}\tau'', \quad z = x * y]$$

The side conditions from the application of this rule are trivial, are automatically proved, and are not seen by the user. We now nominate the mid-condition $z = 0 \wedge t = x$, and in conjunction with the *Introduce Sequential Composition* rule of Section 6.4, introduce a sequential composition statement, as follows:

$$\begin{array}{ll} \sqsubseteq_{\tau''} & t, z : [\exists\, x.x : \Pi_{\{t\}}\tau'', \quad z = 0 \wedge t = x]; \\ & t, z : [z = 0 \wedge t = x, \quad z = x * y] \end{array}$$

There are no visible side-conditions, and we can open on the first statement by appealing to the left-argument refinement-monotonicity of sequential composition rule as listed in Appendix C. This leaves us with the following focus, which we can transform to a multiple assignment statement by using the *Introduce Multiple-Variable Assignment* rule from Section 6.4:

$$\begin{array}{ll} & t, z : [\exists\, x.x : \Pi_{\{t\}}\tau'', \quad z = 0 \wedge t = x] \\ \sqsubseteq_{\tau''} & t, z := x, 0 \end{array}$$

The side-conditions from this transformation can mostly be proved automatically. We are presented with a typing side-condition which we can prove by supplying the hint that $0 : \mathbb{N}$. We can close this window, and open on the second statement. We will transform this into a while-do loop by using the *Introduce While-Do Loop* rule of Section 6.4 and nominating a guard $t \neq 0$, an invariant $z + (t * y) = x * y$, and a variant function t.

$$
\begin{array}{ll}
 & t, z : [z = 0 \wedge t = x, \quad z = x * y] \\
\sqsubseteq_{\tau''} & \textsf{while } t \neq 0 \textsf{ do} \\
 & \quad \textsf{con } V{:}\mathbb{N}.\ t, z : [t \neq 0 \wedge z + (t * y) = x * y \wedge t = V, \\
 & \qquad\qquad z + (t * y) = x * y \wedge t < V] \\
 & \textsf{od}
\end{array}
$$

We are presented with two arithmetic side-conditions, that the invariant is initially true, and that at the end of the loop the post-condition is satisfied. That is, for $\{t, x, y, z\} \in \mathbb{N}$, we must show: $0 + (x * y) = x * y$ and $t = 0 \wedge z + (t * y) = x * y \Rightarrow z = x * y$. These can both be proved by the user.

We open on body of loop and the bound logical constant statement, and transform the specification statement to a multiple-variable assignment as follows:

$$
\begin{array}{ll}
 & t, z : [t \neq 0 \wedge z + (t * y) = x * y \wedge t = V, \\
 & \qquad z + (t * y) = x * y \wedge t < V] \\
\sqsubseteq_{\tau''} & t, z := t - 1, z + y
\end{array}
$$

The user must supply hints to satisfy the following typing side-conditions: $t : \mathbb{N} \Rightarrow t - 1 : \mathbb{N}$, and $z : \mathbb{N} \wedge y : \mathbb{N} \Rightarrow z + y : \mathbb{N}$. This will leave us with two main side-conditions, that the loop body maintains the invariant, and that it decreases the variant function. That is, for $\{t, x, y, z\} \in \mathbb{N}$, we must show:

$$
\begin{array}{rcl}
t \neq 0 \wedge z + (t * y) = x * y & \Rightarrow & z + y + ((t - 1) * y) = x * y \\
t \neq 0 & \Rightarrow & t - 1 < t
\end{array}
$$

These facts of arithmetic can be proved by the user.

We have now completed our development. We can close back to the top level, stopping along the way to remove the logical constant which no longer appears in the body of the loop. We use the *Remove Logical Constant* rule from Section 4.2 as follows:

$$
\begin{array}{ll}
 & \textsf{con } V{:}\mathbb{N}.\ t, z := t - 1, z + y \\
\sqsubseteq_{\tau''} & t, z := t - 1, z + y
\end{array}
$$

We are left with our final theorem, whose conclusion is as follows:

$$
\begin{array}{ll}
 & r : [\text{true}, \quad r = a * b] \\
\sqsubseteq_{\tau} & \mathit{Mult}([\textsf{value } x : \mathbb{N}, \textsf{value } y : \mathbb{N}, \textsf{result } z : \mathbb{N}]) \\
 & \textsf{int} = z : [\text{true}, \quad z = x * y] \\
 & \textsf{imp} = \\
 & \quad \textsf{begin } t : \mathbb{N}. \\
 & \qquad t, z := x, 0; \\
 & \qquad \textsf{while } t \neq 0 \textsf{ do } t, z := t - 1, z + y \textsf{ od} \\
 & \quad \textsf{end} \\
 & \textsf{in} \\
 & \mathit{Mult}([a, b, r])
\end{array}
$$

Chapter 7

Data Refinement

We define a data-refinement relation, which allows us to change the data-type invariant and representation of program variables. We first define data-refinement at the set-transformer level, and then lift it in a manner analogous to the lifting of procedural refinement. Finally, we list a collection of data-refinement rules, discuss the data-refinement of procedure interfaces, and comment on the use of contextual information in practical data-refinement.

Data-refinement concerns the correctness-preserving re-implementation of programs. An initial procedural program refinement will typically use program variables on abstract types in order to highlight the simple core structure of an algorithm. We can use data-refinement to change the invariant on our abstract types, or to transform abstract types into more concrete types. Alternative data-representations can provide us with opportunities for more efficient implementations. Moreover, sufficiently concrete implementations will be realisable in a programming language with a real-world compiler.

Formal techniques for changing data representations in programs were first proposed by Hoare [48]. Rules for data-refinement have been widely published [78, 76, 14, 11, 34, 100]. In this chapter, we describe a mechanisation of forward data-refinement similar to the formalisation by von Wright [100]. Our work differs in that our data-refinement operator does not have operands for adding and removing program variables. Lacking this frame information, we cannot prove a rule which will allow us to add or remove program variables while performing a data-refinement. However, we can change the representation and type of the values of program variables. Also, we don't pose universal restrictions on data-refinement relations, but instead consider restrictions as they arise in the proof of individual data-refinement rules.

Below, we provide a list of piece-wise data-refinement laws for the statements considered in this thesis. Of special interest will be the rules for blocks and procedures. Our rules for data-refining blocks let us data-refine the body of a block under a different data-refinement relation to the relation external to the block. In particular, procedural refinement is a special case of data-refinement, and so we present rules which let us introduce a data-refinement by procedurally refining local blocks. Similarly, special cases of our parameterised procedure data-refinement rules allow us to refine a procedure by data-refinement. Our procedure data-refinement rules are also notable in letting us data-refine procedure calls under whatever data-refinement relation is in force at the point of the procedure call.

7.1 Data Refinement

Our definition of forward data-refinement uses relational assertion for abstraction, and (converse) relational non-deterministic assignment for representation. They form an adjoint pair, as follows. For $R : \mathbb{P}(K \times A)$,

$$\begin{array}{l} \{R\}_{A,K}; [R^{-1}]_{K,A} \sqsubseteq \mathsf{Skip}_K \\ \mathsf{Skip}_A \sqsubseteq [R^{-1}]_{K,A}; \{R\}_{A,K} \end{array}$$

Our definition of the data-refinement of set transformer statements is as follows:

$$a \sqsubseteq (R)\ k \ \widehat{=}\ \{R\}_{\bigcup(\mathrm{dom}(a)),\bigcup(\mathrm{dom}(k))}; a \sqsubseteq k; \{R\}_{\bigcup(\mathrm{dom}(a)),\bigcup(\mathrm{dom}(k))}$$

We extract the abstract and concrete state types from the operands, just as we did in Chapter 3. Hence, for $a : \mathcal{P}_A$ and $k : \mathcal{P}_K$, we have the simpler equivalence:

$$a \sqsubseteq (R)\ k \ \equiv\ \{R\}_{A,K}; a \sqsubseteq k; \{R\}_{A,K}$$

There are three other well-known equivalent definitions [100]. For $a : \mathcal{M}_A$, $k : \mathcal{M}_K$ and $R : \mathbb{P}(K \times A)$, we have the following equivalences:

$$\begin{array}{lcl} a \sqsubseteq (R)\ k & \equiv & a \sqsubseteq [R^{-1}]_{K,A}; k; \{R\}_{A,K} \\ a \sqsubseteq (R)\ k & \equiv & \{R\}_{A,K}; a; [R^{-1}]_{K,A} \sqsubseteq k \\ a \sqsubseteq (R)\ k & \equiv & a; [R^{-1}]_{K,A} \sqsubseteq [R^{-1}]_{K,A}; k \end{array}$$

Procedural refinement is a special case of data-refinement over homogeneous state types with the identity relation. Given $a, b : \mathcal{P}_A$, we have:

$$a \sqsubseteq b \equiv a \sqsubseteq (\lambda\, x : A.\, x)\ b$$

7.2 Lifting Data-Refinement

We can lift data-refinement in the same way that we lifted refinement in Chapter 6, i.e. we supply typings once at the top of a data-refinement, and implicitly pass them to sub-components. However when lifting data-refinement we supply two typings: one for abstract state type and one for the concrete state type. Our definition for lifted data-refinement is as follows:

$$a \sqsubseteq_{\kappa,\alpha} (R)\ k \quad \widehat{=} \quad a(\alpha) \sqsubseteq (\{\langle k, a\rangle : \mathcal{S}_\kappa \times \mathcal{S}_\alpha \mid R(k,a)\})\ k(\kappa)$$

Theorems proved at the set-transformer level can be recast at this lifted level. For example, that procedural refinement is the same as the identity data-refinement is stated as follows. For $a : \boldsymbol{\mathcal{M}}_\tau$ and $b : \boldsymbol{\mathcal{M}}_\tau$,

$$a \sqsubseteq_\tau b \quad \equiv \quad a \sqsubseteq_{\tau,\tau} (\lambda x\ y.\, x = y)\ b$$

7.3 Data-Refinement Rules

In this section we give theorems for composing data-refinement, and then list rules for the piece-wise data-refinement of the statements considered in this thesis. Following that, we discuss the data-refinement of procedure interfaces, and also the limitations of the data-refinement rules presented here with regard to the contextual data-refinement of programs. Our rules are neither calculational [68], nor complete [34].

7.3.1 Composing Data-Refinement

Data-refinement is not a transitive relation, and so cannot be used with earlier versions of window inference. However, it does compose with procedural refinement, and so will work with the flexible window inference described in Chapter 2. For $a, b : \boldsymbol{\mathcal{M}}_\alpha$ and $c : \boldsymbol{\mathcal{M}}_\kappa$, we have:

$$\frac{a \sqsubseteq_\alpha b \quad b \sqsubseteq_{\kappa,\alpha} (R)\ c}{a \sqsubseteq_{\kappa,\alpha} (R)\ c} \qquad \frac{a \sqsubseteq_{\kappa,\alpha} (R)\ b \quad b \sqsubseteq_\kappa c}{a \sqsubseteq_{\kappa,\alpha} (R)\ c}$$

Of less interest from a methodological perspective is the fact that we can compose data-refinement with itself. For $a : \boldsymbol{\mathcal{M}}_A$, $b : \boldsymbol{\mathcal{M}}_B$, and $c : \boldsymbol{\mathcal{M}}_C$, we have the following:

$$\frac{a \sqsubseteq_{B,A} (R)\ b \quad b \sqsubseteq_{C,B} (S)\ c}{a \sqsubseteq_{C,A} (S \circ R)\ c}$$

7.3.2 Data-Refinement Laws

Here we list the rules allowing the piece-wise data-refinement of lifted statements in our user-level refinement language. We first list the rules for atomic statements, and then for compound statements and procedures.

Skip

An abstract skip statement is data-refined under any relation to a concrete skip statement:

$$\mathsf{Skip} \sqsubseteq_{\kappa,\alpha}(R)\ \mathsf{Skip}$$

Assertion

An abstract assertion statement can be data-refined to a concrete assertion statement which respects the original condition mapped through the data-refinement relation:

$$\{P\} \sqsubseteq_{\kappa,\alpha}(R)\ \{\lambda\, k.\, \exists\, a : \mathcal{S}_\alpha.\, R(k,a) \wedge P(a)\}$$

Non-Deterministic Assignment

An abstract non-deterministic assignment statement can be data-refined to a concrete non-deterministic assignment statement which respects the original condition mapped through the data-refinement relation:

$$[P] \sqsubseteq_{\kappa,\alpha}(R)\ [\lambda\, k.\, \exists\, a : \mathcal{S}_\alpha.\, R(k,a) \wedge P(a)]$$

Free Specification Statement

To data-refine a free specification statement, the concrete precondition must preserve the abstract precondition mapped through the data-refinement relation. Also, assuming that the concrete (and abstract) preconditions do hold, the concrete postcondition must be able to establish some corresponding abstract postcondition:

$$\frac{\begin{array}{c}\forall\, k : \mathcal{S}_\kappa\ \ a : \mathcal{S}_\alpha.\, R(k,a) \wedge P_a(a) \Rightarrow P_k(k) \\ \left(\begin{array}{l}\forall\, ki, ko : \mathcal{S}_\kappa\ \ ai : \mathcal{S}_\alpha. \\ \quad R(ki,ai) \wedge P_a(ai) \wedge P_k(ki) \wedge Q_k(ko) \Rightarrow \\ \quad (\exists\, ao : \mathcal{S}_\alpha.\, R(ko,ao) \wedge Q_a(ao))\end{array}\right)\end{array}}{[P_a,\ Q_a] \sqsubseteq_{\kappa,\alpha}(R)\ [P_k,\ Q_k]}$$

Framed Specification Statement

To data-refine a framed specification statement, the concrete precondition must preserve the abstract precondition mapped through the data-refinement relation. Also, assuming that the concrete (and abstract) preconditions do hold, and that the concrete specification statement respects its frame, the concrete postcondition must be able to establish some corresponding abstract postcondition, while also respecting the abstract frame:

$$\frac{\begin{array}{c}\forall\, k : \mathcal{S}_\kappa \;\; a : \mathcal{S}_\alpha .\, R(k,a) \wedge P_a(a) \Rightarrow P_k(k) \\ \left(\begin{array}{l}\forall\, ki, ko : \mathcal{S}_\kappa \;\; ai : \mathcal{S}_\alpha . \\ \quad R(ki,ai) \wedge P_a(ai) \wedge P_k(ki) \wedge ki \text{ dsub } w_k = ko \text{ dsub } w_k \wedge Q_k(ko) \Rightarrow \\ \quad (\exists\, ao : \mathcal{S}_\alpha .\, R(ko,ao) \wedge Q_a(ao) \wedge ai \text{ dsub } w_a = ao \text{ dsub } w_a)\end{array}\right)\end{array}}{w_a : [P_a, \quad Q_a] \sqsubseteq_{\kappa,\alpha}(R) \;\; w_k : [P_k, \quad Q_k]}$$

State Assignment

We can data-refine a state assignment when the abstract state function F and the concrete state function G are both well-typed given the 'invariant' information in the data-refinement relation. i.e. we must prove the following side-conditions:

$$\forall\, a : \mathcal{S}_\alpha .\, (\exists\, k : \mathcal{S}_\kappa .\, R(k,a)) \Rightarrow F(a) : \mathcal{S}_\alpha$$
$$\forall\, k : \mathcal{S}_\kappa .\, (\exists\, a : \mathcal{S}_\alpha .\, R(k,a)) \Rightarrow G(k) : \mathcal{S}_\kappa$$

Then, we have the following rule, which lets us data-refine a state given that the state assignment functions preserve the data-refinement relation:

$$\frac{\forall\, a : \mathcal{S}_\alpha \;\; k : \mathcal{S}_\kappa .\, R(k,a) \Rightarrow R(G(k), F(a))}{\langle F \rangle \sqsubseteq_{\kappa,\alpha}(R) \;\; \langle G \rangle}$$

Single-Variable Assignment

We can data-refine a single-variable assignment when the abstract and concrete assignments are well-typed given the 'invariant' information in the data-refinement relation. That is, we must prove the following side-conditions:

$$\forall\, a : \mathcal{S}_\alpha .\, (\exists\, k : \mathcal{S}_\kappa .\, R(k,a)) \Rightarrow a[E(a)/v] : \mathcal{S}_\alpha$$
$$\forall\, k : \mathcal{S}_\kappa .\, (\exists\, a : \mathcal{S}_\alpha .\, R(k,a)) \Rightarrow k[F(k)/w] : \mathcal{S}_\kappa$$

Then, the following rule lets us data-refine a single-variable assignment statement if the state updates preserve the data-refinement relation:

$$\frac{\forall\, a : \mathcal{S}_\alpha \;\; k : \mathcal{S}_\kappa .\, R(k,a) \Rightarrow R(k[F(k)/w], a[E(a)/v])}{v := E \sqsubseteq_{\kappa,\alpha}(R) \;\; w := F}$$

Multiple-Variable Assignment

We can data-refine a multiple-variable assignment when the abstract and concrete assignments are well-typed given the 'invariant' information in the data-refinement relation. That is, we must prove the following side-conditions:

$$\forall\, a : \mathcal{S}_\alpha .\, (\exists\, k .\, R(k, a)) \Rightarrow a \oplus E(a) : \mathcal{S}_\alpha$$
$$\forall\, k : \mathcal{S}_\kappa .\, (\exists\, k .\, R(k, a)) \Rightarrow k \oplus F(k) : \mathcal{S}_\kappa$$

Then, the following rule lets us data-refine a multiple-variable assignment statement if the state updates preserve the data-refinement relation:

$$\frac{\forall\, a : \mathcal{S}_\alpha \;\; k : \mathcal{S}_\kappa .\, R(k, a) \Rightarrow R(k \oplus F(k), a \oplus E(a))}{\langle \overrightarrow{E} \rangle \sqsubseteq_{\kappa,\alpha} (R) \;\; \langle \overrightarrow{F} \rangle}$$

Sequential Composition

The sequential composition of two abstract statements can be data-refined to the sequential composition of two correspondingly data-refined concrete statements: For $l_a, r_a : \boldsymbol{\mathcal{M}}_\alpha$ and $l_k, r_k : \boldsymbol{\mathcal{M}}_\kappa$, we have:

$$\frac{l_a \sqsubseteq_{\kappa,\alpha} (R) \;\; l_k \qquad r_a \sqsubseteq_{\kappa,\alpha} (R) \;\; r_k}{l_a ; r_a \sqsubseteq_{\kappa,\alpha} (R) \;\; l_k ; r_k}$$

Angelic and Demonic Choice

The angelic (demonic) choice between two abstract statements can be data-refined to the angelic (demonic) choice between two correspondingly data-refined concrete statements. For $l_a, r_a : \boldsymbol{\mathcal{M}}_\alpha$ and $l_k, r_k : \boldsymbol{\mathcal{M}}_\kappa$, we have:

$$\frac{l_a \sqsubseteq_{\kappa,\alpha} (R) \;\; l_k \qquad r_a \sqsubseteq_{\kappa,\alpha} (R) \;\; r_k}{l_a \sqcup r_a \sqsubseteq_{\kappa,\alpha} (R) \;\; l_k \sqcup r_k} \qquad \frac{l_a \sqsubseteq_{\kappa,\alpha} (R) \;\; l_k \qquad r_a \sqsubseteq_{\kappa,\alpha} (R) \;\; r_k}{l_a \sqcap r_a \sqsubseteq_{\kappa,\alpha} (R) \;\; l_k \sqcap r_k}$$

Alternation

We can data-refine an alternation statement when we can data-refine each branch independently, and when the concrete condition is equivalent to the abstract condition. We also require that for any abstract state satisfying the guard, all other abstract states reachable through the data-refinement relation also satisfy the guard. For $l_a, r_a : \boldsymbol{\mathcal{M}}_\alpha$ and $l_k, r_k : \boldsymbol{\mathcal{M}}_\kappa$, we have:

$$\frac{\begin{array}{c} l_a \sqsubseteq_{\kappa,\alpha} (R) \;\; l_k \\ r_a \sqsubseteq_{\kappa,\alpha} (R) \;\; r_k \\ \forall\, a, b : \mathcal{S}_\alpha \;\; k : \mathcal{S}_\kappa .\, G_a(a) \wedge R(k, a) \wedge R(k, b) \Rightarrow G_a(b) \\ \forall\, k : \mathcal{S}_\kappa .\, (\exists\, a : \mathcal{S}_\alpha .\, G_a(a) \wedge R(k, a)) \Leftrightarrow G_k(k) \end{array}}{\mathsf{if}\ G_a\ \mathsf{then}\ l_a\ \mathsf{else}\ r_a\ \mathsf{fi} \sqsubseteq_{\kappa,\alpha} (R) \;\; \mathsf{if}\ G_k\ \mathsf{then}\ l_k\ \mathsf{else}\ r_k\ \mathsf{fi}}$$

Logical Constants and Logical Variables

If we can data-refine the body of a (bound) logical constant statement, then we can data-refine the logical constant statement as a whole. i.e. if for any x (in T) we can prove $a(x) : \boldsymbol{\mathcal{M}}_\alpha$ and $k(x) : \boldsymbol{\mathcal{M}}_\kappa$, then we have:

$$\frac{\forall\, x.\ a(x) \sqsubseteq_{\kappa,\alpha}(R)\ k(x)}{\mathsf{con}\ x.\ a(x) \sqsubseteq_{\kappa,\alpha}(R)\ \mathsf{con}\ x.\ k(x)} \qquad \frac{\forall\, x.\ T\, a(x) \sqsubseteq_{\kappa,\alpha}(R)\ k(x)}{\mathsf{con}\ x : T.\ a(x) \sqsubseteq_{\kappa,\alpha}(R)\ \mathsf{con}\ x : T.\ k(x)}$$

Similarly for (bound) logical variable statements, if for any x (in T) we can prove $a(x) : \boldsymbol{\mathcal{M}}_\alpha$ and $k(x) : \boldsymbol{\mathcal{M}}_\kappa$, then we have:

$$\frac{\forall\, x.\ a(x) \sqsubseteq_{\kappa,\alpha}(R)\ k(x)}{\mathsf{var}\ x.\ a(x) \sqsubseteq_{\kappa,\alpha}(R)\ \mathsf{var}\ x.\ k(x)} \qquad \frac{\forall\, x.\ T\, a(x) \sqsubseteq_{\kappa,\alpha}(R)\ k(x)}{\mathsf{var}\ x : T.\ a(x) \sqsubseteq_{\kappa,\alpha}(R)\ \mathsf{var}\ x : T.\ k(x)}$$

Recursion Blocks

Raw recursion blocks can be data-refined if their bodies can be data-refined given that the recursive calls are data-refined. Note that for the raw recursion block, calls are of the same state-type as the recursion block. For $\mathsf{regular}_\kappa(G)$, and F preserving $\boldsymbol{\mathcal{M}}_\alpha$, we have:

$$\frac{\bigwedge a\ k.\ a \sqsubseteq_{\kappa,\alpha}(R)\ k \Longrightarrow F(a) \sqsubseteq_{\kappa,\alpha}(R)\ G(k)}{\mathsf{re}(F)\,\mathsf{er} \sqsubseteq_{\kappa,\alpha}(R)\ \mathsf{re}(G)\,\mathsf{er}}$$

While-Do Loops

A while-do loop can be data-refined when its body can be data-refined, and when the concrete guarding condition is equivalent to the abstract condition mapped through the data-refinement relation. We also require that for any abstract state satisfying the guard, all related abstract states also satisfy the guard. For $b_a : \boldsymbol{\mathcal{M}}_\alpha\ b_k : \boldsymbol{\mathcal{M}}_\kappa$ we have the following:

$$\frac{\begin{array}{c}\forall\, k : \mathcal{S}_\kappa\ a : \mathcal{S}_\alpha.\ R(k,a) \wedge G(a) \Rightarrow (\forall\, b : \mathcal{S}_\alpha.\ R(k,b) \Rightarrow G(b)) \\ b_a \sqsubseteq_{\kappa,\alpha}(R)\ b_k\end{array}}{\mathsf{while}\ G\ \mathsf{do}\ b_a\ \mathsf{od} \sqsubseteq_{\kappa,\alpha}(R)\ \mathsf{while}\ \lambda\, k.\, \exists\, a : \mathcal{S}_\alpha.\ R(k,a) \wedge G(a)\ \mathsf{do}\ b_k\ \mathsf{od}}$$

Single-Variable Blocks

In order to data-refine a single-variable block, the body of the block must be data-refined, possibly under some different internal data-refinement relation. The internal data-refinement relation must be preserved on the initialised states from the external data-refinement relation on the original states. The external data-refinement relation on the final state must be established from

the internal data-refinement relation on the final state of the body of the block. That is, for $v, w : \mathcal{V}$, $a : \boldsymbol{\mathcal{M}}_{\alpha[T_a/v]}$, and $k : \boldsymbol{\mathcal{M}}_{\kappa[T_k/w]}$, we have the following:

$$\frac{\begin{array}{c} a \sqsubseteq_{\kappa[T_k/w],\alpha[T_a/v]} (R')\ k \\ \left(\begin{array}{c} \forall\, k' : \mathcal{S}_{\kappa[T_k/w]}\ k : \mathcal{S}_\kappa\ a : \mathcal{S}_\alpha .\, k \operatorname{dsub} \{w\} = k' \operatorname{dsub} \{w\} \land R(k,a) \Rightarrow \\ (\exists\, a' : \mathcal{S}_{\alpha[T_a/v]} .\, R'(k', a') \land a \operatorname{dsub} \{v\} = a' \operatorname{dsub} \{v\}) \end{array}\right) \\ \left(\begin{array}{c} \forall\, k' : \mathcal{S}_{\kappa[T_k/w]}\ a' : \mathcal{S}_{\alpha[T_a/v]}\ k : \mathcal{S}_\kappa\ a : \mathcal{S}_\alpha .\, R'(k', a') \land R(k,a) \Rightarrow \\ R(k'[k\text{`}w/w], a'[a\text{`}v/v]) \end{array}\right) \end{array}}{\mathsf{begin}\ v : T_a .\ a\ \mathsf{end} \sqsubseteq_{\kappa,\alpha} (R)\ \mathsf{begin}\ w : T_k .\ k\ \mathsf{end}}$$

A special case of the above theorem is when the external data-refinement relation is in fact the identity function, i.e. when it corresponds to procedural refinement. Then, we have the following theorem, which lets us refine a block by data-refining its body. For $v, w : \mathcal{V}$, $a : \boldsymbol{\mathcal{M}}_{\tau[T_a/v]}$, and $k : \boldsymbol{\mathcal{M}}_{\tau[T_k/w]}$, we have:

$$\frac{\begin{array}{c} a \sqsubseteq_{\tau[T_k/w],\tau[T_a/v]} (R)\ k \\ \left(\begin{array}{c} \forall\, s : \mathcal{S}_\tau\ k' : \mathcal{S}_{\tau[T_k/w]} .\, s \operatorname{dsub} w = k' \operatorname{dsub} w \Rightarrow \\ (\exists\, a' : \mathcal{S}_{\tau[T_a/v]} .\, R(k', a') \land s \operatorname{dsub} \{v\} = a' \operatorname{dsub} \{v\}) \end{array}\right) \\ \forall\, s : \mathcal{S}_\tau\ k' : \mathcal{S}_{\tau[T_k/w]}\ a' : \mathcal{S}_{\tau[T_a/v]} .\, R(k', a') \Rightarrow k'[s\text{`}w/w] = a'[s\text{`}v/v] \end{array}}{\mathsf{begin}\ v : T_a .\ a\ \mathsf{end}\ \sqsubseteq_\tau\ \mathsf{begin}\ w : T_k .\ k\ \mathsf{end}}$$

Multiple-Variable Blocks

In order to data-refine a multiple-variable block, the body of the block must be data-refined, possibly under some different internal data-refinement relation. The internal data-refinement relation must be preserved on the initialised states from the external data-refinement relation on the original states. The external data-refinement relation on the final state must be established from the internal data-refinement relation on the final state of the body of the block. That is, for $v, w : \mathcal{V}$, $a : \boldsymbol{\mathcal{M}}_{\alpha[T_a/v]}$, and $k : \boldsymbol{\mathcal{M}}_{\kappa[T_k/w]}$, we have the following. When $\operatorname{dom}(D_a) \subseteq \mathcal{V}$, $\operatorname{dom}(D_k) \subseteq \mathcal{V}$, $a : \boldsymbol{\mathcal{M}}_{\alpha \boxplus D_a}$, and $k : \boldsymbol{\mathcal{M}}_{\kappa \boxplus D_k}$, we have:

$$\frac{\begin{array}{c} a \sqsubseteq_{\kappa \boxplus D_k, \alpha \boxplus D_a} (R')\ k \\ \left(\begin{array}{c} \forall\, k' : \mathcal{S}_{\kappa \boxplus D_k}\ k : \mathcal{S}_\kappa\ a : \mathcal{S}_\alpha . \\ k \operatorname{dsub} \operatorname{dom}(D_k) = k' \operatorname{dsub} \operatorname{dom}(D_k) \land R(k,a) \Rightarrow \\ (\exists\, a' : \mathcal{S}_{\alpha \boxplus D_a} .\, R'(k', a') \land a \operatorname{dsub} \operatorname{dom}(D_a) = a' \operatorname{dsub} \operatorname{dom}(D_a)) \end{array}\right) \\ \left(\begin{array}{c} \forall\, k' : \mathcal{S}_{\kappa \boxplus D_k}\ a' : \mathcal{S}_{\alpha \boxplus D_a}\ k : \mathcal{S}_\kappa\ a : \mathcal{S}_\alpha .\, R'(k', a') \land R(k,a) \Rightarrow \\ R(k' \oplus (k \operatorname{dres} \operatorname{dom}(D_k)), a' \oplus (a \operatorname{dres} \operatorname{dom}(D_a))) \end{array}\right) \end{array}}{\mathsf{begin}\ D_a .\ a\ \mathsf{end} \sqsubseteq_{\kappa,\alpha} (R)\ \mathsf{begin}\ D_k .\ k\ \mathsf{end}}$$

Analogously with single-variable blocks, we can procedurally refine a multiple-variable block by data-refining its body. When $\mathrm{dom}(D_a) \subseteq \mathcal{V}$, $\mathrm{dom}(D_k) \subseteq \mathcal{V}$, $a : \boldsymbol{\mathcal{M}}_{\tau \boxplus D_a}$, and $k : \boldsymbol{\mathcal{M}}_{\tau \boxplus D_k}$, we have:

$$\frac{\begin{array}{c} a \sqsubseteq_{\tau \boxplus D_k, \tau \boxplus D_a} (R)\ \ k \\ \left(\begin{array}{l} \forall\, k : \mathcal{S}_{\tau \boxplus D_k}\ \ s : \mathcal{S}_\tau .\ s \ \mathrm{dsub}\ \mathrm{dom}(D_k) = k \ \mathrm{dsub}\ \mathrm{dom}(D_k) \Rightarrow \\ \quad (\exists\, a : \mathcal{S}_{\tau \boxplus D_a} .\ R(k, a) \wedge s \ \mathrm{dsub}\ \mathrm{dom}(D_a) = a \ \mathrm{dsub}\ \mathrm{dom}(D_a)) \end{array}\right) \\ \left(\begin{array}{l} \forall\, k : \mathcal{S}_{\tau \boxplus D_k}\ \ a : \mathcal{S}_{\tau \boxplus D_a}\ \ s : \mathcal{S}_\tau .\ R(k, a) \Rightarrow \\ \quad k \oplus (s \ \mathrm{dres}\ \mathrm{dom}(D_k)) = a \oplus (s \ \mathrm{dres}\ \mathrm{dom}(D_a)) \end{array}\right) \end{array}}{\mathsf{begin}\ D_a .\ a\ \mathsf{end} \sqsubseteq_\tau \ \mathsf{begin}\ D_k .\ k\ \mathsf{end}}$$

Interfaced Procedures

We can data-refine an interfaced procedure by data-refining its body and its calling instance. The external data-refinement relation will be the same for data-refining the body, but just as with local blocks, we might have a different data-refinement relation for the procedure implementation inside the parameterisation. Moreover, in the calling context, each instance of the procedure call may occur within the context of a different data-refinement relation. The data-refinement rule we present will deal with all of these difficulties. Here we will present a general form of the procedure data-refinement rule. Below, in Section 7.3.3, we will comment on how we can prove a more useful form of this rule when the procedure interface is a specification statement.

In the following, the external concrete and abstract declaration typings will be κ and α. The concrete and abstract implementation typings will be $\mathbb{T}_{D_k,\kappa}$ and $\mathbb{T}_{D_a,\alpha}$. The various calling typings will usually be denoted τ. For expository purposes, we'll define the following abbreviations. Our rule will have contextual information allowing us to data-refine a procedure call P with arguments a to a procedure call Q with arguments b. We represent this information with the following abbreviation:

$$\begin{array}{l} DCALL(P, Q, R, \kappa, \alpha, D_k, D_a) \mathrel{\widehat{=}} \\ \quad \bigwedge S\ \ \kappa'\ \ \alpha'\ \ a\ \ b. \\ \qquad [\![INITIAL(S, \kappa', \alpha', a, b, R, D_k, \kappa, D_a, \alpha); \\ \qquad\ \ FINAL(S, \kappa', \alpha', a, b, R, D_k, \kappa, D_a, \alpha)]\!] \Longrightarrow \\ \qquad\quad P(a) \sqsubseteq_{\kappa', \alpha'} (S)\ \ Q(b) \end{array}$$

The initialisation and finalisation conditions are defined as follows. They require that the initialisation and finalisation expressions are well-typed, and that the calling context's data-refinement relation S is suitably tied to the declaration context's data-refinement relation R:

$$
\begin{array}{l}
INITIAL(S, \kappa', \alpha', a, b, R, D_k, \kappa, D_a, \alpha) \mathrel{\widehat{=}} \\
\quad \forall\, y : \mathcal{S}_{\kappa'}\ \ i : \mathcal{S}_{\alpha'} .\, S(y, i) \Rightarrow (\forall\, k' : \mathcal{S}_{\mathbb{T}_{D_k,\kappa}} .\, \exists\, a' : \mathcal{S}_{\mathbb{T}_{D_a,\alpha}} . \\
\qquad (\exists\, w .\, \mathbb{I}_{D_k,b}(y) : \Pi_w \mathbb{T}_{D_k,\kappa} \wedge w \subseteq \mathcal{V}) \wedge \\
\qquad (\exists\, w .\, \mathbb{I}_{D_a,a}(i) : \Pi_w \mathbb{T}_{D_a,\alpha} \wedge w \subseteq \mathcal{V}) \wedge \\
\qquad R(k' \oplus \mathbb{I}_{D_k,b}(y), a' \oplus \mathbb{I}_{D_a,a}(i)))
\end{array}
$$

$$
\begin{array}{l}
FINAL(S, \kappa', \alpha', a, b, R, D_k, \kappa, D_a, \alpha) \mathrel{\widehat{=}} \\
\quad \forall\, y : \mathcal{S}_{\kappa'}\ \ i : \mathcal{S}_{\alpha'}\ \ k' : \mathcal{S}_{\mathbb{T}_{D_k,\kappa}}\ \ a' : \mathcal{S}_{\mathbb{T}_{D_a,\alpha}} .\, S(y, i) \wedge R(k', a') \Rightarrow \\
\qquad (\exists\, w .\, \mathbb{F}_{D_k,b}(k') : \Pi_w \kappa' \wedge w \subseteq \mathcal{V}) \wedge \\
\qquad (\exists\, w .\, \mathbb{F}_{D_a,a}(a') : \Pi_w \alpha' \wedge w \subseteq \mathcal{V}) \wedge \\
\qquad S(y \oplus \mathbb{F}_{D_k,b}(k'), i \oplus \mathbb{F}_{D_a,a}(a'))
\end{array}
$$

We will require that the concrete and abstract procedure implementations are monotonic predicate transformers i.e. $B_a : \boldsymbol{\mathcal{M}}_{\mathbb{T}_{D_a,\alpha}}$ and $B_k : \boldsymbol{\mathcal{M}}_{\mathbb{T}_{D_k,\kappa}}$, and that the calling contexts are monotonic predicate transformers, i.e. $(\bigwedge P .\, P : \boldsymbol{\mathcal{M}} \Longrightarrow C_a(P) : \boldsymbol{\mathcal{M}}_\alpha)$ and $(\bigwedge P .\, P : \boldsymbol{\mathcal{M}} \Longrightarrow C_k(P) : \boldsymbol{\mathcal{M}}_\kappa)$. These conditions can usually be automatically proved in a mechanised interactive environment. This leaves us with the main procedure data-refinement theorem, as follows:

$$
\frac{
\begin{array}{c}
I_a \ \sqsubseteq_{\mathbb{T}_{D_a,\alpha}}\ B_a \Longrightarrow I_k \ \sqsubseteq_{\mathbb{T}_{D_k,\kappa}}\ B_k \\
B_a \sqsubseteq_{\mathbb{T}_{D_k,\kappa},\mathbb{T}_{D_a,\alpha}} (R')\ B_k \\
\left(\bigwedge P\ \ Q .\, [\![P : \boldsymbol{\mathcal{M}};\ Q : \boldsymbol{\mathcal{M}};\ DCALL(P, Q, R, \kappa, \alpha, D_k, D_a)]\!] \Longrightarrow \right. \\
\left. \quad C_a(P) \sqsubseteq_{\kappa,\alpha} (R)\ \ C_k(Q) \right)
\end{array}
}{
\begin{array}{c}
\mathsf{proc}\ P(D_a)\ \mathsf{int} = I_a\ \mathsf{imp} = B_a\ \mathsf{in}\ C_a(P) \\
\sqsubseteq_{\kappa,\alpha} (R)\ \ \mathsf{proc}\ P(D_k)\ \mathsf{int} = I_k\ \mathsf{imp} = B_k\ \mathsf{in}\ C_k(P)
\end{array}
}
$$

This requires us to data-refine the calling context and the implementation. We are also required to show, given the old abstract interface is implemented by the old abstract implementation, that the new concrete interface implements the new concrete implementation. This is not a piece-wise condition, and so looks unsuitable for practical use. (In effect this would require us to replay the entire derivation of the concrete procedure implementation, which would obviate most of the gain from procedural abstraction.) However, when our procedure interface is a specification statement, we can prove a piece-wise form of this data-refinement rule which will require us to prove that the new concrete interface is an anti–data-refinement of the abstract interface. We discuss this further in Section 7.3.3.

Recursive Procedures

Data-refining recursive procedures is very similar to data-refining interfaced procedures. In interfaced procedures we data-refined the calling context un-

der the assumption that we could data-refine procedure calls. With recursive procedures, we will do this not just for initial calls in the calling context, but also for recursive calls in the implementation of the procedure. Just as for interfaced procedures, we will present a general form of data-refinement for recursive procedures. Similarly, we will be able to specialise this rule when our interface is a specification statement, yielding more practical piece-wise rules.

We will require various well-formedness conditions on recursive procedures we wish to data-refine. The abstract and concrete interfaces must be monotonic predicate transformers on their respective declaration types, i.e. $I_a : \boldsymbol{\mathcal{M}}_{\mathbb{T}_{D_a,\alpha}}$ and $I_k : \boldsymbol{\mathcal{M}}_{\mathbb{T}_{D_k,\kappa}}$. We also require that the calling contexts are well-typed, i.e.
$(\bigwedge P.\, P : \boldsymbol{\mathcal{M}} \Longrightarrow C_a(P) : \boldsymbol{\mathcal{M}}_\alpha)$ and $(\bigwedge P.\, P : \boldsymbol{\mathcal{M}} \Longrightarrow C_k(P) : \boldsymbol{\mathcal{M}}_\kappa)$.
and that the recursive calls are well-typed, i.e.:

$$(\bigwedge P\ v.[\![v : N;\ P : \boldsymbol{\mathcal{M}}]\!] \Longrightarrow B_a(v,P) : \boldsymbol{\mathcal{M}}_{\mathbb{T}_{D_a,\alpha}})$$
$$(\bigwedge P\ v.[\![v : N;\ P : \boldsymbol{\mathcal{M}}]\!] \Longrightarrow B_k(v,P) : \boldsymbol{\mathcal{M}}_{\mathbb{T}_{D_k,\kappa}})$$

Finally, the concrete implementation must be refinement monotonic, as follows:

$$(\bigwedge P\ Q\ v.[\![v : N;\ (\bigwedge \tau\ a.\, P(a) \sqsubseteq_\tau Q(a))]\!] \Longrightarrow B_k(v,P) \sqsubseteq_{\mathbb{T}_{D_k,\kappa}} B_k(v,Q))$$

The obligations above will all be automatically proved in a mechanised interactive environment, leaving us with the following data-refinement rule for recursive procedures:

$$\frac{\begin{array}{l}
\left(\begin{array}{l}\bigwedge k\ a.[\![a : \boldsymbol{\mathcal{M}}_{\mathbb{T}_{D_a,\alpha}};\ k : \boldsymbol{\mathcal{M}}_{\mathbb{T}_{D_k,\kappa}};\ a \sqsubseteq_{\mathbb{T}_{D_k,\kappa},\mathbb{T}_{D_a,\alpha}}(R')\ k;\ I_a \sqsubseteq_{\mathbb{T}_{D_a,\alpha}} a]\!] \Longrightarrow \\ \quad I_k \sqsubseteq_{\mathbb{T}_{D_k,\kappa}} k\end{array}\right) \\
\left(\begin{array}{l}\bigwedge k\ a\ v.[\![v : N;\ a(v) : \boldsymbol{\mathcal{M}}_{\mathbb{T}_{D_a,\alpha}};\ k(v) : \boldsymbol{\mathcal{M}}_{\mathbb{T}_{D_k,\kappa}}; \\ \qquad a(v) \sqsubseteq_{\mathbb{T}_{D_k,\kappa},\mathbb{T}_{D_a,\alpha}}(R')\ k(v);\ \{\lambda s.\, V_a(v,s)\}; I_a \sqsubseteq_{\mathbb{T}_{D_a,\alpha}} a(v)]\!] \Longrightarrow \\ \quad \{\lambda s.\, V_k(v,s)\}; I_k \sqsubseteq_{\mathbb{T}_{D_k,\kappa}} k(v)\end{array}\right) \\
\left(\begin{array}{l}\bigwedge P\ Q\ v.[\![v : N;\ P : \boldsymbol{\mathcal{M}};\ Q : \boldsymbol{\mathcal{M}};\ DCALL(P,Q,R',\kappa,\alpha,D_k,D_a)]\!] \Longrightarrow \\ \quad B_a(v,P) \sqsubseteq_{\mathbb{T}_{D_k,\kappa},\mathbb{T}_{D_a,\alpha}}(R')\ B_k(v,Q)\end{array}\right) \\
\left(\begin{array}{l}\bigwedge P\ Q.[\![P : \boldsymbol{\mathcal{M}};\ Q : \boldsymbol{\mathcal{M}};\ DCALL(P,Q,R,\kappa,\alpha,D_k,D_a)]\!] \Longrightarrow \\ \quad C_a(P) \sqsubseteq_{\kappa,\alpha}(R)\ C_k(Q)\end{array}\right)
\end{array}}{\begin{array}{r}\mathsf{rec}\ P(D_a)\ \mathsf{int} = I_a\ \mathsf{var} = v : N,\ V_a(v)\ \mathsf{imp} = B_a(v,P)\ \mathsf{in}\ C_a(P) \\ \sqsubseteq_{\kappa,\alpha}(R)\ \mathsf{rec}\ P(D_k)\ \mathsf{int} = I_k\ \mathsf{var} = v : N,\ V_k(v)\ \mathsf{imp} = B_k(v,P)\ \mathsf{in}\ C_k(P)\end{array}}$$

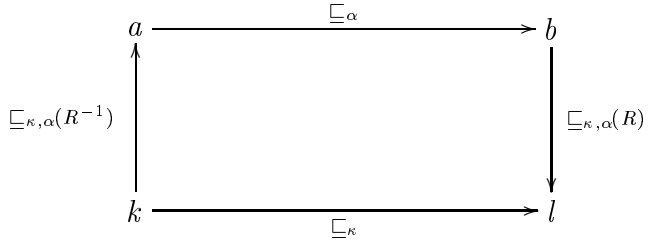
Importantly, this rule allows us to data-refine sub-components of the implementation and calling part of the procedure at different typings and with different data-refinement relations, depending on what (perhaps deeply

nested) context occurs at calls to the procedure. Also, when $\kappa = \alpha$ and the outer refinement R is the identity function, we could prove a theorem which allows us to refine a recursive procedure by data-refinement.

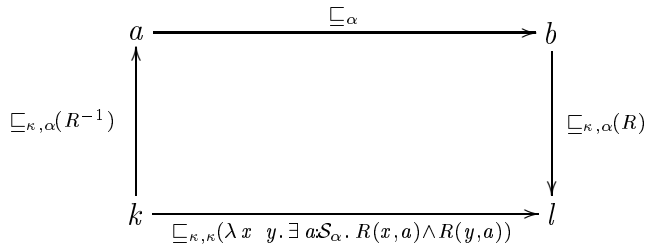
The theorem above is the general theorem for data-refining recursive procedures. When the interface is a specification statement, we can absorb the assert statement with the precondition of the specification statement, and also appeal to interface data-refinement theorems shown below in Section 7.3.3. We will not show these variants here. However, note that in the general theorem above, it appears that we have duplicated the interface refinement obligation. This 'duplication' is logically necessary: one obligation corresponds to the initial call, and the other to the recursive call. However, this does not lead to a duplication of proof obligations when we use the rule in practice. When our interface is a specification statement, the specialised forms of this general theorem can merge the common parts of these two obligations.

7.3.3 Interface Data-Refinement

Our generic procedure data-refinement rules go some way to supporting a piece-wise data-refinement methodology, as they allow us to data-refine the body and call of a procedure separately. However, given that we can perform these individual data-refinements, we are still required to establish that our new concrete interface is indeed an interface to our new concrete procedure body. We would prefer to just anti–data-refine the interfaces separately, rather than re-play the entire development of the concrete procedure implementation. So, given that our abstract interface a is implemented by b, that b data-refines to a concrete implementation l, and that our concrete interface k anti–data-refines a, we would like to be able to show that our concrete interface k refines to its implementation l. That is, we would like the following diagram to hold:



However, in general, we can't show the procedural refinement of k to l, but rather only that the following homogeneous data-refinement holds, when $a, b : \boldsymbol{\mathcal{M}}_\alpha$ and $k, l : \boldsymbol{\mathcal{M}}_\kappa$:



We must place conditions on R to make this homogeneous data-refinement $k \sqsubseteq_{\kappa,\kappa} (\lambda x\ \ y. \exists\, a{:}\mathcal{S}_\alpha. R(x,a) \wedge R(y,a))\ \ l$ correspond to the procedural refinement $k\ \ \sqsubseteq_\kappa\ \ l$. For example, a sufficient condition is that R is a total function.

However, when the interface is a specification statement, we can bundle our conditions on R into a context which can take advantage of the specification's precondition.

For free specification statements we have the following theorem. When $b : \boldsymbol{\mathcal{M}}_\alpha$ and $l : \boldsymbol{\mathcal{M}}_\kappa$,

$$\frac{\begin{array}{c}\forall\, k : \mathcal{S}_\kappa.\, P_k(k) \Rightarrow (\exists\, a : \mathcal{S}_\alpha.\, P_a(a) \wedge R(k,a)) \\ (\exists\, k : \mathcal{S}_\kappa.\, P_k(k)) \Rightarrow (\forall\, k : \mathcal{S}_\kappa.\, (\exists\, a.\, Q_a(a) \wedge R(k,a)) \Rightarrow Q_k(k)) \\ [P_a,\ \ Q_a]\ \ \sqsubseteq_\alpha\ \ b \qquad\qquad b \sqsubseteq_{\kappa,\alpha} (R)\ \ l\end{array}}{[P_k,\ \ Q_k]\ \ \sqsubseteq_\kappa\ \ l}$$

We have a similar theorem when our interface is a framed specification statement. Again, for $b : \boldsymbol{\mathcal{M}}_\alpha$ and $l : \boldsymbol{\mathcal{M}}_\kappa$, we have the following:

$$\frac{\begin{array}{c}\left(\begin{array}{l}\forall\, p_a, q_a : \mathcal{S}_\alpha\ \ p_k, q_k : \mathcal{S}_\kappa. \\ \quad P_a(p_a) \wedge Q_a(q_a) \wedge R(p_k,p_a) \wedge R(q_k,q_a) \wedge p_a \text{ dsub } v = q_a \text{ dsub } v \Rightarrow \\ \quad p_k \text{ dsub } w = q_k \text{ dsub } w\end{array}\right) \\ \forall\, k : \mathcal{S}_\kappa.\, P_k(k) \Rightarrow (\exists\, a : \mathcal{S}_\alpha.\, P_a(a) \wedge R(k,a)) \\ \left(\begin{array}{l}\forall\, i, o : \mathcal{S}_\kappa.\, P_k(i) \wedge (\exists\, a.\, Q_a(a) \wedge R(o,a)) \wedge i \text{ dsub } w = o \text{ dsub } w \Rightarrow \\ \quad Q_k(o)\end{array}\right) \\ v : [P_a,\ \ Q_a]\ \ \sqsubseteq_\alpha\ \ b \qquad\qquad b \sqsubseteq_{\kappa,\alpha} (R)\ \ l\end{array}}{w : [P_k,\ \ Q_k]\ \ \sqsubseteq_\kappa\ \ l}$$

We can use these conditions to 'simplify' our general interfaced procedure and recursive procedure data-refinement rules.

7.3.4 Contextual Data-Refinement

Our data-refinement rules have a limitation for practical use which is shared by standard presentations of data-refinement in the literature. Our rules

do not make use of information from the context of the development. For example, when we data-refine the branches of an alternation statement, we are not in general able to make use of the contextual information in the guard of the alternation.

A general way to solve this problem would be to propagate this contextual information throughout our development by using assertion statements. For example, an alternation data-refinement rule could introduce context in assumptions as follows. For $l_a, r_a : \boldsymbol{\mathcal{M}}_\alpha$ and $l_k, r_k : \boldsymbol{\mathcal{M}}_\kappa$, we have:

$$\frac{\begin{array}{c}\{\!|\, G \,|\!\};\, l_a \sqsubseteq_{\kappa,\alpha} (R)\ \{\!|\, \lambda\, s.\, \exists\, a : \mathcal{S}_\alpha.\ G(a) \wedge R(s,a) \,|\!\};\, l_k \\ \{\!|\, \lambda\, s.\, \neg\ G(s) \,|\!\};\, r_a \sqsubseteq_{\kappa,\alpha} (R)\ \{\!|\, \lambda\, s.\, \forall\, a : \mathcal{S}_\alpha.\ R(s,a) \Rightarrow \neg\ G(a) \,|\!\};\, r_k\end{array}}{\mathsf{if}\ G\ \mathsf{then}\ l_a\ \mathsf{else}\ r_a\ \mathsf{fi} \sqsubseteq_{\kappa,\alpha} (R)\ \mathsf{if}\ \lambda\, s.\, \exists\, a : \mathcal{S}_\alpha.\ G(a) \wedge R(s,a)\ \mathsf{then}\ l_k\ \mathsf{else}\ r_k\ \mathsf{fi}}$$

However, the work required to support this mode of development would go beyond the scope of this thesis. We can work around this problem by utilising the fact that specification statements carry their context in their preconditions—if the only atomic statements we data-refine are initialisations, specification statements, or interfaced procedure calls, we will not encounter any limitations on data-refinement in practice.

Chapter 8

Case Study: Propositional Tautology Checking

We demonstrate the use of our refinement theory through a case study: the refinement of the specification of a propositional tautology checker to a partially-implemented decision tree algorithm, and its subsequent data-refinement to an algorithm over reduced ordered negation trees. The algorithm we present is based on the binary decision diagram algorithm. The data-refinement progresses through several stages, first introducing several stronger data-type invariants, and then changing the data-type from decision trees to negation decision trees. We make particular use of statements acting on named program variables, and demonstrate that the theory of refinement developed in this thesis supports a modular style of development. However, the case-study has been proved using the mechanised refinement theory described in this thesis, and demonstrates the feasibility of the theory for application to non-trivial program development. For this presentation, we hide some details of the mechanisation beneath a thin layer of sugared syntax, but we alert the presence of proof obligations as they arise.

Efficient propositional tautology checking is at the heart of many modern model checking tools and hence plays an important role in the debugging and formal checking of hardware and communication protocols. A binary decision diagram (BDD) is a data structure for an efficient implementation of propositional tautology checking. BDDs can be seen as an efficient implementation of truth tables. In this chapter we formalise this perspective by presenting an BDD-related algorithm as a data-refinement from truth tables and decision trees. In this, we follow Harrison's [44, p162] informal remarks:

> The basic idea of binary decision diagrams is to build up a

> 'decision tree' [...] In that simple form, binary decision diagrams would merely be a way of organising a truth table, and therefore have little interest. But the following two refinements often lead to an enormous increase in their efficiency.
>
> - Rather than using trees, use directed acyclic graphs, sharing any subexpressions which arise, as proposed by Lee (1959) and Akers (1978).
> - Choose a canonical ordering of the variables at the outset, and arrange the graph such that the variables occur in that order down any branch, as proposed by Bryant (1986).

As with other previous correctness arguments for BDD algorithms [63, 67] our final algorithm uses trees and not graphs. I would expect that our work could be extended to utilise graphs sharing equivalent subexpressions; this has not been attempted because of time constraints. Our implementation reflects many other important aspects of BDD implementations, and includes an optimisation for negating decision trees suggested by Brace, Rudell and Bryant [21].

After presenting our specification of tautology checking, we give data refinements, in turn, to decision trees, reduced decision trees, ordered decision trees, negation decision trees, and finally reduced ordered negation decision trees. A data refinement effectively adds extra restrictions on an underlying data type, and provides corresponding opportunities to give more efficient and simpler implementations. This form of presentation highlights the specific role played by each part of the invariant on the final datatype.

8.1 The Initial Specification

The tautology checker will operate over finite propositional formulae containing variables constructed from a given set of variable names $\mathbb{V}$, and basic logical connectives. This language is a small, but complete set of connectives for first-order propositional logic:

$$\text{prop} \ ::= \ \text{VAR}\langle\langle\mathbb{V}\rangle\rangle \mid \text{NOT}\langle\langle\text{prop}\rangle\rangle \mid \text{prop AND prop}$$

We define discriminators is_Var, is_Not, and is_And, and destructors VarV, NotA, AndL, and AndR in the normal way. We consider a variable-valuation to be represented by a set of variables. So, for a variable-valuation V, and a variable v, if $v \in V$, then VAR v evaluates to true and otherwise to false. The meaning of a proposition is defined by $[\![\_]\!]\_ : \text{prop} \times \mathbb{P}\,\mathbb{V} \to \mathbb{B}$, a Boolean function defined recursively over propositions:

$$
\begin{array}{lcl}
[\![\text{VAR } v]\!]_V & \widehat{=} & v \in V \\
[\![\text{NOT } p]\!]_V & \widehat{=} & \neg\, [\![p]\!]_V \\
[\![p_1 \text{ AND } p_2]\!]_V & \widehat{=} & [\![p_1]\!]_V \wedge [\![p_2]\!]_V
\end{array}
$$

Tautologies are those propositions which are always true, that is, propositions which evaluate to true under every variable-valuation. So, the specification for our tautology checker can be stated as follows:

$$r : [\text{true}, \quad r \equiv \forall\, V : \mathbb{P}\,\mathbb{V}.\, [\![p]\!]_V]$$

Here, and throughout this chapter, we will refer to τ, a top-level typing environment such that $\tau(r) = \mathbb{B}$ and $\tau(p) = \text{prop}$.

A different and more demanding specification might also call for a counter-example if the proposition isn't valid. We will leave that elaboration for another time! A naive iterative solution to our specification could lead to the truth table algorithm, i.e. generate each variable-valuation in turn and evaluate the proposition under that variable-valuation.

8.2 Decision Trees

Rather than testing the proposition for validity directly, we construct an equivalent *decision tree*, and then test it for validity.

8.2.1 Notation and Data Types

A tree can be defined inductively as:

$$\text{tree} \;::=\; \text{Leaf}\langle\langle \mathbb{B} \rangle\rangle \;\big|\; \text{Node}\langle\langle \mathbb{V} \times \text{tree} \times \text{tree} \rangle\rangle$$

We define discriminators is_Leaf and is_Node, and destructors LeafT, NodeV, NodeL, and NodeR in the normal way. The left and right branches of a tree node represent the 'true' and 'false' branches of the node's variable-valuation. Thus $\{\!|\ \_\ |\!\}\_ : \text{tree} \times \mathbb{P}\,\mathbb{V} \to \mathbb{B}$, the 'meaning' of a tree for a particular variable-valuation can be defined as follows:

$$
\begin{array}{lcl}
\{\!|\, \text{Leaf}(b)\, |\!\}_V & \widehat{=} & b \\
\{\!|\, \text{Node}(v, l, r)\, |\!\}_V & \widehat{=} & \text{cond}(v \in V, \{\!|\, l\, |\!\}_V, \{\!|\, r\, |\!\}_V)
\end{array}
$$

However, instead of checking a tree's validity by testing it for each variable-valuation in turn, it might be easier if we could just check that all of the leaves of the tree are true. We define the leaves of a tree, tleaves : tree $\to \mathbb{B}$, as follows:

$$\begin{array}{lcl} \text{tleaves}(\text{Leaf}(b)) & \widehat{=} & b \\ \text{tleaves}(\text{Node}(v,l,r)) & \widehat{=} & \text{tleaves}(l) \wedge \text{tleaves}(r) \end{array}$$

We can certainly prove that, for t : tree:

$$\text{tleaves}(t) \Rightarrow \forall\, V : \mathbb{P}\,\mathbb{V}.\ \{\!|\, t \,|\!\}_V$$

However, the converse does not hold, because repeated variables in a root-to-leaf path are not ruled out. In such a tree, for some variable-valuations, the meaning function never considers some leaves, which may be false-valued. We need to restrict our attention to trees in which a variable can only appear once on any path from the root to any leaf. We define the set of these trees to be dtree:

$$\begin{array}{l} \text{Leaf}(b) \in \text{dtree} \\ \text{Node}(v,l,r) \in \text{dtree} \equiv v \notin \text{tvars}(l) \wedge v \notin \text{tvars}(r) \wedge l \in \text{dtree} \wedge r \in \text{dtree} \end{array}$$

where tvars : tree $\to \mathbb{P}\,\mathbb{V}$, the variables in a tree, is defined as follows:

$$\begin{array}{lcl} \text{tvars}(\text{Leaf}(b)) & \widehat{=} & \{\} \\ \text{tvars}(\text{Node}(v,l,r)) & \widehat{=} & \{v\} \cup \text{tvars}(l) \cup \text{tvars}(r) \end{array}$$

Now we can prove, for t : dtree:

$$\text{tleaves}(t) \equiv \forall\, V : \mathbb{P}\,\mathbb{V}.\ \{\!|\, t \,|\!\}_V$$

8.2.2 The Decision Tree Algorithm

Using the above equivalence we refine our initial specification, as follows:

$$\begin{array}{l} \quad r : [\text{true},\quad r \equiv \forall\, V : \mathbb{P}\,\mathbb{V}.\ [\![p]\!]_V] \\ \sqsubseteq_\tau \quad \textit{“Introduce Local Block, Sequential Composition”} \\ \quad \textsf{begin}\ t : \text{dtree}. \\ \qquad t : [\text{true},\quad (\forall\, V : \mathbb{P}\,\mathbb{V}.\ [\![p]\!]_V) \Leftrightarrow (\forall\, V : \mathbb{P}\,\mathbb{V}.\ \{\!|\, t \,|\!\}_V)]; \\ \qquad r : [(\forall\, V : \mathbb{P}\,\mathbb{V}.\ [\![p]\!]_V) \Leftrightarrow (\forall\, V : \mathbb{P}\,\mathbb{V}.\ \{\!|\, t \,|\!\}_V),\quad r \equiv \text{tleaves}(t)] \\ \quad \textsf{end} \end{array}$$

The first specification statement can be refined as shown below by constructing a decision tree point-wise equivalent to the input proposition, and by considering the possible cases of the input proposition p. In the context of this block, we will work in a generic typing setting τ_1, where $\tau_1(p) = \text{prop}$ and $\tau_1(t) = \text{dtree}$. For example, the typing $\tau[\text{dtree}/t]$ satisfies τ_1.

$$\begin{array}{l} \qquad t : [\text{true},\quad (\forall\, V : \mathbb{P}\,\mathbb{V}.\ [\![p]\!]_V) \Leftrightarrow (\forall\, V : \mathbb{P}\,\mathbb{V}.\ \{\!|\, t \,|\!\}_V)]; \\ \sqsubseteq_{\tau_1} \quad \textit{“Strengthen Postcondition”} \\ \qquad t : [\text{true},\quad \forall\, V : \mathbb{P}\,\mathbb{V}.\ [\![p]\!]_V \Leftrightarrow \{\!|\, t \,|\!\}_V] \end{array}$$

This post-condition will appear often in the following development. We'll abbreviate it as $BUILD(p, t)$.

$t : [\text{true}, \quad BUILD(p, t)]$
$\sqsubseteq_{\tau_1}$ *"Introduce Alternation, Local Block, Seq. Comp'n, Str. Post."*
if is_Var(p) then
  $t : [\text{is\_Var}(p), \quad t = \text{Node}(\text{VarV}(p), \text{Leaf}(\text{true}), \text{Leaf}(\text{false}))]$
else if is_Not(p) then
  begin e : dtree.
    $e : [\text{is\_Not}(p), \quad BUILD(\text{NotA}(p), e)]$;
    $t : [\text{is\_Not}(p) \wedge BUILD(\text{NotA}(p), e), \forall\, V : \mathbb{P}\,\mathbb{V}.\, \{\!|\, t \,|\!\}_V \Leftrightarrow \neg\, \{\!|\, e \,|\!\}_V]$
  end
else
  begin a, b : dtree.
    $a : [\text{is\_And}(p), \quad BUILD(\text{AndL}(p), a)]$;
    $b : [\text{is\_And}(p), \quad BUILD(\text{AndR}(p), b)]$;
    $t : [\text{is\_And}(p) \wedge BUILD(\text{AndL}(p), a) \wedge BUILD(\text{AndR}(p), b),$
      $\forall\, V : \mathbb{P}\,\mathbb{V}.\, \{\!|\, t \,|\!\}_V \Leftrightarrow \{\!|\, a \,|\!\}_V \wedge \{\!|\, b \,|\!\}_V]$
  end
fi

The development immediately above could be re-worked into a recursive procedure which established $BUILD$ for its arguments. Termination would then be guaranteed because its arguments always decrease in the subprop ordering. We will do this later, but for the moment we leave the development as it is.

There are now three undeveloped parts of this program: negating a tree, conjoining two trees, and checking that the leaves of the final tree are all true. We will develop the code for the final stage of checking the leaves of the tree in Section 8.3, and turn to consider the code for negating trees in Section 8.5. For now, though, consider the sub-problem of conjoining two trees. We further develop this by considering cases of the two trees a and b. Let us first define the following abbreviation:

$$JOIN(t, a, b) \mathrel{\widehat{=}} \forall\, V : \mathbb{P}\,\mathbb{V}.\, \{\!|\, t \,|\!\}_V \Leftrightarrow \{\!|\, a \,|\!\}_V \wedge \{\!|\, b \,|\!\}_V$$

In this context we will work with a generic typing τ_2 where $\{\tau_2(t), \tau_2(a), \tau_2(b)\} \subseteq \text{dtree}$.

$$
\begin{array}{l}
\quad t : [\text{true}, \quad JOIN(t, a, b)] \\
\sqsubseteq_{\tau_2} \quad \textit{“Introduce Alternation”} \\
\quad \mathsf{if}\ \text{is\_Leaf}(a)\ \mathsf{then}\ t : [\text{is\_Leaf}(a), \quad JOIN(t, a, b)] \\
\quad \mathsf{else\ if}\ \text{is\_Leaf}(b)\ \mathsf{then}\ t : [\text{is\_Leaf}(b), \quad JOIN(t, a, b)] \\
\quad \mathsf{else}\ t : [\text{is\_Node}(a) \wedge \text{is\_Node}(b), \quad JOIN(t, a, b)] \\
\quad \mathsf{fi}
\end{array}
$$

The first two specification statements are similar. Note that $JOIN(t, a, b) \Leftrightarrow JOIN(t, b, a)$. Let's consider a general case $JOIN(t, l, n)$, where l is a leaf node. Here we use a typing τ_3, such that $\{\tau_3(t), \tau_3(l), \tau_3(n)\} \subseteq \text{dtree}$. The conjunction of the two nodes will be the leaf node (or the other node) when the Boolean flag in the leaf is false (or true, respectively).

$$
\begin{array}{l}
\quad t : [\text{is\_Leaf}(l), \quad JOIN(t, l, n)] \\
\sqsubseteq_{\tau_3} \quad \textit{“Introduce Alternation, Assignment”} \\
\quad \mathsf{if}\ l = \text{Leaf}(\text{true})\ \mathsf{then}\ t := n\ \mathsf{else}\ t := l\ \mathsf{fi}
\end{array}
$$

Now let's consider conjoining two non-leaf nodes. Clearly if the trees are identical, then we can simply choose one of them:

$$
\begin{array}{l}
\quad t : [\text{is\_Node}(a) \wedge \text{is\_Node}(b), \quad JOIN(t, a, b)] \\
\sqsubseteq_{\tau_2} \quad \textit{“Introduce Alternation, Assignment”} \\
\quad \mathsf{if}\ a = b\ \mathsf{then}\ t := a \\
\quad \mathsf{else}\ t : [a \neq b \wedge \text{is\_Node}(a) \wedge \text{is\_Node}(b), \quad JOIN(t, a, b)] \\
\quad \mathsf{fi}
\end{array}
$$

However, there's not much more we can do with this statement at the moment. We need to produce a well-typed decision tree from the conjunction of the two trees. However, there's not yet a straightforward way to tell if the there are variables shared in the two trees. In particular, we can't yet easily show that any resulting tree won't have repeated variables on any root-to-leaf path. We leave the further development of this until Section 8.4.

8.3 Reduced Decision Trees

Let us now consider the ultimate specification statement in the main program:

$$r : [\text{true}, \quad r \equiv \text{tleaves}(t)]$$

It tests the validity of the final decision tree by checking that all of its leaves are true. A naive way of checking all of the leaves would involve recursively descending throughout the tree to check each leaf in turn. However, if

a node has two identical subtrees then we need check only one branch, rather than both. This simplification is justified because:

$$\text{tleaves}(\text{Node}(v, l, l)) \Leftrightarrow \text{tleaves}(l)$$

We call trees which do not have identical children 'reduced trees' and define them as follows:

$$\begin{array}{ll} \text{Leaf}(b) \in \text{rtree} & \\ \text{Node}(v, l, r) \in \text{rtree} \equiv & v \notin \text{tvars}(l) \wedge v \notin \text{tvars}(r) \wedge \\ & l \in \text{rtree} \wedge r \in \text{rtree} \wedge l \neq r \end{array}$$

Now, trivially by induction, all rtrees are dtrees. That is, rtree $\subseteq$ dtree, and from a simple induction and argument from contradiction, we have for t : rtree:

$$\text{tleaves}(t) \equiv t = \text{Leaf}(\text{true})$$

That is, to check that the leaves of a reduced decision tree are all true, we need only check that the tree is a true leaf node.

8.3.1 A Data Refinement to Reduced Trees

Recall that the current state of our top-level refinement is as follows.

$$\begin{array}{l} \quad r : [\text{true}, \quad r \equiv \forall\, V : \mathbb{P}\,\mathbb{V}.\, [\![p]\!]_V\,] \\ \sqsubseteq_\tau \\ \quad \textsf{begin}\ t : \text{dtree}. \\ \qquad t : [\text{true}, \quad (\forall\, V : \mathbb{P}\,\mathbb{V}.\, [\![p]\!]_V) \Leftrightarrow (\forall\, V : \mathbb{P}\,\mathbb{V}.\, \{\!|\, t \,|\!\}_V)]; \\ \qquad r : [(\forall\, V : \mathbb{P}\,\mathbb{V}.\, [\![p]\!]_V) \Leftrightarrow (\forall\, V : \mathbb{P}\,\mathbb{V}.\, \{\!|\, t \,|\!\}_V), \quad r \equiv \text{tleaves}(t)] \\ \quad \textsf{end} \end{array}$$

The data-refinement that we will perform is trivial: we just replace the type t : dtree by the type t : rtree in the block above. No change of representation takes place, and the predicates in our specification statements remain the same, but we do strengthen the underlying data-type invariant.

$$\begin{array}{l} \quad t : [\text{true}, \quad (\forall\, V : \mathbb{P}\,\mathbb{V}.\, [\![p]\!]_V) \Leftrightarrow (\forall\, V : \mathbb{P}\,\mathbb{V}.\, \{\!|\, t \,|\!\}_V)]; \\ \quad r : [(\forall\, V : \mathbb{P}\,\mathbb{V}.\, [\![p]\!]_V) \Leftrightarrow (\forall\, V : \mathbb{P}\,\mathbb{V}.\, \{\!|\, t \,|\!\}_V), \quad r \equiv \text{tleaves}(t)] \\ \sqsubseteq_{T[\text{rtree}/t], T[\text{dtree}/t]} (\lambda\, k \;\; a \,.\, k = a) \\ \quad t : [\text{true}, \quad (\forall\, V : \mathbb{P}\,\mathbb{V}.\, [\![p]\!]_V) \Leftrightarrow (\forall\, V : \mathbb{P}\,\mathbb{V}.\, \{\!|\, t \,|\!\}_V)]; \\ \quad r : [(\forall\, V : \mathbb{P}\,\mathbb{V}.\, [\![p]\!]_V) \Leftrightarrow (\forall\, V : \mathbb{P}\,\mathbb{V}.\, \{\!|\, t \,|\!\}_V), \quad r \equiv \text{tleaves}(t)] \end{array}$$

We can refine the block in our main development by data-refining its body using the above data-refinement. Thus, our current top-level refinement is now characterised by the following theorem.

$$
\begin{array}{l}
\quad r : [\text{true}, \quad r \equiv \forall\, V : \mathbb{P}\,\mathbb{V}.\, [\![p]\!]_V] \\
\sqsubseteq_\tau \\
\quad \textsf{begin}\ t : \text{rtree}. \\
\qquad t : [\text{true}, \quad (\forall\, V : \mathbb{P}\,\mathbb{V}.\, [\![p]\!]_V) \Leftrightarrow (\forall\, V : \mathbb{P}\,\mathbb{V}.\, \{\!| t |\!\}_V)]; \\
\qquad r : [(\forall\, V : \mathbb{P}\,\mathbb{V}.\, [\![p]\!]_V) \Leftrightarrow (\forall\, V : \mathbb{P}\,\mathbb{V}.\, \{\!| t |\!\}_V), \quad r \equiv \text{tleaves}(t)] \\
\quad \textsf{end}
\end{array}
$$

In the context of this stronger type of reduced trees, we can refine the ultimate specification statement to the following constant-time assignment:

$$
\begin{array}{ll}
\sqsubseteq_{T[\text{rtree}/t]} & \textit{“Introduce Assignment”} \\
\quad r := t = \text{Leaf}(\text{true}) &
\end{array}
$$

8.4 Ordered Decision Trees

We return to the problem of conjoining two non-leaf nodes. Recall that decision trees have the constraint that variables should not be repeated on any root-to-leaf path. This allows us to equate a tree's validity with it having only true leaves. How can we conjoin two trees and satisfy this condition? We need a constant-time test which lets us know about all of the variables in an entire sub-tree. One way to do this is to strictly order the variables in the tree.

We define otree to be the trees whose nodes have variables which are greater than the variables of any of their immediate children:

$$
\begin{array}{ll}
\text{Leaf}(b) \in \text{otree} & \\
\text{Node}(v, l, r) \in \text{otree} \equiv & \text{is\_Node}(l) \Rightarrow \text{NodeV}(l) < v \wedge \\
& \text{is\_Node}(r) \Rightarrow \text{NodeV}(r) < v \wedge \\
& l \in \text{otree} \wedge r \in \text{otree}
\end{array}
$$

By induction, this means that the variable of a node in an ordered tree is greater than the variables in any of its descendants, not just its immediate children. Thus the ordering conditions maintain the decision tree conditions, i.e. otree $\subseteq$ dtree.

8.4.1 A Data Refinement to Ordered Trees

In the development dealing with the conjunction of non-leaf nodes, we will data-refine decision trees d, a, and b to ordered trees. Like our data-refinement at the top-level to reduced trees, this data-refinement will be an identity data-refinement which does not introduce a new representation, but does strengthen the data-type invariant on the underlying typing environment. Our data-refinement is as follows:

$$
\begin{array}{l}
\quad \mathsf{if}\ a = b\ \mathsf{then}\ t := a \\
\quad \mathsf{else}\ t : [a \neq b \wedge \mathrm{is\_Node}(a) \wedge \mathrm{is\_Node}(b),\quad JOIN(t, a, b)] \\
\quad \mathsf{fi} \\
\sqsubseteq_{\tau_2[\mathrm{otree},\mathrm{otree},\mathrm{otree}/a,b,t],\tau_2} (\lambda k\ \ a.\, k = a) \\
\quad \mathsf{if}\ a = b\ \mathsf{then}\ t := a \\
\quad \mathsf{else}\ t : [a \neq b \wedge \mathrm{is\_Node}(a) \wedge \mathrm{is\_Node}(b),\quad JOIN(t, a, b)] \\
\quad \mathsf{fi}
\end{array}
$$

Our new stronger typing environment provides us with opportunities for further procedural refinement. We will declare three new program variables v, x, and y, which we will use to construct our new tree:

$$
\begin{array}{l}
\quad t : [a \neq b \wedge \mathrm{is\_Node}(a) \wedge \mathrm{is\_Node}(b),\quad JOIN(t, a, b)] \\
\sqsubseteq_{\tau_2[\mathrm{otree},\mathrm{otree},\mathrm{otree}/a,b,t]} \quad \textit{“Introduce Local Block, Seq. Comp’n”} \\
\quad \mathsf{begin}\ v : \mathbb{V}, x : \mathrm{otree}, y : \mathrm{otree}. \\
\qquad v, x, y : [a \neq b \wedge \mathrm{is\_Node}(a) \wedge \mathrm{is\_Node}(b),\quad JOIN(\mathrm{Node}(v, x, y), a, b)]; \\
\qquad t : [a \neq b \wedge \mathrm{is\_Node}(a) \wedge \mathrm{is\_Node}(b) \wedge JOIN(\mathrm{Node}(v, x, y), a, b), \\
\qquad\quad JOIN(t, a, b)] \\
\quad \mathsf{end}
\end{array}
$$

Let us look at the first specification statement. In this context we define τ_4, an abbreviation for our typing environment:
$\tau_4 \mathrel{\widehat{=}} \tau_2[\mathrm{otree}, \mathrm{otree}, \mathrm{otree}, \mathbb{V}, \mathrm{otree}, \mathrm{otree}/a, b, t, v, x, y]$.

We choose the value of the new variable v to be the greatest of the two nodes’ variables. We choose x and y depending upon the following equivalences. When $A \mathrel{\widehat{=}} (P \Rightarrow X) \wedge (\neg\, P \Rightarrow Y)$, and
$B \mathrel{\widehat{=}} (Q \Rightarrow W) \wedge (\neg\, Q \Rightarrow Z)$, then we have the following:

$$
\begin{array}{lll}
A \wedge B & \Leftrightarrow & (P \Rightarrow (X \wedge W)) \wedge (\neg\, P \Rightarrow (Y \wedge Z)) \quad \text{when } P = Q \\
A \wedge B & \Leftrightarrow & (P \Rightarrow (X \wedge B)) \wedge (\neg\, P \Rightarrow (Y \wedge B)) \\
A \wedge B & \Leftrightarrow & (Q \Rightarrow (A \wedge W)) \wedge (\neg\, Q \Rightarrow (A \wedge Z))
\end{array}
$$

So, the refinement is as follows:

$v, x, y : [a \neq b \wedge \text{is\_Node}(a) \wedge \text{is\_Node}(b), \quad JOIN(\text{Node}(v, x, y), a, b)]$

$\sqsubseteq_{\tau_4}$ *"Introduce Alternation"*

if $\text{NodeV}(a) = \text{NodeV}(b)$ then

$v, x, y : [a \neq b \wedge \text{is\_Node}(a) \wedge \text{is\_Node}(b) \wedge \text{NodeV}(a) = \text{NodeV}(b),$
$v = \text{NodeV}(a) \wedge$
$JOIN(x, \text{NodeL}(a), \text{NodeL}(b)) \wedge JOIN(y, \text{NodeR}(a), \text{NodeR}(b))]$

else if $\text{NodeV}(a) < \text{NodeV}(b)$ then

$v, x, y : [a \neq b \wedge \text{is\_Node}(a) \wedge \text{is\_Node}(b) \wedge \text{NodeV}(a) < \text{NodeV}(b),$
$v = \text{NodeV}(b) \wedge$
$JOIN(x, a, \text{NodeL}(b)) \wedge JOIN(y, a, \text{NodeR}(b))]$

else

$v, x, y : [a \neq b \wedge \text{is\_Node}(a) \wedge \text{is\_Node}(b) \wedge \text{NodeV}(b) < \text{NodeV}(a),$
$v = \text{NodeV}(a) \wedge$
$JOIN(x, \text{NodeL}(a), b) \wedge JOIN(y, \text{NodeR}(a), b)]$

fi fi

In order to satisfy the well-formedness side-conditions for this refinement, we need to know that v is greater than any variable in the created subtrees x or y. We strengthen our definition of JOIN to the following:

$$JOIN(t, a, b) \;\widehat{=}\; \text{tvars}(t) \subseteq \text{tvars}(a) \cup \text{tvars}(b) \wedge \forall\, V : \mathbb{P}\,\mathbb{V}.\, \{\!|\, t \,|\!\}_V \Leftrightarrow \{\!|\, a \,|\!\}_V \wedge \{\!|\, b \,|\!\}_V$$

The previous development can be replayed under this stronger definition, and it allows us to prove the the well-formedness conditions discussed above.

We will leave the development of the remaining specification statements until later, and now turn to developing code to negate a decision tree.

8.5 Negation Decision Trees

We can negate a decision tree by negating the value of all of its leaves. We can define a tree negation operator negtree : $\mathbb{B} \times \text{tree} \to \text{tree}$ as follows:

$$\text{negtree}(p, \text{Leaf}(b)) \;\widehat{=}\; \text{Leaf}(p \equiv b)$$
$$\text{negtree}(p, \text{Node}(v, l, r)) \;\widehat{=}\; \text{Node}(v, \text{negtree}(p, l), \text{negtree}(p, r))$$

This is defined over a Boolean p for later convenience. This operator has our desired properties. For t : tree and $V : \mathbb{P}\,\mathbb{V}$,

$$\{\!|\, \text{negtree}(\text{false}, t) \,|\!\}_V \Leftrightarrow (\neg\, \{\!|\, t \,|\!\}_V)$$
$$\text{negtree}(\text{true}, t) = t$$

However, to implement tree negation in this naive way would be expensive. Brace, Rudell and Bryant [21] proposed an optimisation to the standard BDD algorithm to quickly negate a BDD. The idea is for edges of the tree to have a negation flag, which indicates whether the incident tree is to be read as having been negated. This allows a constant-time negation operation: simply flip the negation flag for the tree under consideration.

We follow this idea, and define a type of proto-negation trees Ntree as follows:

$$\text{Ntree} \ ::= \ \text{NLeaf} \mid \text{NNode}\langle\langle \mathbb{V} \times \text{Ntree} \times \mathbb{B} \times \text{Ntree} \rangle\rangle$$

We define discriminators is_NLeaf and is_NNode, and destructors NNodeV, NNodeL, NNodeN and NNodeR in the normal way. The Boolean value influences the interpretation of the right subtree. Brace, Rudell and Bryant discuss how having flags on each subtree can break the canonicality of BDDs. They leave space for flags on all edges, but impose constraints limiting left edges to true. We simply exclude this redundancy from our representation.

We can recover a tree from a proto-negation tree by using the tree_of_Ntree : Ntree $\to$ tree operator defined as follows:

$$\begin{array}{l} \text{tree\_of\_Ntree}(\text{NLeaf}) \mathrel{\widehat{=}} \text{Leaf}(\text{true}) \\ \text{tree\_of\_Ntree}(\text{NNode}(v, l, p, r)) \mathrel{\widehat{=}} \\ \quad \text{Node}(v, \text{tree\_of\_Ntree}(l), \text{negtree}(b, \text{tree\_of\_Ntree}(r))) \end{array}$$

Proto-negation trees do not allow us to represent all trees. For example, there is no representation of Leaf(false). We define negation trees proper as proto-negation trees paired with a flag to influence the interpretation of the whole tree:

$$\text{ntree} \mathrel{\widehat{=}} \mathbb{B} \times \text{Ntree}$$

We define discriminators is_nleaf and is_nnode, and destructors nneg, nvar, and nrneg by lifting from the corresponding operators on proto-negation trees, as follows:

$$\begin{array}{lcllcl} \text{is\_nnode}(t) & \mathrel{\widehat{=}} & \text{is\_NNode}(\text{snd}(t)) & \text{nneg}(t) & \mathrel{\widehat{=}} & \text{fst}(t) \\ \text{is\_nleaf}(t) & \mathrel{\widehat{=}} & \text{is\_NLeaf}(\text{snd}(t)) & \text{nvar}(t) & \mathrel{\widehat{=}} & \text{NNodeV}(\text{snd}(t)) \\ & & & \text{nrneg}(t) & \mathrel{\widehat{=}} & \text{NNodeN}(\text{snd}(t)) \end{array}$$

The 'desctructors' for left and right subtrees (nleft, and nright) result in negation trees which take into account interpretation under the various flags:

$$\begin{array}{lcl} \text{nleft}(t) & \mathrel{\widehat{=}} & \langle \text{nneg}(t), \text{NNodeL}(\text{snd}(t)) \rangle \\ \text{nright}(t) & \mathrel{\widehat{=}} & \langle \text{nneg}(t) \equiv \text{nrneg}(t), \text{NNodeR}(\text{snd}(t)) \rangle \end{array}$$

We can extract a tree from a negation tree by using the tree_of_ntree : ntree $\rightarrow$ tree operator defined as follows:

$$\text{tree\_of\_ntree}(t) \;\widehat{=}\; \text{negtree}(\text{fst}(t), \text{tree\_of\_Ntree}(\text{snd}(t)))$$

So, the negation of a negation-tree can be performed by the following constant-time operation:

$$\text{neg\_ntree}(t) \mathrel{\widehat{=}} \langle \neg\ \text{fst}(t), \text{snd}(t) \rangle$$

It has our desired property. For t : ntree, and $V : \mathbb{P}\,\mathbb{V}$, we have:

$$\{\!|\, \text{tree\_of\_ntree}(t) \,|\!\}_V \Leftrightarrow (\neg\ \{\!|\, \text{tree\_of\_ntree}(\text{neg\_ntree}(t)) \,|\!\}_V)$$

8.5.1 A Data Refinement to Negation Trees

The trees in our negation specification are decision trees. We define proto-negation decision trees dNtree to be those proto-negation trees whose corresponding trees are decision trees:

$$\text{dNtree} \mathrel{\widehat{=}} \{t : \text{Ntree} \mid \text{tree\_of\_Ntree}(t) \in \text{dtree}\}$$

Decision negation trees can then be defined as follows:

$$\text{dntree} \mathrel{\widehat{=}} \mathbb{B} \times \text{dNtree}$$

We need to data-refine the whole context for tree negation. For space reasons we do not show it, but it is a direct piece-wise data-refinement with the following relation:

$$\sqsubseteq_{T[\text{dntree}/t],\, T[\text{dtree}/t]} (\lambda\, k\ \ a.\ a`t = \text{tree\_of\_ntree}(k`t) \wedge a\ \text{dsub}\ \{t\} = k\ \text{dsub}\ \{t\})$$

The abstract and concrete values of program variable t are represented by $a`t$ and $k`t$ respectively. The last conjunct is similar to the frame of a framed specification statement, and ensures that variables not under consideration are not affected by the data-refinement.

In this new more concrete context, we can refine the negation operation to a more efficient-looking specification statement:

$$\begin{array}{l} t : [\text{is\_Not}(p) \wedge \mathit{BUILD}(\mathit{NotArg}(p), e),\quad \forall\, V : \mathbb{P}\,\mathbb{V}.\ \{\!|\, t \,|\!\}_x \Leftrightarrow (\neg\ \{\!|\, e \,|\!\}_V)] \\ \sqsubseteq_{B[\mathit{dntree}/e]} \\ t : [\text{is\_Not}(p) \wedge \mathit{BUILD}(\text{NotA}(p), \text{tree\_of\_ntree}(e)),\quad t = \text{neg\_ntree}(e)] \end{array}$$

8.6 Reduced Ordered Negation Trees

In our refinement so far we introduced reduced trees and negation trees to allow us to implement constant time tests for the validity of a tree and tree negation, and also ordered trees in order to simplify the conjunction of tree nodes. These constraints are all orthogonal and do not interact in our refinement in any deliterious way. In fact, reduced ordered trees are canonical. For reduced-ordered decision trees defined as follows:

$$\text{rotree} \mathrel{\widehat{=}} \text{rtree} \cap \text{otree}$$

We have, for a, b : rontree:

$$(\forall\, V : \mathbb{P}\,\mathbb{V}.\ \{\!|\, a \,|\!\}_V \Leftrightarrow \{\!|\, b \,|\!\}_V) \equiv (a = b)$$

This observation is Bryant's important contribution [23] to the popularity of BDDs, and is critical to the efficiency of BDD algorithms. In a graph with maximal sharing of identical subgraphs, canonicality means this is also maximal sharing of logically equivalent subgraphs. Thus, equivalent subexpressions are never recomputed.

We now fit our constraints on reduced and ordered trees onto our datatype of negation trees, and then complete the development of an executable program.

8.6.1 A Data Refinement to Reduced Ordered Negation Trees

We define reduced ordered proto-negation trees, and then reduced ordered negation trees as follows:

$$\begin{array}{lcl} \text{roNtree} & \mathrel{\widehat{=}} & \{t : \text{Ntree} \mid \text{tree\_of\_Ntree}(t) \in \text{rotree}\} \\ \text{rontree} & \mathrel{\widehat{=}} & \mathbb{B} \times \text{roNtree} \end{array}$$

We data-refine all of the code fragments we have developed so far, changing the value of any occurrences variables v of type otree or rtree to variables v with value tree_of_ntree(v) and type rontree. Tests is_Leaf become is_nleaf, and constants Leaf(b) become $\langle b, \text{NLeaf}\rangle$. The data-refinements are similar to those shown previously, and all include a framing condition to constrain the effect of the data-refinement.

8.6.2 Introducing Procedures and Recursive Procedures

We now collect together our code-fragments, and introduce (recursive) procedures to implement our algorithm.

We start with the current state of our top-level development, adapted from Section 8.3:

$$
\begin{array}{l}
\mathsf{begin}\ t : \mathrm{rontree}. \\
\quad t : [\mathrm{true},\quad BUILD(p, \mathrm{tree\_of\_ntree}(t)]; \\
\quad r := t = \langle \mathrm{true}, \mathrm{Leaf}(\mathrm{true}) \rangle \\
\mathsf{end}
\end{array}
$$

Using our recursive procedure introduction rule for framed specification statement interfaces, we can introduce the *Build* procedure by refinement as follows:

$$
\begin{array}{l}
\sqsubseteq_\tau \\
\quad \mathsf{rec}\ Build(\mathsf{value}\ p : \mathrm{prop}, \mathsf{result}\ t : \mathrm{rontree}) \\
\quad \mathsf{int} = t : [\mathrm{true},\quad BUILD(p, \mathrm{tree\_of\_ntree}(t))] \\
\quad \mathsf{var} = V : \mathrm{prop}, p\ \mathrm{subprop}\ V \\
\quad \mathsf{imp} = t : [p = V,\quad BUILD(p, \mathrm{tree\_of\_ntree}(t))] \\
\quad \mathsf{in} \\
\quad \mathsf{begin}\ t : \mathrm{rontree}. \\
\qquad t : [\mathrm{true},\quad BUILD(p, \mathrm{tree\_of\_ntree}(t)]; \\
\qquad r := t = \langle \mathrm{true}, \mathrm{Leaf}(\mathrm{true}) \rangle \\
\quad \mathsf{end}
\end{array}
$$

The initial specification statement is a (trivial) instance of *Build*'s interface and so, using the refinement-monotonicity of initial calls to recursive procedures, we can refine the initial specification statement to $Build(p, t)$:

$$
\begin{array}{l}
\sqsubseteq_\tau \\
\quad \mathsf{rec}\ Build(\mathsf{value}\ p : \mathrm{prop}, \mathsf{result}\ t : \mathrm{rontree}) \\
\quad \mathsf{int} = t : [\mathrm{true},\quad BUILD(p, \mathrm{tree\_of\_ntree}(t))] \\
\quad \mathsf{var} = V : \mathrm{prop}, p\ \mathrm{subprop}\ V \\
\quad \mathsf{imp} = t : [p = V,\quad BUILD(p, \mathrm{tree\_of\_ntree}(t))] \\
\quad \mathsf{in} \\
\quad \mathsf{begin}\ t : \mathrm{rontree}. \\
\qquad Build(p, t); \\
\qquad r := t = \langle \mathrm{true}, \mathrm{Leaf}(\mathrm{true}) \rangle \\
\quad \mathsf{end}
\end{array}
$$

We then refine the implementation of *Build* by appealing to the refinement-monotonicity of recursive procedure implementations, and by using an adaptation of the code fragment shown in Section 8.2.2. The specification statements containing postconditions *BUILD* are instances of the interface to the *Build* procedure, and as they decrease their instance of p in the subprop ordering, we can refine them to calls to *Build*.

In order to implement the *JOIN* specification statement, we declare a recursive procedure *Join* with the following declaration:

$$
\begin{array}{ll}
\mathsf{rec} & \mathit{Join}(\mathsf{value}\ a : \mathrm{rontree}, \mathsf{value}\ b : \mathrm{rontree}, \mathsf{result}\ t : \mathrm{rontree}) \\
\mathsf{int} = & t : [\mathrm{true},\ \ \mathit{JOIN}(\mathrm{tree\_of\_ntree}(a), \mathrm{tree\_of\_ntree}(b), \mathrm{tree\_of\_ntree}(t))] \\
\mathsf{var} = & T : \mathrm{tree} \times \mathrm{tree}, \\
& \langle \mathrm{tree\_of\_ntree}(a), \mathrm{tree\_of\_ntree}(b) \rangle\ (\mathrm{subtree} \times\!\times \mathrm{subtree})\ T
\end{array}
$$

The $\times\!\times$ operator is the lexicographic ordering on pairs, as the *Join* procedure may reduce either of its first two arguments. The development follows in a manner largely similar to the *Build* procedure. During the development of the implementation to *Join* we introduce two procedures: *JoinLeaf* for joining leaves, and *JoinNode* for joining nodes. *JoinNode* contains recursive calls to *Join*. We present the final code for our algorithm in Appendix D.

8.7 Remarks

The data refinement here has been presented in several stages. We have incrementally imposed constraints, which have allowed us to develop parts of our program efficiently. This shows us where each part of our final data-type invariant is used:

decision: Decision trees let us test validity of a tree by checking that its leaves are true.

reduced: A reduced decision tree lets us test the validity of a tree by checking that it is the true leaf.

ordered: Ordered trees are decision trees, and the ordering constraint allows us, when conjoining trees, to determine in constant time that the variable in the conjoined tree is different to any of the variables occurring in the subtrees.

negation: Negation trees allow us to implement tree negation operation in constant time.

The final program is not a *fait accompli*—other implementations are certainly possible, and carving the problem in other ways may reveal other insights.

Our semantics seems to serve as a nice level of abstraction for tackling correctness issues in program development, where we have avoided details concerning machine or programming language peculiarities. The perhaps laborious nature of the development given here is essentially due to the nature of the problem and our chosen exposition, rather than being due to 'artificial' hurdles imposed by the semantics of our refinement langauge.

Chapter 9

Conclusion

Refinement tools must give ready access to the standard results in classical mathematics, must be expressive enough to represent all of the statements in our refinement language, should support the process of our refinement methodology, and should facilitate the valid realisation of completely developed programs. This dissertation has presented new ideas and techniques which endeavour to address these requirements.

We described a generalised form of window inference which allows the transformation of terms under non-preorder relations. We can make use of this generality in the transformation of programs under the data-refinement relation. Another benefit of this inference scheme is that it allows window opening at various points on the top-level term. The form of window inference rules motivated the form of many of the refinement rules provided.

Our refinement theory is mechanised in the theorem prover Isabelle/ZF. Isabelle is an LCF theorem prover, whose rigorous support provides us with the security to push the boundaries of expressing complex and subtle refinement rules. The semantics for our refinement language is given definitionally in Isabelle/ZF, a logic of untyped set theory. Using a shallow embedding within a classical logic gives us license to freely mix our refinement logic with ordinary logic. These considerations are the basis for our confidence that the refinement rules presented in this thesis are sound and that derivations done in the tool will be logically accurate. Isabelle/ZF's set theory is a mechanisation of standard classical mathematics. The large collection of mathematical results stemming from this basis is available both for the development of the refinement theory, and for the development of theories of particular application domains.

As one would hope, most of the refinement rules presented in this thesis are similar to existing rules in the literature. Nonetheless, we have also presented some newly mechanised refinement rules, as well as some original

refinement rules not previously seen in the literature.

We presented newly mechanised rules for introducing variable assignment statements and for transforming framed specification statements. Our explicit representation of variable names makes it possible to express these statements and rules. These rules resemble previously mechanised rules for state assignment and nondeterministic assignment statements [2, 13], but use explicit variable names to limit the effect of the statements. Our rules for introducing and refining local blocks appear similar to earlier mechanised rules for local blocks, but have a different effect in that they can hide variable names appearing in a higher scope.

Named program variables are central to programming languages, and also to the refinement calculus. However, unlike normal programming languages, the refinement calculus allows us to introduce arbitrary abstract types for local program variables. This makes it more difficult to represent state types for refinement languages. Previous representations in simply-typed frameworks had either not represented variable names, or else had fixed the range of types for program variables. Isabelle/ZF provides us with the flexibility to represent states as dependently typed functions from variable names to under-specified families of types. This has allowed us to explore definitions for constructs such as framed specification statements, variable assignment, local blocks and procedure parameterisation which have not previously been represented in a uniform way within a classical logic. Our parameterisation mechanism is also novel in performing the instantiation of multiple formal parameters in parallel, rather than sequentially, as suggested by previous presentations [70].

The expressiveness of Isabelle/ZF is achieved at the cost of explicit typing. Bounding sets must be supplied for many set-theoretic operators. We have been able to ameliorate this burden of the explicit representation by using the meta-logic of Isabelle/ZF to lift our set-transformer semantics to predicate transformers. In our set-transformer representation of weakest precondition semantics, we represented conditions and Boolean expressions as sets of states, and value-returning expressions as object-logic functions on states. In the lifted language, conditions and Boolean expressions become predicates on states, and value-returning expressions become meta-level functions. Moreover, when we lift the set-transformer language, we can abstract the state-type so that we are only required to mention the state-type once at the top of a program. Lifted statements implicitly pass the state type to subcomponents. When we fixed the structure of our program states to represent program variables, we were able to revisit the lifting of our set-transformer language so as to exploit this extra structure. Instead of abstracting the state-type, we instead abstracted the typing for the state type. This served

as a convenient abbreviation, but also facilitated the presentation of type updating operators for blocks and parameterisation. These statements implicitly modify the state type passed to sub-components.

Using a shallow embedding within a classical logic lets us freely mix our refinement theory with ordinary logic. This has proved critical for stating and proving refinement rules for interfaced recursion blocks and (recursive) procedures. We have introduced original rules for interfaced recursion blocks and (recursive) interfaced parameterised procedures. We contextually embed refinement assumptions in assertion statements within another refinement, which allows us to prove rules supporting step-wise refinement of recursion blocks and (recursive) procedures. These rules make significant use of Isabelle's meta-logic. They are schematic rules where the expression of obligations is delayed until the introduction of a procedure call. This allows us to make procedure calls from any type, and with any actual arguments. The appropriateness of the type and arguments is determined by side-conditions which are instantiated by the meta-logic when we make use of the schematic assumptions.

Our data-refinement rules are novel in changing the data-refinement relation within nested local blocks or procedures. This allows us to perform a data-refinement top-down in one fell swoop. Earlier presentations of data-refinement would have us repeatedly perform separate data-refinements on nested blocks, and/or compose the results in a bottom-up fashion [78, 76, 14, 68].

Our non-trivial case study demonstrated that our refinement theory is applicable to the refinement and data-refinement of a non-trivial program involving recursive procedures.

9.1 Future Work

This dissertation opens up many opportunities for further work. There is scope to investigate our inference environment, to extend our refinement language, to modify our representation of states, and to work on strengthening the link between our refinement language and real world programming languages.

Flexible window inference as presented in this dissertation is a general framework for transformational reasoning. It admits arbitrary composable relations, and allows window opening at multiple points. Further investigation is required to determine a set of constraints or modes of use which will make this general framework into a readily usable interactive inference tool. Work by von Wright [103] addresses this issue.

The procedural refinement and data-refinement rules we have presented take advantage of state-dependent contextual information when it is explicitly carried in the pre-condition of specification statements. However, a more general approach to using contextual information would be to use Nickson and Hayes' program window inference [82, 83]. This raises a general problem for the LCF theorem proving community, as this kind of 'modal' context is not currently well supported by Isabelle or other LCF theorem provers.

The refinement language presented in this thesis made use of the expressiveness of untyped set theory to represent statements not seen in earlier similar mechanisations. However, there are other constructs seen in the refinement literature: we have not considered multiply guarded alternation or looping statements, have not given a complete rule for data-refinement, and have not included frames for adding and removing program variables during data-refinement. Adding these statements and backward data-refinement in our setting should be possible. Nonetheless, there are more exotic constructs which would require us to change our semantic setting. These include trace-invariants [69, 97] exception statements and loops with multiple exits [8, 91, 55], constructs for parallelism [10, 18], probabilistic refinement [74], and real timed refinement [33, 46]. We could also change our semantics for expressions to allow side-effects. This has not been examined closely in the refinement literature, but work has been done in this area for program verification [19, 17, 84].

The procedures described in this thesis do not admit global variables: we chaotically set the local state before applying the initialisation part of the parameterisation. It would not be enough to chaotically set just the formal parameters, because a procedure is well-typed only at its declaration type, and can't depend on its calling type. We could remedy this situation by attaching a list of used global variables to our procedure declarations, similar to that used in proof rules given for the verification of procedures. [39, 30, 62] It would then be possible to establish a uniform declaration state respecting the type of these global variables. However, the situation then is still not ideal: the representation of our state as a simple mapping from variables to values would mean that the 'global' variables used in our procedure implementations would in effect be dynamically bound to the most local variable with that name. To properly model global variables, we should use a more sophisticated representation of states, say as pair of mappings: one from variables to locations and another from locations to values. This state type would allow us to more accurately model global variables, aliasing, pointers, and array indexing. The construction of a refinement language using this state type should be able to re-use all of the general set transformer work presented here in Chapter 3.

These considerations lead us to a more general question: how should we deal with the correspondence between our formal model of program development and actual programming languages in the real world? Working at the formal level provides us with a nice separation of concerns from the particular limitations of concrete compilers, operating systems and hardware. However, it could be argued that our refinement language is so abstract that there is no clear link between our developments and programs in the real world. The standard answer to this question in the refinement community is to take a 'leap of faith', and transliterate concrete refinements into programs in some actual programming language. A more principled way of doing this would be to transform our concrete refinement to another formally defined programming language whose semantics can be used by a programmer as the main reference in the construction of a compiler for that language. This could be achieved by using a structured operational semantics as our language specification, or else by investigating operational interpretations of weakest precondition semantics. In going from the formal to the real, there will always be some extra-logical leap, but it should be possible to demonstrate that this is valid.

Appendix A

Notation

We summarise the formal notation used in this thesis. Where appropriate we provide Isabelle's corresponding ASCII syntax. For notation which is identical in the thesis and in the mechanisation, we list Isabelle's syntax as $\sim$.

A.1 Isabelle's Meta-Logic

$A \rightleftharpoons B$		Macro abbreviation.
$A \mathrel{\widehat{=}} B$	`A == B`	Meta-level equality.
$A \Longrightarrow B$	`A ==> B`	Meta-level implication.
$F(x)$	`F(x)`	Meta-level function application.
$\lambda x.\, F(x)$	`%x. F(x)`	Meta-level lambda abstraction.
$\bigwedge x.\, P(x)$	`!!x. P(x)`	Meta-level universal quantification.

A.2 Isabelle/ZF

All of the operators listed here are standard in Isabelle/ZF, with the exception of domain restriction, domain subtraction, and function overriding.

$\mathsf{if}(P, A, B)$	$\sim$	Conditional expression.
$\forall x.\, P(x)$	`ALL x. P(x)`	Unbounded universal quantification.
$\exists x.\, P(x)$	`EX x. P(x)`	Unbounded universal quantification.
$\forall x : A.\, P(x)$	`ALL x:A. P(x)`	Bounded universal quantification.
$\exists x : A.\, P(x)$	`EX x:A. P(x)`	Bounded universal quantification.
$\bigcap_{x:A}.F(x)$	`INT x:A. F(x)`	Indexed intersection of family of sets.
$\bigcup_{x:A}.F(x)$	`UN x:A. F(x)`	Indexed intersection of family of sets.

$\mathbb{P}\,A$	`Pow(A)`	Power set of set A.
$\{x : A \mid P(x)\}$	`{x:A. P(x)}`	Set comprehension.
$\{F(x).\, x : A\}$	`{F(x). x:A}`	Image of meta function F on A.
$\{x.\, y : A \mid P(x,y)\}$	`{x. y:A, P(x,y)}`	Replacement of x for y s.t. $P(x,y)$.
$\langle a, b\rangle$	$\sim$	Ordered pairing.
$\Pi_X\, Y$	`Pi(X,Y)`	Dependent function space.
$\Pi_{x:X}.\, Y(x)$	`PROD x:X. Y(x)`	Syntax for above.
$f`a$	$\sim$	Function application.
$f``a$	$\sim$	Relation image.
$\lambda\, x : A.\, F(x)$	`lam x:A. F(x)`	Lambda abstraction.
$\mathrm{dom}(f)$	`domain(f)`	Domain of function.
f dres g	`f domres A`	Domain restriction of f with A.
f dsub g	`f domsub A`	Domain subtraction of f with A.
$f \oplus g$	`f fovr g`	Function overriding of f with g.
$\mathrm{wf}_A\,(R)$	`wf[A](R)`	Relation R is well-founded on A.
$\mathrm{lfp}_A(F)$	`lfp(A,F)`	Least-fixed point of F on A.

A.3 Set Transformer Syntax

Below is the syntax used for the statements and auxilliary operators described in Chapters 3 and 5.

Skip_A	$\sim$	Skip.
$a; b$	`a Seq b`	Sequential composition.
Abort_A	$\sim$	Abort.
Magic_A	$\sim$	Magic.
Chaos_A	$\sim$	Chaos.
$\mathsf{Choose}(w)_A$	$\sim$	Chaotic variable choice.
$\{P\}_A$	`Assert(P,A)`	Assertion.
$(P)_A{\rightarrow}$	`Guard(P,A)`	Guarding.
$[P]_A$	`Nondass(P,A)`	Non-deterministic assignment.
${}^{\perp}[P,\ Q]_A$	`Spec(P,Q,A)`	Fixed specification.
${}^{\top}[P,\ Q]_A$	`Sspec(P,Q,A)`	Free specification.
$w : [P,\ \ Q]_A$	`Fspec(w,P,Q,A)`	Framed specification.
$\{R\}_A$	`Rassert(R,A)`	Relational assertion.
$o =_A i.\, R(i,o)$	`o =(A) i. R(i,o)`	Syntax for above.
$[R]_A$	`Rnondass(R,A)`	Relational non-deterministic assignment.
$o :=_A i.\, R(i,o)$	`o :=(A) i. R(i,o)`	Syntax for above.
$\langle F\rangle_A$	`Fassign(F,A)`	State-assignment.

$v :=_A E$	`Assign(v,E,A)`	Single-variable assignment.
$\langle \overrightarrow{M} \rangle_A$	`Massign(M,A)`	Multiple-variable assignment.
$a \sqcup b$	`a Ach b`	Binary angelic choice.
$a \sqcap b$	`a Dch b`	Binary demonic choice.
if g then a else b fi	`Iffi(g,a,b)`	Alternation.
$\bigsqcup_A C$	`Achoice(C,A)`	Generalised angelic choice.
$\bigsqcap_A C$	`Dchoice(C,A)`	Generalised demonic choice.
re$_A$ $N.$ $F(N)$ er	`Mmu(%N. F(N),A)`	Recursion.
while g do c od	`Do(g,c)`	While-do looping.
con$_A$ $x.$ $c(x)$	`con(A) x. c(x)`	Logical constant.
var$_A$ $x.$ $c(x)$	`var(A) x. c(x)`	Logical variable.
con $x{:}X.$ $c(x)$	$\sim$	Bounded logical constant.
var $x{:}X.$ $c(x)$	$\sim$	Bounded logical variable.
begin$_A$ $v.$ c end	`Block(v,c,A)`	Single-variable block.
begin$_A$ $\overrightarrow{D}.$ c end	`Mblock(D,c,A)`	Multiple-variable block.
Param$_A(c, I, F)$	$\sim$	Parameterisation.
$\mathrm{Dom}(C)$	`Domain(C)`	Domain of family of set functions.
$\mathcal{P}_{A,B}$	`hptrans(A,B)`	Hetrogenous set transformers.
$\mathcal{P}_A$	`ptrans(A)`	Homogenous set transformers.
$\mathcal{M}_{A,B}$	`hmtrans(A,B)`	Hetrogenous monotonic set transformers.
$\mathcal{M}_A$	`mtrans(A)`	Homogenous monotonic set transformers.
monotonic(c)	$\sim$	Subset-monotonicity of set transformer c.
strict(c)	$\sim$	Strictness ('feasibility') of set transformer c.
terminating(c)	$\sim$	Universal termination of set transformer c.
monotype$_A(F)$	$\sim$	Typing for meta function F.
pmonotonic$_A(F)$	$\sim$	Refinement-monotonicity for meta function F.
regular$_A(F)$	$\sim$	Typing and refinement-monotonicity.
$a \sqsubseteq b$	`a refs b`	Refinement.
$\{P\}\ c\ \{Q\}$	`Correct(P,c,Q)`	Total correctness.

The syntax for interfaced recursion is as follows.
re$_A$ $x{:}T, N \sqsupseteq c(x).$ $d(x,N)$ er `Bmu(T, %x. c(x), %x N. d(x,N))`

A.4 Lifted Set-Transformer Syntax

Following is the syntax used for the statements and auxilliary operators described in Chapter 4.

$\boldsymbol{\mathcal{M}}_A$ `MTRANS(A)` Monotonic predicate transformer.

$a \sqsubseteq_A b$	`a REFS(A) b`	Refinement.
Skip	`SKIP`	Skip.
$a; b$	`a SEQ b`	Sequential composition.
Abort	`ABORT`	Abort.
Magic	`MAGIC`	Magic.
Chaos	`CHAOS`	Chaos.
$\{P\}$	`ASSERT(P)`	Assertion.
$(P)\rightarrow$	`GUARD(P,A)`	Guarding.
$[P]$	`NONDASS(P)`	Non-deterministic assignment.
$[P,\ Q]$	`SSPEC(P,Q)`	Free specification.
$\{R\}$	`RASSERT(R)`	Relational assertion.
$o = i.\, R(i,o)$	`o = i. R(i,o)`	Syntax for above.
$[R]$	`RNONDASS(R)`	Relational non-deterministic assignment.
$o := i.\, R(i,o)$	`o := i. R(i,o)`	Syntax for above.
$\langle F \rangle$	`FASSIGN(F)`	State-assignment.
$v := E$	`ASSIGN(v,E)`	Single-variable assignment.
$\langle \overrightarrow{M} \rangle$	`MASSIGN(M)`	Multiple-variable assignment.
$a \sqcup b$	`a ACH b`	Binary angelic choice.
$a \sqcap b$	`a DCH b`	Binary demonic choice.
if g then a else b fi	`IFFI(g,a,b)`	Alternation.
re $N.\ F(N)$ er	`MMU(%N. F(N))`	Recursion.
while g do c od	`WHILE g DO c`	While-do looping.
con $x.\ c(x)$	`CON x. c(x)`	Logical constant.
var $x.\ c(x)$	`VAR x. c(x)`	Logical variable.
con $x{:}X.\ c(x)$	`CON x:X. c(x)`	Bounded logical constant.
var $x{:}X.\ c(x)$	`VAR x:X. c(x)`	Bounded logical variable.

The syntax for interfaced recursion is as follows.
re $x:T, N \sqsupseteq c(x).\ d(x,N)$ er `RE x:T, N SFER c(x). d(x,N) ER`

A.5 Typing-Lifted Set-Transformer Syntax

Following is the syntax used for the statements and auxilliary operators described in Chapter 6. All Isabelle-syntax statements are all caps, but otherwise are often equivalent to the notation used in this dissertation.

$\mathcal{V}$	`VId`	Variable names.
$\mathcal{S}_\tau$	`VState(tau)`	State type of typing.
$\tau[T/v]$	`tau[T//v]`	Typing substitution.

$T \boxplus S$	`T TOVR S`	Typing overriding.
$a \sqsubseteq_\tau b$	`a REFS(T) b`	Refinement.
$\boldsymbol{\mathcal{M}}_\tau$	`MTRANS(T)`	Monotonic predicate transformer.
$P : \boldsymbol{\mathcal{M}}$	`(!!S a. P(a):MTRANS(S))`	Mtrans procedure call.
Skip	$\sim$	Skip.
$a; b$	`a SEQ b`	Sequential composition.
Abort	$\sim$	Abort.
Magic	$\sim$	Magic.
Chaos	$\sim$	Chaos.
Choose(w)	$\sim$	Chaotic variable choice.
$\{P\}$	`ASSERT(P)`	Assertion.
$(P)\rightarrow$	`GUARD(P)`	Guarding.
$[P]$	`NONDASS(P)`	Non-deterministic assignment.
$[P,\ Q]$	`SSPEC(P,Q)`	Free specification.
$w : [P,\ \ Q]$	`FSPEC(w,P,Q)`	Framed specification.
$\{R\}$	`RASSERT(R)`	Relational assertion.
$[R]$	`RNONDASS(R)`	Relational non-deterministic assign.
$\langle F \rangle$	`FASSIGN(F)`	State-assignment.
$v := E$	`ASSIGN(v,E)`	Single-variable assignment.
$\langle \overrightarrow{M} \rangle$	`MASSIGN(M)`	Multiple-variable assignment.
$a \sqcup b$	`a ACH b`	Binary angelic choice.
$a \sqcap b$	`a DCH b`	Binary demonic choice.
if g then a else b fi	$\sim$	Alternation.
re $N.\ F(N)$ er	`MMU(%N. F(N))`	Recursion.
while g do c od	$\sim$	While-do looping.
con $x.\ c(x)$	$\sim$	Logical constant.
var $x.\ c(x)$	$\sim$	Logical variable.
con $x{:}X.\ c(x)$	$\sim$	Bounded logical constant.
var $x{:}X.\ c(x)$	$\sim$	Bounded logical variable.
begin $v : T.\ c$ end	$\sim$	Single-variable block.
begin $D.\ c$ end	$\sim$	Multiple-variable block.
PARAM(c, S, I, F)	$\sim$	Parameterisation.

The syntax for interfaced procedures, and recursive procedures is as follows.

proc $P(D)$ int $= I$ imp $= B$ in $C(P)$

```
PROC P(D) INT === I IMP === B IN C(P)
```

rec $P(D)$ int $= I$ var $= x : T, V(x)$ imp $= B(x, P)$ in $C(P)$

```
REC P(D) INT === I VAR === x:T, V(x) IMP === B(x,P) IN C(P)
```

A.6 Case Study

All of the operators listed here are standard in Isabelle/ZF, with the exception of domain restriction, domain subtraction, and function overriding.

$\mathbb{V}$	`variable`	Variables allowed in a proposition.
prop	$\sim$	A language of propositions.
VAR(v)	$\sim$	A constructor for propositions.
NOT(p)	$\sim$	A constructor for propositions.
a AND b	$\sim$	A constructor for propositions.
is_Var(p)	`is_VAR(p)`	A discriminator for propositions.
is_Not(p)	`is_NOT(p)`	A discriminator for propositions.
is_And(p)	`is_AND(p)`	A discriminator for propositions.
VarV(p)	`VAR_var(p)`	A destructor for propositions.
NotA(p)	`NOT_arg(p)`	A destructor for propositions.
AndL(p)	`AND_left(p)`	A destructor for propositions.
AndR(p)	`AND_right(p)`	A destructor for propositions.
$[\![p]\!]_V$	`mprop(p,V)`	The meaning of a proposition.
tree	$\sim$	Boolean-leaved binary trees.
Leaf(b)	`TreeLeaf`	A constructor for trees.
Node(v, l, r)	`TreeNode(v,l,r)`	A constructor for trees.
is_Leaf(t)	`is_TreeLeaf(t)`	A discriminator for trees.
is_Node(t)	`is_TreeNode(t)`	A discriminator for trees.
LeafT(t)	`TreeLeaf_tag(t)`	A destructor for trees.
NodeV(t)	`TreeNode_var(t)`	A destructor for trees.
NodeL(t)	`TreeNode_left(t)`	A destructor for trees.
NodeR(t)	`TreeNode_right(t)`	A destructor for trees.
$\{\!\mid p \mid\!\}_V$	`mtree(p,V)`	The meaning of a tree.
tleaves(t)	$\sim$	A tree has all true leaves.
tvars(t)	`treevars(t)`	The set of variables in a tree.
negtree(t)	$\sim$	Negating all the leaves of a tree.
dtree	$\sim$	Decision trees.
rtree	$\sim$	Reduced trees.
otree	$\sim$	Ordered trees.
rotree	$\sim$	Reduced ordered trees.
Ntree	$\sim$	Proto-negation trees.
NLeaf	`NTreeLeaf`	A constructor for proto-negation trees.
NNode(v, l, p, r)	`NTreeNode(v,l,p,r)`	A constructor for proto-negation trees.
is_NLeaf(t)	`is_NTreeLeaf(t)`	A discriminator for proto-negation trees.
is_NNode(t)	`is_NTreeNode(t)`	A discriminator for proto-negation trees.
NNodeV(t)	`NTreeNode_var(t)`	A destructor for proto-negation trees.
NNodeL(t)	`NTreeNode_left(t)`	A destructor for proto-negation trees.

NNodeN(t)	`NTreeNode_rneg(t)`	A destructor for proto-negation trees.
NNodeR(t)	`NTreeNode_right(t)`	A destructor for proto-negation trees.
tree_of_Ntree	$\sim$	The tree got from a proto-negation tree.
dNtree	$\sim$	Proto-negation decision trees.
roNtree	$\sim$	Reduced ordered proto-negation trees.
ntree	$\sim$	Negation trees.
is_nleaf(t)	`is_ntreeleaf(t)`	A 'discriminator' for negation trees.
is_nnode(t)	`is_ntreenode(t)`	A 'discriminator' for negation trees.
nneg(t)	`ntree_neg`	A 'destructor' for negation trees.
nvar(t)	`ntree_var`	A 'destructor' for negation trees.
nrneg(t)	`ntree_rneg`	A 'destructor' for negation trees.
nleft(t)	`ntree_left`	A 'destructor' for negation trees.
nright(t)	`ntree_right`	A 'destructor' for negation trees.
tree_of_ntree(t)	$\sim$	The tree got from a negation tree.
dntree	$\sim$	Negation decision trees.
rontree	$\sim$	Reduced ordered negation trees.

Appendix B

Definitions

B.1 Set Transformers

Here we list the definitions of statements and auxilliary operators described in Chapters 3 and 5. For statements defined in terms of the domain of one of their sub-components, we also list the simplified equivalent forms which assume that the sub-components are well-typed.

B.1.1 Atomic Statements

$$\mathsf{Skip}_A \mathrel{\widehat{=}} \lambda\, q{:}\mathbb{P}\, A.\, q \qquad \mathsf{Magic}_A \mathrel{\widehat{=}} \lambda\, q{:}\mathbb{P}\, A.\, A$$
$$\mathsf{Chaos}_A \mathrel{\widehat{=}} \lambda\, q{:}\mathbb{P}\, A.\, \mathsf{if}(A = q, A, \varnothing) \qquad \mathsf{Abort}_A \mathrel{\widehat{=}} \lambda\, q{:}\mathbb{P}\, A.\, \varnothing$$

$$\begin{aligned}
\mathsf{Choose}(w)_A &\mathrel{\widehat{=}} \lambda\, q{:}\mathbb{P}(A).\, \{i{:}A \mid \{o{:}A \mid i \operatorname{dsub} w = o \operatorname{dsub} w\} \subseteq q\} \\
\{P\}_A &\mathrel{\widehat{=}} \lambda\, q{:}\mathbb{P}\, A.\, P \cap q \\
(Q)_A{\rightarrow} &\mathrel{\widehat{=}} \lambda\, q{:}\mathbb{P}\, A.\, (A - Q) \cup q \\
[Q]_A &\mathrel{\widehat{=}} \lambda\, q{:}\mathbb{P}\, A.\, \mathsf{if}(Q \subseteq q, A, \varnothing) \\
{}^{\perp}[P,\ Q]_A &\mathrel{\widehat{=}} \lambda\, q{:}\mathbb{P}\, A.\, P \cap ((A - Q) \cup q) \\
{}^{\top}[P,\ Q]_A &\mathrel{\widehat{=}} \lambda\, q{:}\mathbb{P}\, A.\, \mathsf{if}(Q \subseteq q, P, \varnothing) \\
w : [P,\ \ Q]_A &\mathrel{\widehat{=}} \lambda\, q{:}\mathbb{P}(A).\, \{i{:}A \mid \ i \in P \wedge \\
&\qquad\qquad \{o{:}A \mid \ o \in Q \wedge i \operatorname{dsub} w = o \operatorname{dsub} w\} \subseteq q\} \\
\{R\}_A &\mathrel{\widehat{=}} \lambda\, q{:}\mathbb{P}\, A.\, \{s{:}A \mid R\text{“}\{s\} \cap q \neq \varnothing\} \\
[R]_A &\mathrel{\widehat{=}} \lambda\, q{:}\mathbb{P}\, A.\, \{s{:}A \mid R\text{“}\{s\} \subseteq q\}
\end{aligned}$$

We use the following sugared syntax for relational assertion and nondeterministic assignment when the relation is constructed by set comprehension.

$$\begin{array}{rcl} o =_A i.\, R(i,o) & \rightleftharpoons & \{\!\{\{\langle i,o\rangle : A \times A \mid R(i,o)\}\}\!\}_A \\ o :=_A i.\, R(i,o) & \rightleftharpoons & [\{\langle i,o\rangle : A \times A \mid R(i,o)\}]_A \end{array}$$

With these abbrevations, we have the following equivalences.

$$\begin{array}{rcl} o =_A i.\, R(i,o) & = & \lambda\, q{:}\mathbb{P}\, A.\, \{i{:}A \mid \{o{:}A \mid R(i,o)\} \cap q \neq \varnothing\} \\ o :=_A i.\, R(i,o) & = & \lambda\, q{:}\mathbb{P}\, A.\, \{i{:}A \mid \{o{:}A \mid R(i,o)\} \subseteq q\} \end{array}$$

The various forms of assignment are defined as follows.

$$\begin{array}{rcl} \langle F \rangle_A & \mathrel{\widehat{=}} & \lambda\, q{:}\mathbb{P}\, A.\, \{s{:}\mathrm{dom}(F) \mid F\text{`}s \in q\} \\ \langle \overrightarrow{M} \rangle_A & \mathrel{\widehat{=}} & \lambda\, q{:}\mathbb{P}(A).\, \{s{:}\mathrm{dom}(M) \mid s \oplus F\text{`}s \in q\} \\ v :=_A E & \mathrel{\widehat{=}} & \lambda\, q{:}\mathbb{P}(A).\, \{s{:}\mathrm{dom}(E) \mid s[{}^{E\text{`}s}/_v] \in q\} \end{array}$$

When $F, E, M : B \to X$, we have the following equivalences for assignment statements.

$$\begin{array}{rcl} \langle F \rangle_A & = & \lambda\, q{:}\mathbb{P}\, A.\, \{s{:}B \mid F\text{`}s \in q\} \\ v :=_A E & = & \lambda\, q{:}\mathbb{P}(A).\, \{s{:}B \mid s[{}^{E\text{`}s}/_v] \in q\} \\ \langle \overrightarrow{M} \rangle_A & = & \lambda\, q{:}\mathbb{P}(A).\, \{s{:}B \mid s \oplus F\text{`}s \in q\} \end{array}$$

B.1.2 Compound Statements

$$\begin{array}{rcl} a; b & \mathrel{\widehat{=}} & \lambda\, q{:}\mathrm{dom}(b).\, a\text{`}(b\text{`}q) \\ \mathsf{if}\ g\ \mathsf{then}\ b\ \mathsf{else}\ c\ \mathsf{fi} & \mathrel{\widehat{=}} & \lambda\, q{:}\mathrm{dom}(b) \cup \mathrm{dom}(c). \\ & & \quad (g \cap b\text{`}q) \cup ((\bigcup(\mathrm{dom}(b) \cup \mathrm{dom}(c)) - g) \cap c\text{`}q) \\ a \sqcup b & \mathrel{\widehat{=}} & \lambda\, q{:}\mathrm{dom}(b) \cup \mathrm{dom}(c).\, b\text{`}q \cup c\text{`}q \\ a \sqcap b & \mathrel{\widehat{=}} & \lambda\, q{:}\mathrm{dom}(b) \cup \mathrm{dom}(c).\, b\text{`}q \cap c\text{`}q \\ \bigsqcup_A C & \mathrel{\widehat{=}} & \lambda\, q{:}\mathbb{P}\, A.\, \bigcup_{c:C} .c\text{`}q \\ \bigsqcap_A C & \mathrel{\widehat{=}} & \lambda\, q{:}\mathbb{P}\, A.\, \bigcap_{c:C} .c\text{`}q \\ \mathsf{con}_A\ x{:}T.\ c(x) & \mathrel{\widehat{=}} & \lambda\, q{:}\mathbb{P}\, A.\, \bigcup_{e:T} .c(e)\text{`}q \end{array}$$

$$\mathsf{var}_A\ x:T.\ c(x) \mathrel{\widehat{=}} \lambda\, q{:}\mathbb{P}\, A.\, \bigcap e : T.c(e)`q$$

$$\mathsf{con}\ x.\ c(x) \mathrel{\widehat{=}} \lambda\, q{:}\mathrm{Dom}(c).\, \{s{:}\bigcup \mathrm{Dom}(c) \mid \exists\, v.\, s \in c(v)`q\}$$

$$\mathsf{var}\ x.\ c(x) \mathrel{\widehat{=}} \lambda\, q{:}\mathrm{Dom}(c).\, \{s{:}\bigcup \mathrm{Dom}(c) \mid \forall\, v.\, s \in c(v)`q\}$$

$$\mathsf{re}_A\ X.\ F(X)\ \mathsf{er} \mathrel{\widehat{=}} \bigsqcap\nolimits_A \{c{:}\mathcal{M}_A \mid F(c) \sqsubseteq c\}$$

$$\mathsf{while}\ g\ \mathsf{do}\ c\ \mathsf{od} \mathrel{\widehat{=}} \mathsf{re}_{\bigcup \mathrm{dom}(c)}\ X.\ \mathsf{if}\ g\ \mathsf{then}\ c; X\ \mathsf{else}\ \mathsf{Skip}_{\bigcup \mathrm{dom}(c)}\ \mathsf{fi}\ \mathsf{er}$$

$$\begin{aligned}\mathsf{begin}_A\ v.\ c\ \mathsf{end} \mathrel{\widehat{=}} \mathsf{store}_A\ s.\ & \mathsf{Choose}(\{v\})\bigcup(\mathrm{dom}(c)), A;\\ & c; \langle \lambda\, x : \bigcup(\mathrm{dom}(c)).\, x[s`v/v]\rangle_A\end{aligned}$$

$$\begin{aligned}\mathsf{begin}_A\ \overrightarrow{w}.\ c\ \mathsf{end} \mathrel{\widehat{=}} \mathsf{store}_A\ s.\ & \mathsf{Choose}(v)\bigcup(\mathrm{dom}(c)), A;\\ & c; \langle \lambda\, x : \bigcup(\mathrm{dom}(c)).\, x \oplus (s\ \mathrm{dres}\ w)\rangle_A\end{aligned}$$

$$\begin{aligned}\mathsf{Param}_A(c, I, F) \mathrel{\widehat{=}} \mathsf{store}_A\ i.\ & \mathsf{Chaos}_{\bigcup(\mathrm{dom}(c)),A};\\ & \langle \lambda\, s : \bigcup(\mathrm{dom}(c)).\, s \oplus I(i)\rangle_{\bigcup(\mathrm{dom}(c))};\\ & c;\\ & \langle \lambda\, s : \bigcup(\mathrm{dom}(c)).\, i \oplus F(s)\rangle_A\end{aligned}$$

Interfaced recursion is defined in terms of raw recursion and assertion statements.

$$\begin{aligned}&\mathsf{re}_A\ x:T, N \sqsupseteq I(x).\ c(x,N)\ \mathsf{er} \mathrel{\widehat{=}}\\ &\quad \mathsf{re}_A\ N.\ \mathsf{con}_A\ x:T.\ \{\!\{\_{:}A \mid I(x) \sqsubseteq N\}\!\}_A; c(x,N)\ \mathsf{er}\end{aligned}$$

We can prove various simpler equivalences for compound statements, as follows For $b, c : \mathcal{P}_{A,B}$ we have:

$$\mathsf{if}\ g\ \mathsf{then}\ b\ \mathsf{else}\ c\ \mathsf{fi} = \lambda\, q{:}\mathbb{P}\, A.\, (g \cap b`q) \cup ((A - g) \cap c`q)$$

$$b \sqcup c = \lambda\, q{:}\mathbb{P}\, A.\, b`q \cup c`q$$

$$b \sqcap c = \lambda\, q{:}\mathbb{P}\, A.\, b`q \cap c`q$$

$$a; b = \lambda\, q{:}\mathbb{P}\, A.\, a`(b`q)$$

$$\mathsf{while}\ g\ \mathsf{do}\ c\ \mathsf{od} = \mathsf{re}_A\ X.\ \mathsf{if}\ g\ \mathsf{then}\ c; X\ \mathsf{else}\ \mathsf{Skip}_A\ \mathsf{fi}\ \mathsf{er}$$

If $c(e) : \mathcal{P}_{A,B}$ for all e, then $\mathrm{Dom}(c) = \mathbb{P}\, A$ and so we have:

$$\mathsf{con}\ x.\ c(x) = \lambda\, q{:}\mathbb{P}\, A.\, \{s{:}A \mid \exists\, v.\, s \in c(v)`q\}$$

$$\mathsf{var}\ x.\ c(x) = \lambda\, q{:}\mathbb{P}\, A.\, \{s{:}A \mid \forall\, v.\, s \in c(v)`q\}$$

And when $c : \mathcal{P}_B$, we have the following equivalences.

$$\begin{aligned}
\mathsf{begin}_A\ v.\ c\ \mathsf{end} &= \mathsf{store}_A\ s.\ \mathsf{Choose}(\{v\})B, A; c; \langle \lambda\, x : B.\, x[s`v/v] \rangle_A \\
\mathsf{begin}_A\ \overrightarrow{w}.\ c\ \mathsf{end} &= \mathsf{store}_A\ s.\ \mathsf{Choose}(v)B, A; c; \langle \lambda\, x : B.\, x \oplus (s\ \mathrm{dres}\ w) \rangle_A \\
\mathsf{Param}_A(c, I, F) &= \mathsf{store}_A\ i.\ \mathsf{Chaos}_{B,A}; \langle \lambda\, s : B.\, s \oplus I(i) \rangle_B; c; \langle \lambda\, s : B.\, i \oplus F(s) \rangle_A
\end{aligned}$$

B.1.3 Auxilliary Operators

$$\begin{aligned}
\mathcal{P}_{A,B} &\mathrel{\widehat{=}} \mathbb{P}\, A \to \mathbb{P}\, B \\
\mathcal{P}_A &\rightleftharpoons \mathcal{P}_{A,A} \\
\mathsf{monotonic}(c) &\mathrel{\widehat{=}} \forall\, a\ b : \mathrm{dom}(c).\, a \subseteq b \Rightarrow c`a \subseteq c`b \\
\mathcal{M}_{A,B} &\mathrel{\widehat{=}} \{c : \mathcal{P}_{A,B} \mid \mathsf{monotonic}(c)\} \\
\mathcal{M}_A &\rightleftharpoons \mathcal{M}_{A,A} \\
\mathsf{strict}(c) &\mathrel{\widehat{=}} c`\varnothing = \varnothing \\
\mathsf{terminating} c &\mathrel{\widehat{=}} c`(\bigcup \mathrm{dom}(c)) = \bigcup \mathrm{dom}(c) \\
\mathsf{monotype}_A(f) &\mathrel{\widehat{=}} \forall\, c : A.\, F(c) : A \\
\mathsf{pmonotonic}_A(f) &\mathrel{\widehat{=}} \forall\, a : \mathcal{M}_A\ b : \mathcal{M}_A.\, a \sqsubseteq b \Rightarrow F(a) \sqsubseteq F(b) \\
\mathsf{regular}_A(f) &\mathrel{\widehat{=}} \mathsf{pmonotonic}_A(F) \wedge \mathsf{monotype}_A(f) \\
\mathrm{Dom}(c) &\mathrel{\widehat{=}} \epsilon\, d.\ \forall\, v.\, d = \mathrm{dom}(c(v))
\end{aligned}$$

B.2 Lifted Set-Transformers

Here we list the definitions of statements and auxilliary operators described in Chapter 4.

$$\begin{aligned}
\mathsf{Skip} &\mathrel{\widehat{=}} \lambda\, A.\, \mathsf{Skip}_A \\
\mathsf{Abort} &\mathrel{\widehat{=}} \lambda\, A.\, \mathsf{Abort}_A \\
\mathsf{Magic} &\mathrel{\widehat{=}} \lambda\, A.\, \mathsf{Magic}_A \\
\mathsf{Chaos} &\mathrel{\widehat{=}} \lambda\, A.\, \mathsf{Chaos}_{A,A} \\
\{P\} &\mathrel{\widehat{=}} \lambda\, A.\, \{\!\{\{s : A \mid P(s)\}\}\!\}_A \\
(Q)\!\to &\mathrel{\widehat{=}} \lambda\, A.\, (\{s : A \mid Q(s)\})_A \to \\
[Q] &\mathrel{\widehat{=}} \lambda\, A.\, [\{s : A \mid Q(s)\}]_A \\
[P,\ Q] &\mathrel{\widehat{=}} \lambda\, A.\, {}^{\top}[\{s : A \mid P(s)\},\ \{s : A \mid Q(s)\}]_A \\
\langle F \rangle &\mathrel{\widehat{=}} \lambda\, A.\, \langle \lambda\, x : A.\, F(x) \rangle_A
\end{aligned}$$

$$
\begin{array}{rcl}
o = i.\, R(i,o) & \widehat{=} & \lambda\, A.\, o =_A i.\, R(i,o) \\
o := i.\, R(i,o) & \widehat{=} & \lambda\, A.\, o :=_A i.\, R(i,o) \\
a; b & \widehat{=} & \lambda\, A.\, a(A); b(A) \\
a \sqcup b & \widehat{=} & \lambda\, A.\, a(A) \sqcup b(A) \\
a \sqcap b & \widehat{=} & \lambda\, A.\, a(A) \sqcap b(A) \\
\mathsf{if}\ g\ \mathsf{then}\ a\ \mathsf{else}\ b\ \mathsf{fi} & \widehat{=} & \lambda\, A.\, \mathsf{if}\ \{x{:}A \mid g(x)\}\ \mathsf{then}\ a(A)\ \mathsf{else}\ b(A)\ \mathsf{fi} \\
\mathsf{while}\ g\ \mathsf{do}\ c\ \mathsf{od} & \widehat{=} & \lambda\, A.\, \mathsf{while}\ \{x{:}A \mid g(x)\}\ \mathsf{do}\ c(A)\ \mathsf{od} \\
\mathsf{re}\ N.\ c(N)\ \mathsf{er} & \widehat{=} & \lambda\, A.\, \mathsf{re}_A\ N.\ c(\lambda\_.\, N, A)\ \mathsf{er} \\
\mathsf{re}\ x{:}T, N \sqsupseteq I(x).\ c(x,N)\ \mathsf{er} & \widehat{=} & \lambda\, S.\, \mathsf{re}\ x{:}[, \mathcal{S}_\tau \sqsupseteq].\ T\ \mathsf{er} N I(x) c(x, \lambda\_.\, N) \\
\mathsf{con}\ x.\ c(x) & \widehat{=} & \lambda\, A.\, \mathsf{con}\ x.\ c(x, A) \\
\mathsf{var}\ x.\ c(x) & \widehat{=} & \lambda\, A.\, \mathsf{var}\ x.\ c(x, A) \\
\mathsf{con}\ x{:}T.\ c(x) & \widehat{=} & \lambda\, A.\, \mathsf{con}_A\ x{:}T.\ c(x, A) \\
\mathsf{var}\ x{:}T.\ c(x) & \widehat{=} & \lambda\, A.\, \mathsf{var}_A\ x{:}T.\ c(x, A)
\end{array}
$$

B.3 Typing-Lifted Set-Transformers

Here we list the definitions of statements and auxilliary operators described in Chapter 6.

$$
\begin{array}{rcl}
\mathsf{Skip} & \widehat{=} & \lambda\, \tau.\, \mathsf{Skip}_{\mathcal{S}_\tau} \\
\mathsf{Abort} & \widehat{=} & \lambda\, \tau.\, \mathsf{Abort}_{\mathcal{S}_\tau} \\
\mathsf{Magic} & \widehat{=} & \lambda\, \tau.\, \mathsf{Magic}_{\mathcal{S}_\tau} \\
\mathsf{Chaos} & \widehat{=} & \lambda\, \tau.\, \mathsf{Chaos}_{\mathcal{S}_\tau, \mathcal{S}_\tau} \\
\mathsf{Choose}(w) & \widehat{=} & \lambda\, \tau.\, \mathsf{Choose}(w) \mathcal{S}_\tau \\
\{P\} & \widehat{=} & \lambda\, \tau.\, \{\{s{:}\mathcal{S}_\tau \mid P(s)\}\}_{\mathcal{S}_\tau} \\
(Q)\!\rightarrow & \widehat{=} & \lambda\, \tau.\, (\{s{:}\mathcal{S}_\tau \mid Q(s)\})_{\mathcal{S}_\tau}\!\rightarrow \\
[Q] & \widehat{=} & \lambda\, \tau.\, [\{s{:}\mathcal{S}_\tau \mid Q(s)\}]_{\mathcal{S}_\tau} \\
[P,\ Q] & \widehat{=} & \lambda\, \tau.\, {}^{\top}[\{s{:}\mathcal{S}_\tau \mid P(s)\},\ \{s{:}\mathcal{S}_\tau \mid Q(s)\}]_{\mathcal{S}_\tau} \\
v : [P,\ \ Q] & \widehat{=} & \lambda\, \tau.\, v : [\{s{:}\mathcal{S}_\tau \mid P(s)\},\ \ \{s{:}\mathcal{S}_\tau \mid Q(s)\}]_{\mathcal{S}_\tau} \\
\langle F \rangle & \widehat{=} & \lambda\, \tau.\, \langle \lambda\, x{:}\mathcal{S}_\tau.\, F(x) \rangle_{\mathcal{S}_\tau} \\
v := E & \widehat{=} & \lambda\, \tau.\, v :=_{\mathcal{S}_\tau} (\lambda\, s{:}\mathcal{S}_\tau.\, E(s)) \\
\langle \overrightarrow{F} \rangle & \widehat{=} & \lambda\, \tau.\, \langle \overrightarrow{\lambda\, x{:}\mathcal{S}_\tau.\, M(x)} \rangle_{\mathcal{S}_\tau} \\
o = i.\, R(i,o) & \widehat{=} & \lambda\, \tau.\, o =_{\mathcal{S}_\tau} i.\, R(i,o)
\end{array}
$$

$$
\begin{array}{rcl}
o := i.\, R(i,o) & \widehat{=} & \lambda\tau.\, o :=_{\mathcal{S}_\tau} i.\, R(i,o) \\
a ; b & \widehat{=} & \lambda\tau.\, a(\tau) ; b(\tau) \\
a \sqcup b & \widehat{=} & \lambda\tau.\, a(\tau) \sqcup b(\tau) \\
a \sqcap b & \widehat{=} & \lambda\tau.\, a(\tau) \sqcap b(\tau) \\
\mathsf{if}\ g\ \mathsf{then}\ a\ \mathsf{else}\ b\ \mathsf{fi} & \widehat{=} & \lambda\tau.\, \mathsf{if}\ \{x{:}\mathcal{S}_\tau \mid g(x)\}\ \mathsf{then}\ a(\tau)\ \mathsf{else}\ b(\tau)\ \mathsf{fi} \\
\mathsf{while}\ g\ \mathsf{do}\ c\ \mathsf{od} & \widehat{=} & \lambda\tau.\, \mathsf{while}\ \{x{:}\mathcal{S}_\tau \mid g(x)\}\ \mathsf{do}\ c(\tau)\ \mathsf{od} \\
\mathsf{re}\ N.\ c(N)\ \mathsf{er} & \widehat{=} & \lambda\tau.\, \mathsf{re}_{\mathcal{S}_\tau}\ N.\ c(\lambda\_.\, N, \tau)\ \mathsf{er} \\
\mathsf{re}\ x{:}T, N \sqsupseteq I(x).\ c(x,N)\ \mathsf{er} & \widehat{=} & \lambda\tau.\, \mathsf{re}_{\mathcal{S}_\tau}\ x{:}T, N \sqsupseteq I(x).\ c(x, \lambda\_.\, N)\ \mathsf{er} \\
\mathsf{con}\ x.\ c(x) & \widehat{=} & \lambda\tau.\, \mathsf{con}\ x.\ c(x,\tau) \\
\mathsf{var}\ x.\ c(x) & \widehat{=} & \lambda\tau.\, \mathsf{var}\ x.\ c(x,\tau) \\
\mathsf{con}\ x{:}T.\ c(x) & \widehat{=} & \lambda\tau.\, \mathsf{con}_{\mathcal{S}_\tau}\ x{:}T.\ c(x,\tau) \\
\mathsf{var}\ x{:}T.\ c(x) & \widehat{=} & \lambda\tau.\, \mathsf{var}_{\mathcal{S}_\tau}\ x{:}T.\ c(x,\tau) \\
\mathsf{begin}\ v : T.\ c\ \mathsf{end} & \widehat{=} & \lambda\tau.\, \mathsf{begin}_{\mathcal{S}_\tau}\ v.\ c(\tau[T/v])\ \mathsf{end} \\
\mathsf{begin}\ D.\ c\ \mathsf{end} & \widehat{=} & \lambda\tau.\, \mathsf{begin}_{\mathcal{S}_\tau}\ \overrightarrow{\mathrm{dom}(D)}.\ c(\tau \boxplus D)\ \mathsf{end}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{proc}\ P(D)\ \mathsf{int} = I\ \mathsf{imp} = B\ \mathsf{in}\ C(P) \mathrel{\widehat{=}} \\
\quad \mathbf{let}\ P = \lambda\, T\ \ a.\, \mathsf{PARAM}(B, \mathbb{T}_{D,T}, \mathbb{I}_{D,a}, \mathbb{F}_{D,a}) \\
\quad \mathbf{in}\ \lambda\, T.\, (\{\, \lambda\_.\, I\ \sqsubseteq_{\mathbb{T}_{D,T}}\ B\,\}\, ;\, C(P(T)))(T)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{rec}\ \ P(D)\ \ \mathsf{int} = I\ \mathsf{var} = v : V, R(v)\ \mathsf{imp} = B(v,P)\ \mathsf{in}\ C(P) \mathrel{\widehat{=}} \\
\quad \mathbf{let}\ P = \lambda\, T\ \ a.\, \mathsf{PARAM}(B'(T), \mathbb{T}_{D,T}, \mathbb{I}_{D,a}, \mathbb{F}_{D,a}) \\
\quad \mathbf{in}\ \lambda\, T.\, (\{\, \lambda\_.\, I\ \sqsubseteq_{\mathbb{T}_{D,T}}\ B'(T)\,\}\, ;\, C(P(T)))(T) \\
\quad \mathbf{where}\ B'(T) = \\
\qquad \mathsf{re} P.\ \mathsf{con}\ \ v{:}V\,. \\
\qquad\quad \{\, \lambda\_.\, \{\, \lambda\, s.\, R(v,s)\,\}\, ;\, I\ \sqsubseteq_{\mathbb{T}_{D,T}}\ P\,\}\, ; \\
\qquad\quad B(v, \lambda\, a.\, \mathsf{PARAM}(P, \mathbb{T}_{D,T}, \mathbb{I}_{D,a}, \mathbb{F}_{D,a})) \\
\qquad \mathsf{er}
\end{array}
$$

Appendix C

Theorems

C.1 Set Transformers

Here we list typing and refinement-monotonicity theorems for the set transformer statements described in Chapters 3 and 5.

C.1.1 Monotonic Set Transformers

$$
\begin{array}{rcl}
\mathsf{Skip}_A & : & \mathcal{M}_A \\
\mathsf{Abort}_A & : & \mathcal{M}_{A,B} \\
\mathsf{Magic}_A & : & \mathcal{M}_A \\
\mathsf{Chaos}_A & : & \mathcal{M}_A \\
\mathsf{Choose}(w)_A & : & \mathcal{M}_A
\end{array}
\qquad
\begin{array}{rcl}
\{P\}_A & : & \mathcal{M}_A \\
(Q)_A{\rightarrow} & : & \mathcal{M}_A \\
[Q]_A & : & \mathcal{M}_A \\
\langle F \rangle_A & : & \mathcal{M}_{A,B}{}^{\dagger} \\
v :=_A E & : & \mathcal{M}_{A,B}{}^{\dagger} \\
\langle \overrightarrow{M} \rangle_{A,B} & : & \mathcal{M}_{A,B}
\end{array}
\qquad
\begin{array}{rcl}
{}^{\perp}[P,\ Q]_A & : & \mathcal{M}_A \\
{}^{\top}[P,\ Q]_A & : & \mathcal{M}_A \\
w : [P,\ \ Q]_A & : & \mathcal{M}_A \\
\{R\}_A & : & \mathcal{M}_A \\
[R]_A & : & \mathcal{M}_A
\end{array}
$$

†For $E, F : B \to X$

$$\frac{a : \mathcal{M}_{B,C} \quad b : \mathcal{M}_{A,B}}{a; b : \mathcal{M}_{A,C}} \qquad \frac{a : \mathcal{M}_A \quad b : \mathcal{M}_A}{\mathsf{if}\ g\ \mathsf{then}\ a\ \mathsf{else}\ b\ \mathsf{fi} : \mathcal{M}_A}$$

$$\frac{a : \mathcal{M}_{A,B} \quad b : \mathcal{M}_{A,B}}{a \sqcup b : \mathcal{M}_{A,B}} \qquad \frac{a : \mathcal{M}_{A,B} \quad b : \mathcal{M}_{A,B}}{a \sqcap b : \mathcal{M}_{A,B}}$$

$$\frac{C : \mathbb{P}\,\mathcal{M}_{A,B}}{\bigsqcup_A C : \mathcal{M}_{A,B}} \qquad \frac{C : \mathbb{P}\,\mathcal{M}_{A,B}}{\bigsqcap_A C : \mathcal{M}_{A,B}}$$

$$\frac{\forall\, x.\ c(x) : \mathcal{M}_A}{\mathsf{con}\ x.\ c(x) : \mathcal{M}_A} \qquad \frac{\forall\, x.\ c(x) : \mathcal{M}_A}{\mathsf{var}\ x.\ c(x) : \mathcal{M}_A}$$

$$\frac{\forall\, x : T.\ c(x) : \mathcal{M}_{A,B}}{\mathsf{con}_A\ x : T.\ c(x) : \mathcal{M}_{A,B}} \qquad \frac{\forall\, x : T.\ c(x) : \mathcal{M}_{A,B}}{\mathsf{var}_A\ x : T.\ c(x) : \mathcal{M}_{A,B}}$$

$$\frac{\forall\, x.\ a(x) : \mathcal{M}_{A,B}}{\mathsf{store}_B\ x.\ a(x) : \mathcal{M}_{A,B}} \qquad \mathsf{re}_A\ N.\ F(N)\ \mathsf{er} : \mathcal{M}_A$$

$$\frac{c : \mathcal{M}_A}{\mathsf{while}\ g\ \mathsf{do}\ c\ \mathsf{od} : \mathcal{M}_A} \qquad \mathsf{re}_A\ x : T, N \sqsupseteq I(x).\ c(x, N)\ \mathsf{er} : \mathcal{M}_A$$

$$\frac{c : \mathcal{M}_B}{\mathsf{begin}_A\ v.\ c\ \mathsf{end} : \mathcal{M}_A} \qquad \frac{c : \mathcal{M}_B}{\mathsf{begin}_A\ \overrightarrow{w}.\ c\ \mathsf{end} : \mathcal{M}_A}$$

$$\frac{c : \mathcal{M}_B}{\mathsf{Param}_A(c, I, F) : \mathcal{M}_A}$$

C.1.2 Refinement Monotonicity

$$\frac{P \subseteq P' \quad P \cap Q' \subseteq Q}{{}^{\perp}[P,\ Q]A \sqsubseteq {}^{\perp}[P',\ Q']A}$$

$$\frac{P \subseteq P' \quad P \neq 0 \Rightarrow Q' \subseteq Q}{{}^{\top}[P,\ Q]A, B \sqsubseteq {}^{\top}[P',\ Q']A, B}$$

$$\frac{P \subseteq P' \qquad \begin{array}{r}\forall\, i : P.\, \{o : Q' \mid i\ \mathrm{dsub}\ v = o\ \mathrm{dsub}\ v\} \subseteq \\ \{o : Q \mid i\ \mathrm{dsub}\ v = o\ \mathrm{dsub}\ v\}\end{array}}{v : [P,\ \ Q]A, B \sqsubseteq v : [P',\ \ Q']A, B}$$

Given $a : \mathcal{M}_{B,C}$ and $c, d : \mathcal{M}_{A,B}$, we have:

$$\frac{a \sqsubseteq b \quad c \sqsubseteq d}{a; c \sqsubseteq b; d}$$

Given $a, b, c, d : \mathcal{M}_A$, we have:

$$\frac{a \sqsubseteq b \quad c \sqsubseteq d}{\mathsf{if}\ g\ \mathsf{then}\ a\ \mathsf{else}\ c\ \mathsf{fi} \sqsubseteq \mathsf{if}\ g\ \mathsf{then}\ b\ \mathsf{else}\ d\ \mathsf{fi}}$$

Given $a, b, c, d : \mathcal{M}_{A,B}$, we have:

$$\frac{a \sqsubseteq b \quad c \sqsubseteq d}{a \sqcup c \sqsubseteq b \sqcup d} \quad \frac{a \sqsubseteq b \quad c \sqsubseteq d}{a \sqcap c \sqsubseteq b \sqcap d}$$

If for any x we know $a(x), b(x) : \mathcal{M}_{A,X}$, then:

$$\frac{\forall x.\, a(x) \sqsubseteq b(x)}{\mathsf{con}\ x.\ a(x) \sqsubseteq \mathsf{con}\ x.\ b(x)} \quad \frac{\forall x.\, a(x) \sqsubseteq b(x)}{\mathsf{var}\ x.\ a(x) \sqsubseteq \mathsf{var}\ x.\ b(x)}$$

Given $a(x) : \mathcal{M}_{A,X}$ for any x in T, we have:

$$\frac{\forall x : T.\, a(x) \sqsubseteq b(x)}{\mathsf{con}_A\ x : T.\ a(x) \sqsubseteq \mathsf{con}_A\ x : T.\ b(x)} \quad \frac{\forall x : T.\, a(x) \sqsubseteq b(x)}{\mathsf{var}_A\ x : T.\ a(x) \sqsubseteq \mathsf{var}_A\ x : T.\ b(x)}$$

Given that for any s we have $a(s), b(s) : \mathcal{P}_{A,B}$, then we know:

$$\frac{\forall s : B.\, a(s) \sqsubseteq b(s)}{\mathsf{store}_B\ s.\ a(s) \sqsubseteq \mathsf{store}_B\ s.\ b(s)}$$

Given $a, b : \mathcal{M}_A$, we have:

$$\frac{a \sqsubseteq b}{\mathsf{while}\ g\ \mathsf{do}\ a\ \mathsf{od} \sqsubseteq \mathsf{while}\ g\ \mathsf{do}\ b\ \mathsf{od}}$$

Given $a(x), b(x), d(x) : \mathcal{M}_A$ for any x in $\mathcal{M}_A$, we have:

$$\frac{\forall x : \mathcal{M}_A.\, c(x) \sqsubseteq d(x)}{\mathsf{re}_A\ N.\ c(N)\,\mathsf{er} \sqsubseteq \mathsf{re}_A\ N.\ d(N)\,\mathsf{er}}$$

$$\frac{\forall x : T. \forall N : \mathcal{M}_A.\, c(x) \sqsubseteq \mathsf{K}\ N \Rightarrow a(x, N) \sqsubseteq b(x, N)}{\mathsf{re}_A\ x : T, N \sqsupseteq c(x).\ a(x, N)\,\mathsf{er} \sqsubseteq \mathsf{re}_A\ x : T, N \sqsupseteq c(x).\ b(x, N)\,\mathsf{er}}$$

Given $c, d : \mathcal{M}_B$, then

$$\frac{c \sqsubseteq d}{\mathsf{begin}_A\ v.\ c\ \mathsf{end} \sqsubseteq \mathsf{begin}_A\ v.\ d\ \mathsf{end}}$$

$$\frac{c \sqsubseteq d}{\mathsf{begin}_A\ \overrightarrow{w}.\ c\ \mathsf{end} \sqsubseteq \mathsf{begin}_A\ \overrightarrow{w}.\ d\ \mathsf{end}}$$

Given $c, d : \mathcal{M}_B$, then

$$\frac{c \sqsubseteq d}{\mathsf{Param}_A(c, I, F) \sqsubseteq \mathsf{Param}_A(d, I, F)}$$

C.2 Lifted Set-Transformers

Here we list typing and refinement-monotonicity theorems for the lifted set-transformer statements described in Chapter 4. Note that we have lifted fixed specification statements, and here we denote free specification statetments as $[P,\ Q]$.

C.2.1 Monotonic Predicate Transformers

Skip	:	$\boldsymbol{\mathcal{M}}_A$	$\{P\}$	:	$\boldsymbol{\mathcal{M}}_A$	$[P,\ Q]$	:	$\boldsymbol{\mathcal{M}}_A$
Abort	:	$\boldsymbol{\mathcal{M}}_A$	$(Q)\!\rightarrow$	:	$\boldsymbol{\mathcal{M}}_A$	$\{R\}$	:	$\boldsymbol{\mathcal{M}}_A$
Magic	:	$\boldsymbol{\mathcal{M}}_A$	$[Q]$	:	$\boldsymbol{\mathcal{M}}_A$	$[R]$	:	$\boldsymbol{\mathcal{M}}_A$
Chaos	:	$\boldsymbol{\mathcal{M}}_A$	$\langle F \rangle$	:	$\boldsymbol{\mathcal{M}}_A$			

$$\frac{a : \mathcal{M}_A \quad b : \mathcal{M}_A}{a; b : \mathcal{M}_A} \qquad \frac{a : \mathcal{M}_A \quad b : \mathcal{M}_A}{\mathsf{if}\ g\ \mathsf{then}\ a\ \mathsf{else}\ b\ \mathsf{fi} : \mathcal{M}_A}$$

$$\frac{a : \mathcal{M}_A \quad b : \mathcal{M}_A}{a \sqcup b : \mathcal{M}_A} \qquad \frac{a : \mathcal{M}_A \quad b : \mathcal{M}_A}{a \sqcap b : \mathcal{M}_A}$$

$$\frac{\forall\, x.\ c(x) : \mathcal{M}_A}{\mathsf{con}\ x.\ c(x) : \mathcal{M}_A} \qquad \frac{\forall\, x.\ c(x) : \mathcal{M}_A}{\mathsf{var}\ x.\ c(x) : \mathcal{M}_A}$$

$$\frac{\forall\, x : T.\ c(x) : \mathcal{M}_A}{\mathsf{con}\ x : T.\ c(x) : \mathcal{M}_A} \qquad \frac{\forall\, x : T.\ c(x) : \mathcal{M}_A}{\mathsf{var}\ x : T.\ c(x) : \mathcal{M}_A}$$

$$\frac{c : \mathcal{M}_A}{\mathsf{while}\ g\ \mathsf{do}\ c\ \mathsf{od} : \mathcal{M}_A} \qquad \mathsf{re}\ N.\ F(N)\ \mathsf{er} : \mathcal{M}_A$$

$$\mathsf{re}\ x : T, N \sqsupseteq I(x).\ c(x, N)\ \mathsf{er} : \mathcal{M}_A$$

C.2.2 Refinement Monotonicity

$$\frac{\forall\, s : A.\ P(s) \Rightarrow P'(s) \quad \forall\, s : A.\ Q'(s) \Rightarrow Q(s)}{[P,\ Q] \sqsubseteq_A [P',\ Q']}$$

Given $a, b, c, d : \mathcal{M}_A$, we have:

$$\frac{a \sqsubseteq_A b \quad c \sqsubseteq_A d}{a; c \sqsubseteq_A b; d} \qquad \frac{a \sqsubseteq_A b \quad c \sqsubseteq_A d}{\mathsf{if}\ g\ \mathsf{then}\ a\ \mathsf{else}\ c\ \mathsf{fi} \sqsubseteq_A \mathsf{if}\ g\ \mathsf{then}\ b\ \mathsf{else}\ d\ \mathsf{fi}}$$

$$\frac{a \sqsubseteq_A b \quad c \sqsubseteq_A d}{a \sqcup c \sqsubseteq_A b \sqcup d} \qquad \frac{a \sqsubseteq_A b \quad c \sqsubseteq_A d}{a \sqcap c \sqsubseteq_A b \sqcap d}$$

Given $a(x), b(x) : \mathcal{M}_A$ for any x, we have:

$$\frac{\forall\, x.\ a(x) \sqsubseteq_A b(x)}{\mathsf{con}\ x.\ a(x) \sqsubseteq_A \mathsf{con}\ x.\ b(x)} \qquad \frac{\forall\, x.\ a(x) \sqsubseteq_A b(x)}{\mathsf{var}\ x.\ a(x) \sqsubseteq_A \mathsf{var}\ x.\ b(x)}$$

Given $a(x) : \mathcal{M}_A$ for any x in T, we have:

$$\frac{\forall\, x : T.\ a(x) \sqsubseteq_A b(x)}{\mathsf{con}\ x : T.\ a(x) \sqsubseteq_A \mathsf{con}\ x : T.\ b(x)} \qquad \frac{\forall\, x : T.\ a(x) \sqsubseteq_A b(x)}{\mathsf{var}\ x : T.\ a(x) \sqsubseteq_A \mathsf{var}\ x : T.\ b(x)}$$

Given $a, b : \boldsymbol{\mathcal{M}}_A$, we have:

$$\frac{a \sqsubseteq_A b}{\mathsf{while}\ g\ \mathsf{do}\ a\ \mathsf{od} \sqsubseteq_A \mathsf{while}\ g\ \mathsf{do}\ b\ \mathsf{od}}$$

Given $a(N), b(N), d(N) : \boldsymbol{\mathcal{M}}_A$ for any N in $\boldsymbol{\mathcal{M}}_A$, we have:

$$\frac{\bigwedge N : \boldsymbol{\mathcal{M}}_A.\ c(N) \sqsubseteq_A d(N)}{\mathsf{re}\ N.\ c(N)\ \mathsf{er} \sqsubseteq_A \mathsf{re}\ N.\ d(N)\ \mathsf{er}}$$

$$\frac{\bigwedge x : T\ N : \boldsymbol{\mathcal{M}}_A.\ c(x) \sqsubseteq_A N \Rightarrow a(x, N) \sqsubseteq_A b(x, N)}{\mathsf{re}\ x\!:\!T, N \sqsupseteq c(x).\ a(x, N)\ \mathsf{er} \sqsubseteq_A \mathsf{re}\ x\!:\!T, N \sqsupseteq c(x).\ b(x, N)\ \mathsf{er}}$$

C.3 Typing-Lifted Set-Transformers

Here we list typing and refinement-monotonicity theorems for the lifted set-transformer statements described in Chapter 6.

C.3.1 Monotonic Predicate Transformers

$$\begin{array}{rcl} \mathsf{Skip} & : & \boldsymbol{\mathcal{M}}_\tau \\ \mathsf{Abort} & : & \boldsymbol{\mathcal{M}}_\tau \\ \mathsf{Magic} & : & \boldsymbol{\mathcal{M}}_\tau \\ \mathsf{Chaos} & : & \boldsymbol{\mathcal{M}}_\tau \\ \mathsf{Choose}(w) & : & \boldsymbol{\mathcal{M}}_\tau \end{array} \qquad \begin{array}{rcl} \{P\} & : & \boldsymbol{\mathcal{M}}_\tau \\ (Q)\!\rightarrow & : & \boldsymbol{\mathcal{M}}_\tau \\ [Q] & : & \boldsymbol{\mathcal{M}}_\tau \\ \langle F \rangle & : & \boldsymbol{\mathcal{M}}_\tau \\ v := E & : & \boldsymbol{\mathcal{M}}_\tau \\ \langle \overrightarrow{M} \rangle & : & \boldsymbol{\mathcal{M}}_\tau \end{array} \qquad \begin{array}{rcl} [P,\ Q] & : & \boldsymbol{\mathcal{M}}_\tau \\ w : [P,\ Q] & : & \boldsymbol{\mathcal{M}}_\tau \\ \{R\} & : & \boldsymbol{\mathcal{M}}_\tau \\ [R] & : & \boldsymbol{\mathcal{M}}_\tau \end{array}$$

$$\frac{a : \mathcal{M}_\tau \quad b : \mathcal{M}_\tau}{a; b : \mathcal{M}_\tau} \qquad \frac{a : \mathcal{M}_\tau \quad b : \mathcal{M}_\tau}{\mathsf{if}\ g\ \mathsf{then}\ a\ \mathsf{else}\ b\ \mathsf{fi} : \mathcal{M}_\tau}$$

$$\frac{a : \mathcal{M}_\tau \quad b : \mathcal{M}_\tau}{a \sqcup b : \mathcal{M}_\tau} \qquad \frac{a : \mathcal{M}_\tau \quad b : \mathcal{M}_\tau}{a \sqcap b : \mathcal{M}_\tau}$$

$$\frac{\forall\, x.\ c(x) : \mathcal{M}_\tau}{\mathsf{con}\ x.\ c(x) : \mathcal{M}_\tau} \qquad \frac{\forall\, x.\ c(x) : \mathcal{M}_\tau}{\mathsf{var}\ x.\ c(x) : \mathcal{M}_\tau}$$

$$\frac{\forall\, x : T.\ c(x) : \mathcal{M}_\tau}{\mathsf{con}\ x : T.\ c(x) : \mathcal{M}_\tau} \qquad \frac{\forall\, x : T.\ c(x) : \mathcal{M}_\tau}{\mathsf{var}\ x : T.\ c(x) : \mathcal{M}_\tau}$$

$$\frac{c : \mathcal{M}_\tau}{\mathsf{while}\ g\ \mathsf{do}\ c\ \mathsf{od} : \mathcal{M}_\tau} \qquad \mathsf{re}\ N.\ F(N)\ \mathsf{er} : \mathcal{M}_\tau$$

$$\frac{c : \mathcal{M}_S}{\mathsf{PARAM}(c, S, I, F) : \mathcal{M}_\tau} \qquad \mathsf{re}\ x : T, N \sqsupseteq I(x).\ c(x, N)\ \mathsf{er} : \mathcal{M}_\tau$$

$$\frac{c : \mathcal{M}_{\tau[T/v]}}{\mathsf{begin}\ v : T.\ c\ \mathsf{end} : \mathcal{M}_\tau} \qquad \frac{c : \mathcal{M}_{\tau \boxplus D}}{\mathsf{begin}\ D.\ c\ \mathsf{end} : \mathcal{M}_\tau}$$

$$\frac{B : \mathcal{M}_{\mathbb{T}_{D,T}} \quad (\bigwedge P.\, P : \mathcal{M} \Longrightarrow C(P) : \mathcal{M}_T)}{\mathsf{proc}\ P(D)\,\mathsf{int} = I\,\mathsf{imp} = B\,\mathsf{in}\ C(P) : \mathcal{M}_T}$$

$$\frac{(\bigwedge P.\, P : \mathcal{M} \Longrightarrow C(P) : \mathcal{M}_T)}{\mathsf{rec}\ P(D)\,\mathsf{int} = I\,\mathsf{var} = v : V, R(v)\,\mathsf{imp} = B(v, P)\,\mathsf{in}\ C(P) : \mathcal{M}_T}$$

C.3.2 Refinement Monotonicity

$$\frac{\forall\, s : \mathcal{S}_\tau.\, P(s) \Rightarrow P'(s) \quad \exists\, s : \mathcal{S}_\tau.\, P(s) \Rightarrow \forall\, s : \mathcal{S}_\tau.\, Q'(s) \Rightarrow Q(s)}{[P,\ Q] \sqsubseteq_\tau [P',\ Q']}$$

$$\frac{\forall\, s : \mathcal{S}_\tau.\, P(s) \Rightarrow P'(s) \qquad \begin{array}{c}\forall\, i : \mathcal{S}_\tau.\, P(i) \Rightarrow \\ (\forall\, o : \mathcal{S}_\tau.\, i\,\mathrm{dsub}\, v = o\,\mathrm{dsub}\, v \wedge Q'(o) \Rightarrow Q(o))\end{array}}{v : [P,\ Q] \sqsubseteq_\tau v : [P',\ Q']}$$

Given $a, c, d : \mathcal{M}_\tau$ we have:

$$\frac{a \sqsubseteq_\tau b \quad c \sqsubseteq_\tau d}{a; c \sqsubseteq_\tau b; d}$$

Given $a, b, c, d : \boldsymbol{\mathcal{M}}_\tau$, we have:

$$\frac{a \sqsubseteq_\tau b \quad c \sqsubseteq_\tau d}{\mathsf{if}\ g\ \mathsf{then}\ a\ \mathsf{else}\ c\ \mathsf{fi} \sqsubseteq_\tau \mathsf{if}\ g\ \mathsf{then}\ b\ \mathsf{else}\ d\ \mathsf{fi}}$$

Given $a, b, c, d : \boldsymbol{\mathcal{M}}_\tau$, we have:

$$\frac{a \sqsubseteq_\tau b \quad c \sqsubseteq_\tau d}{a \sqcup c \sqsubseteq_\tau b \sqcup d} \qquad \frac{a \sqsubseteq_\tau b \quad c \sqsubseteq_\tau d}{a \sqcap c \sqsubseteq_\tau b \sqcap d}$$

If for any x we know $a(x), b(x) : \boldsymbol{\mathcal{M}}_\tau$, then:

$$\frac{\forall\, x.\, a(x) \sqsubseteq_\tau b(x)}{\mathsf{con}\ x.\ a(x) \sqsubseteq_\tau \mathsf{con}\ x.\ b(x)} \qquad \frac{\forall\, x.\, a(x) \sqsubseteq_\tau b(x)}{\mathsf{var}\ x.\ a(x) \sqsubseteq_\tau \mathsf{var}\ x.\ b(x)}$$

Given $a(x) : \boldsymbol{\mathcal{M}}_\tau$ for any x in T, we have:

$$\frac{\forall\, x : T.\, a(x) \sqsubseteq_\tau b(x)}{\mathsf{con}\ x : T.\ a(x) \sqsubseteq_\tau \mathsf{con}\ x : T.\ b(x)} \qquad \frac{\forall\, x : T.\, a(x) \sqsubseteq_\tau b(x)}{\mathsf{var}\ x : T.\ a(x) \sqsubseteq_\tau \mathsf{var}\ x : T.\ b(x)}$$

Given $a, b : \boldsymbol{\mathcal{M}}_\tau$, we have:

$$\frac{a \sqsubseteq_\tau b}{\mathsf{while}\ g\ \mathsf{do}\ a\ \mathsf{od} \sqsubseteq_\tau \mathsf{while}\ g\ \mathsf{do}\ b\ \mathsf{od}}$$

Given $a(N), b(N), d(N) : \boldsymbol{\mathcal{M}}_\tau$ for any N in $\boldsymbol{\mathcal{M}}_\tau$, we have:

$$\frac{\bigwedge N : \boldsymbol{\mathcal{M}}_\tau.\, c(N) \sqsubseteq_\tau d(N)}{\mathsf{re}\ N.\ c(N)\,\mathsf{er} \sqsubseteq_\tau \mathsf{re}\ N.\ d(N)\,\mathsf{er}}$$

$$\frac{\bigwedge x : T\ N : \boldsymbol{\mathcal{M}}_\tau.\, c(x) \sqsubseteq_\tau N \Rightarrow a(x, N) \sqsubseteq_\tau b(x, N)}{\mathsf{re}\ x : T, N \sqsupseteq c(x).\ a(x, N)\,\mathsf{er} \sqsubseteq_\tau \mathsf{re}\ x : T, N \sqsupseteq c(x).\ b(x, N)\,\mathsf{er}}$$

Given $c, d : \boldsymbol{\mathcal{M}}_{\tau[T/v]}$, then

$$\frac{c \sqsubseteq_{\tau[T/v]} d}{\mathsf{begin}\ v : T.\ c\ \mathsf{end} \sqsubseteq_\tau \mathsf{begin}\ v : T.\ d\ \mathsf{end}}$$

Given $c, d : \boldsymbol{\mathcal{M}}_{\tau \boxplus D}$, then

$$\frac{c \sqsubseteq_{\tau \boxplus D} d}{\mathsf{begin}\ D.\ c\ \mathsf{end} \sqsubseteq_\tau \mathsf{begin}\ D.\ d\ \mathsf{end}}$$

See Chapter 6 for the listing and discussion of the refinement-monotonicity theorems for procedures and recursive procedures.

Appendix D

Final Code for the Case Study

This appendix contains the final code for the case study implementation of propositional tautology checking given in Chapter 8. We first present the sugared version, and then the explicit version as appears in Isabelle/ZF.

D.1 Final Code: Sugared

We rely on the following assumptions in the proof of the refinement:

- the external typing is well-typed, i.e. $\tau(r) = \mathbb{B}$ and $\tau(p) = \text{prop}$
- the variable names used in the program are in fact variable names, i.e. $\{a, b, e, l, n, p, r, t, v, x, y\} \subseteq \mathcal{V}$
- the variable names are distinct, specifically:
  - $\text{distinct}(\{p, r\})$
  - $\text{distinct}(\{a, b, t, v, x, y\})$
  - $\text{distinct}(\{l, n, t\})$
  - $\text{distinct}(\{a, b, p\})$
  - $\text{distinct}(\{e, t, p\})$

The collected refinement is as follows:

$$r : [\text{true}, \quad r \equiv \forall\, V : \mathbb{P}\,\mathbb{V}.\, [\![p]\!]_V]$$
$$\sqsubseteq_\tau$$

```
rec Build(value p : prop, result t : rontree)
int =
 t : [true,   BUILD(p, tree_of_ntree(t))]
var = V : prop, p subprop V
imp =
 rec Join(value a : rontree, value b : rontree, result t : rontree)
 int =
    t : [true,
         JOIN(tree_of_ntree(a), tree_of_ntree(b), tree_of_ntree(t))]
 var =  T : tree × tree,
         ⟨tree_of_ntree(a), tree_of_ntree(b)⟩ (subtree ×× subtree) T
 imp =
  proc JoinLeaf(value l : rontree, value n : rontree, result t : rontree)
  int =
    t : [is_nleaf(l),
         JOIN(tree_of_ntree(l), tree_of_ntree(n), tree_of_ntree(t))]
  imp =
   if (l = ⟨true, NLeaf⟩) then  t := n else  t := l fi
  in                              (* end of JoinLeaf's implementation *)
  proc JoinNode(value a : rontree, value b : rontree, result t : rontree)
  int =
    t : [v = ⟨tree_of_ntree(a), tree_of_ntree(b)⟩ ∧ is_nnode(a) ∧ is_nnode(b),
         JOIN(tree_of_ntree(a), tree_of_ntree(b), tree_of_ntree(t))
  imp =
   if (a = b) then
      t := a
    else
      begin  v : 𝕍;  x, y : rontree.
       if (nvar(a) = nvar(b)) then
        v := NNodeV(snd(a));
        Join(nleft(a), nleft(b), x);
        Join(nright(a), nright(b), y)
       else if (nvar(a) < nvar(b)) then
        v := NNodeV(snd(b));
        Join(a, nleft(b), x);
```

```
      Join(a, nright(b), y)
     else
      v := NNodeV(snd(a));
      Join(nleft(a), b, x);
      Join(nright(a), b, y)
     fi fi;
     if (x = y) then
      t := x
     else
      t := ⟨fst(x), NNode(v, snd(x), fst(x) ≡ fst(y), snd(y))⟩
     fi
    end
  fi
in          (* end of JoinNode's implementation *)
if (is_nleaf(a)) then
 JoinLeaf(a, b, t)
else if (is_nleaf(b)) then
 JoinLeaf(b, a, t)
else
 JoinNode(a, b, t)
fi fi
in               (* end of Join's implementation *)
if (is_Var(p)) then
 t := ⟨true, NNode(VarV(p), NLeaf, false, NLeaf)⟩)
else if (is_Not(p)) then
begin e : rontree.
 Build(NotA(p), e);
 t := neg_ntree(e)
end
else
begin a, b : rontree.
```

$Build(\mathrm{AndL}(p), a)$;
$Build(\mathrm{AndR}(p), b)$;
$Join(a, b, t)$
end
fi fi
in (* end of Build's implementation *)
begin t : rontree.
$Build(p, t)$;
$r := (t \equiv \langle \mathrm{true}, \mathrm{NLeaf} \rangle)$
end

D.2 Final Code: Isabelle/ZF

This is the final code for the case study refinement as it appears in Isabelle/ZF. It is unaltered except as concerns spacing.

```
[| B(r) = bool; r : VId; B(p) = prop; p : VId; p ~= r;
   t ~= y; t ~= x; t ~= v; b ~= v; b ~= y; b ~= x; a ~= v;
   a ~= y; a ~= x; x ~= y; v ~= y; v ~= x; y : VId; x : VId;
   v : VId; t ~= n; t ~= l; n ~= l; n : VId; l : VId; b ~= a;
   b ~= t; b ~= p; b : VId; a ~= t; a ~= p; a : VId; e ~= t;
   e ~= p; e : VId; t ~= r; t ~= p; t : VId
|] ==>

FSPEC({r}, %s. True,
          %s. s`r = if(ALL V:Pow(variable). mprop(s`p,V), 1, 0))

REFS(B)

REC Build([Val(p, prop), Res(t, rontree)])
INT === FSPEC({t}, %s. True, %s. BUILD(s`p, tree_of_ntree(s`t)))
VAR === va:prop, %s. s`p subprop v
IMP ===
  REC Join([Val(a,rontree), Val(b,rontree), Res(t,rontree)])
  INT ===
    FSPEC({t}, %s. True,
               %s. JOIN(tree_of_ntree(s`a), tree_of_ntree(s`b),
                   tree_of_ntree(s`t)))
  VAR === va:tree * tree,
```

```
        %s. <<tree_of_ntree(s`a), tree_of_ntree(s`b)>, v> :
               subtree_rel ** subtree_rel
IMP ===
  PROC JoinLeaf([Val(l,rontree), Val(n,rontree), Res(t,rontree)])
  INT ===
    FSPEC({t}, %s. is_ntreeleaf(s`l),
               %s. JOIN(tree_of_ntree(s`l), tree_of_ntree(s`n),
                   tree_of_ntree(s`t)))
  IMP ===
    IF (%s. s`l = <1, NTreeLeaf>) THEN
      t := (%s. s`n)
    ELSE
      t := (%s. s`l)
    FI
  IN
  PROC JoinNode([Val(a,rontree), Val(b,rontree), Res(t,rontree)])
  INT ===
    FSPEC({t}, %s. va = <tree_of_ntree(s`a), tree_of_ntree(s`b)> &
                   is_ntreenode(s`a) & is_ntreenode(s`b),
               %s. JOIN(tree_of_ntree(s`a), tree_of_ntree(s`b),
                        tree_of_ntree(s`t)))
  IMP ===
    IF (%s. s`a = s`b) THEN
      t := (%s. s`a)
    ELSE
      MBEGIN {<v, variable>, <x, rontree>, <y, rontree>}.
        IF (%s. ntree_var(s`a) = ntree_var(s`b)) THEN
          v := (%s. NTreeNode_var(snd(s`a))) SEQ
          Join([Arg(%s. ntree_left(s`a)), Arg(%s. ntree_left(s`b)),
                Arg(%s. x)]) SEQ
          Join([Arg(%s. ntree_right(s`a)), Arg(%s. ntree_right(s`b)),
                Arg(%s. y)])
        ELSE IF (%s. ntree_var(s`a) < ntree_var(s`b)) THEN
          v := (%s. NTreeNode_var(snd(s`b))) SEQ
          Join([Arg(%s. s`a), Arg(%s. ntree_left(s`b)),
                Arg(%s. x)]) SEQ
          Join([Arg(%s. s`a), Arg(%s. ntree_right(s`b)),
                Arg(%s. y)])
        ELSE
          v := (%s. NTreeNode_var(snd(s`a))) SEQ
          Join([Arg(%s. ntree_left(s`a)), Arg(%s. s`b),
                Arg(%s. x)]) SEQ
          Join([Arg(%s. ntree_right(s`a)), Arg(%s. s`b),
```

```
                Arg(%s. y)])
          FI FI SEQ
          IF (%s. s‘x = s‘y) THEN
            t := (%s. s‘x)
          ELSE
            t := (%s. <fst(s‘x),
                      NTreeNode(s‘v, snd(s‘x),
                                fst(s‘x) eq fst(s‘y), snd(s‘y))>)
          FI
        END
      FI
    IN
    IF (%s. is_ntreeleaf(s‘a)) THEN
      JoinLeaf([Arg(%s. s‘a), Arg(%s. s‘b), Arg(%s. t)])
    ELSE IF (%s. is_ntreeleaf(s‘b)) THEN
      JoinLeaf([Arg(%s. s‘b), Arg(%s. s‘a), Arg(%s. t)])
    ELSE
      JoinNode([Arg(%s. s‘a), Arg(%s. s‘b), Arg(%s. t)])
    FI FI
  IN
  IF (%s. is_VAR(s‘p)) THEN
    t := (%s. <1, NTreeNode(VAR_var(s‘p), NTreeLeaf, 0, NTreeLeaf)>)
  ELSE IF (%s. is_NOT(s‘p)) THEN
    MBEGIN {<e, rontree>}.
      Build([Arg(%s. NOT_arg(s‘p)), Arg(%s. e)]) SEQ
      t := (%s. neg_ntree(s‘e))
    END
  ELSE
    MBEGIN {<a, rontree>, <b, rontree>}.
      Build([Arg(%s. AND_left(s‘p)), Arg(%s. a)]) SEQ
      Build([Arg(%s. AND_right(s‘p)), Arg(%s. b)]) SEQ
      Join([Arg(%s. s‘a), Arg(%s. s‘b), Arg(%s. t)])
    END
  FI FI
IN
MBEGIN {<t, rontree>}.
  Build([Arg(%s. s‘p), Arg(%s. t)]) SEQ
  r := (%s. if(s‘t = <1, NTreeLeaf>, 1, 0))
END
```

Bibliography

[1] J. R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] S. Agerholm. Mechanizing program verification in HOL. Master's thesis, Computer Science Department, Aarhus University, April 1992.

[3] S. Agerholm. *A HOL Basis for Reasoning about Functional Programs*. PhD thesis, Computer Science Department, Aarhus University, 1994.

[4] S. B. Akers. Binary decision diagrams. *IEEE Transactions of Computers*, C-27(6), June 1978.

[5] C. M. Angelo, L. Claesen, and H. De Man. Degrees of formality in shallow embedding hardware description languages in HOL. In *Proceedings of the 6th International Workshop on Higher Order Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 89–100. Springer-Verlag, 1994.

[6] R. Arthan. A mechanised notation for specifying and verifying Ada programs using Z. Talk at Cambridge University on July 16, 1996.

[7] R. J. R. Back. On the correctness of refinement steps in program development. Technical Report A-1978-4, Åbo Akademi University, 1978.

[8] R. J. R. Back. Exception handling with multi-exit statements. Technical Report IW 125/79, Mathematisch Centrum, Amsterdam, November 1979.

[9] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[10] R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *LNCS*, pages 67–93. Springer-Verlag, 1989.

[11] R. J. R. Back. The lattice of data refinement. Technical Report 130, Åbo Akademi University, 1992.

[12] R. J. R. Back and J. von Wright. A lattice-theoretical basis for a specification language. In J. L. A. van de Snepscheut, editor, *Mathematics of program construction*, volume 375 of *LNCS*, pages 139–156. Springer-Verlag, 1989.

[13] R. J. R. Back and J. von Wright. Refinement calculus, part I: Sequential nondeterministic programs. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *LNCS*, pages 42–66. Springer-Verlag, 1989.

[14] R. J. R. Back and J. von Wright. Command lattices, variable environments and data refinement. Technical Report 102, Åbo Akademi University, 1990.

[15] R. J. R. Back and J. von Wright. Refinement concepts formalized in higher order logic. *Formal Aspects of Computing*, 2(3):247–272, 1990.

[16] R. J. R. Back and J. von Wright. Interpreting nondeterminism in the refinement calculus. In He Jifeng, editor, *Proceedings of the Seventh BCS FACS Refinement Workshop*, pages 87–96, 1996.

[17] Paul E. Black and Phillip J. Windley. Inference rules for progamming languages with side effects in expressions. In *Theorem Proving in Higher Order Logics: 9th International Conference*, number 1125 in LNCS, pages 51–60, August 1996.

[18] J.-P. Bodeveix and M. Filali. A framework for parallel program refinement. In Uffe H. Engberg, Kim G. Larsen, and Arne Skou, editors, *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number NS-95-2 in Notes Series, pages 159–173. BRICS, May 1995.

[19] Hans-Jurgen Boehm. Side effects and aliasing can have simple axiomatic descriptions. *ACM Transactions on Programming Languages and Systems*, 7(4), October 1985.

[20] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, and John Herbert. Experience with embedding hardware description languages. In *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A10 of *IFIP Transactions*, pages 129–156. North-Holland/Elsevier, June 1992.

[21] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*. IEEE, 1990.

[22] F. Brooks. The computer scientist as toolsmith II. *Communications of the ACM*, 39(3):61–68, March 1996.

[23] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions of Computers*, C-35(8), August 1986.

[24] M. Butler, T. Långbacka, and R. Rukšėnas. *Refinement Calculator Tutorial and Manual*, April 20, 1995.

[25] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. Requirements for a program refinement engine. Technical Report 94-43, Software Verification Research Centre, The University of Queensland, November 1994.

[26] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A review of existing refinement tools. Technical Report 94-8, Software Verification Research Centre, The University of Queensland, February 1994.

[27] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. The PRT user manual. Technical Report 95-56, Software Verification Research Centre, The University of Queensland, December 1995.

[28] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. Refinement in Ergo. Technical Report 94-44, Software Verification Research Centre, The University of Queensland, July 1995.

[29] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A tool for developing correct programs by refinement. Technical Report 95-49, Software Verification Research Centre, The University of Queensland, December 1995.

[30] Robert Cartwright and Derek Oppen. The logic of aliasing. *Acta Informatica*, 15:365–384, 1981.

[31] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[32] J. H. Fetzer. Program verification: The very idea. *Communications of the ACM*, 31(9):1048–1063, September 1988.

[33] C. Fidge, P. Kearney, and M. Utting. Quartz: An integrated formal development method for real-time software. Technical Report 94-26, Software Verification Research Centre, The University of Queensland, 1994.

[34] P. H. B. Gardiner and Carroll Morgan. A single complete rule for data refinement. *Formal Aspects of Computing*, 5:367–382, 1993.

[35] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS.* Springer-Verlag, 1979.

[36] M. J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving.* Springer-Verlag, 1989.

[37] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[38] D. Gries and F. B. Schneider. *A Logical Approach to Discrete Math.* Ithaca, 1993.

[39] David Gries and Gary Levin. Assignment and procedure call proof rules. *ACM Transactions on Programming Language Systems*, 2(4):564–579, October 1980.

[40] L. Groves, R. Nickson, and M. Utting. A tactic driven refinement tool. In R. C. Shaw and T. Denvir, editors, *Proceedings of th Fifth Refinement Workshop*, pages 272–297. Springer-Verlag, 1992. Also available as CS-TR-92/5 from the Department of Computer Science, Victoria University of Wellington.

[41] Jim Grundy. Window inference in the HOL system. In M. Archer and et. al., editors, *Proceedings of the International Tutorial and Workshop*

on the HOL Theorem Proving System and its Applications, pages 177–189, Los Alamitos, California, 1991. IEEE Computer Society Press.

[42] Jim Grundy. *A Method of Program Refinement*. PhD thesis, Computer Laboratory, University of Cambridge, 1993. Also available as TR318.

[43] Jim Grundy. Transformational hierarchical reasoning. *The Computer Journal*, 39(4):291–302, February 1996.

[44] J. R. Harrison. Dinary decision diagrams as HOL derived rule. *The Computer Journal*, 38(2), 1995.

[45] J. Hartmanis. On computational complexity and the nature of computer science. *Communications of the ACM*, 37(10), 1994.

[46] I. J. Hayes and C. Fidge. A sequential real-time refinement calculus. Technical Report 97-33, Software Verification Research Centre, The University of Queensland, August 1997.

[47] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[48] C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1(4):271–281, 1972.

[49] P. V. Homeier and D. F. Martin. Trustworthy tools for trustworthy programs: a verified verification condition generator. In Melham, T. F. and Camilleri, J., editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 269–284, Valletta, Malta, September 1994. Springer-Verlag.

[50] M. H. Jarvis, C. C. Morgan, and P. H. B. Gardiner. *The RED Manual*.

[51] C. B. Jones. *Sytematic Software Development using VDM*. Series in Computer Science. Prentice-Hall, second edition, 1990.

[52] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System*. Springer-Verlag, 1991.

[53] Immanuel Kant. *Prolegomena to Any Future Metaphysics*. Supplied by James Fieser (`jfieser@utm.edu`), `http://www.utm.edu/research/iep/text/kant/prolegom/prolegom.txt`, 1793. Based on Carus's 1902 translation.

[54] D. J. King and R. D. Arthan. Development of practical verification tools. *The ICL Systems Journal*, 11(1), May 1996.

[55] Steve King and Carroll Morgan. Exits in the refinement calculus. *Formal Aspects of Computing*, 7:54–76, 1995.

[56] Thomas Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, Laboratory for Foundations of Computer Science, The University of Edinburgh, September 1998.

[57] J. Knappmann. A PVS based tool for developing programs in the refinement calculus. Master's thesis, Christian-Albrechts-University of Kiel, October 1996.

[58] Thomas S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, Chicago, 2nd edition, 1970.

[59] K. Lano. *Specification in B: An Introduction using the B Toolkit*. World Scientific Publishing Company, Imperial College Press, 1996.

[60] C. Y. Lee. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, XXXVII(4), July 1959.

[61] K. Rustan M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In *Proceedings of the Fourth International Workshop on Foundations of Object-Oriented Languages*, January 1997.

[62] Alain J. Martin. A general proof rule for procedures in predicate transformer semantics. *Acta Informatica*, 20:301–313, 1983.

[63] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[64] R. Milner. Is computing an experimental science? Technical Report ECS-LFCS-86-1, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1986.

[65] R. Milner. Semantic ideas in computing. In M. Wand and R. Milner, editors, *Computing Tomorrow: The Future of Research in Computer Science*, pages 246–283. Cambridge University Press, 1997.

[66] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[67] J. S. Moore. Introduction to the OBDD algorithm for the ATP community. *Journal of Automated Reasoning*, 12(1), February 1994.

[68] C. Morgan and P. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990.

[69] C. Morgan and T. Vickers. Types and invariants in the refinement calculus. *Science of Computer Programming*, 14:281–304, 1990.

[70] Carroll Morgan. Procedures, parameters, and abstraction: Separate concerns. *Science of Computer Programming*, 11(1):17–28, 1988.

[71] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[72] Carroll Morgan. *Programming from Specifications*. Prentice-Hall International, 1990.

[73] Carroll Morgan. *Programming from Specifications*. Prentice-Hall International, 2nd edition, 1994.

[74] Carroll Morgan, Annabelle McIver, Karen Seidel, and J. W. Sanders. Probabilistic predicate transformers. Technical Report PRG-TR-4-95, Programming Research Group, Oxford University Computing Laboratory, 1995.

[75] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

[76] J. M. Morris. Piecewise data refinement. In E. W. Dijkstra, editor, *Formal Development of Programs and Proofs*, chapter 10, pages 117–137. Addison-Wesley, 1990.

[77] J. M. Morris. Programs from specifications. In E. W. Dijkstra, editor, *Formal Development of Programs and Proofs*, chapter 9, pages 81–115. Addison-Wesley, 1990.

[78] Joseph M. Morris. Laws of data refinement. *Acta Informatica*, 26:287–308, 1989.

[79] *Formal Methods Specification and Verification Guidebook for Software and Computer Systems*, 1.0 edition, July 1995. Volume 1. NASA-GB-002-95.

[80] R. Nickson. *Tool Support for the Refinement Calculus*. PhD thesis, Victoria University of Wellington, 1993.

[81] R. G. Nickson and L. J. Groves. Metavariables and conditional refinements in the refinement calculus. Technical Report 93-12, Software Verification Research Centre, The University of Queensland, 1993.

[82] R. G. Nickson and I. Hayes. Program window inference. Technical Report 95-29, Software Verification Research Centre, The University of Queensland, 1995.

[83] R. G. Nickson and I. Hayes. Supporting contexts in program refinement. *Science of Computer Programming*, 29(3):279–302, 1997.

[84] Michael Norrish. *C Formalised in HOL*. PhD thesis, Computer Laboratory, University of Cambridge, 1998.

[85] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.

[86] H. Pfeifer, A. Dold, F. W. von Henke, and H. Rueß. Mechanised semantics of simple imperative programming constructs. Technical Report UIB-96-11, Universität Ulm, December 1996.

[87] K. Popper. *Conjectures and Refutations: The Growth of Scientific Knowledge*. Routledge, London, 1963.

[88] Chris H. Pratten. *Refinement in a Language with Procedures and Modules*. PhD thesis, Department of Engineering and Computer Science, University of Southampton, June 1996.

[89] *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall International, 1992.

[90] P. J. Robinson and J. Staples. Formalizing the hierarchical structure of practical mathematical reasoning. *Journal of Logic and Computation*, 3(1), February 1993.

[91] K. Rustan and J. L. A van de Snepscheut. Semantics of exceptions. Technical Report cs-tr-93-34, California Institute of Technology, 1993.

[92] C. Shannon. The mathematical theory of communication. *Bell System Technical Journal*, 27:379–656, 1948.

[93] H. Simon, editor. *Sciences of the Artificial*. The MIT Press, 1st edition, 1980.

[94] H. Simon, editor. *Sciences of the Artificial*. The MIT Press, 3rd edition, 1996.

[95] Mark Staples. Window inference in Isabelle. In L. Paulson, editor, *Proceedings of the First Isabelle User's Workshop*, volume 379, pages 191–205. University of Cambridge Computer Laboratory Technical Report, September 1995.

[96] Mark Staples. Flexible interactive transformational reasoning. In *Fifth Australian Refinement Workshop*, 1996.

[97] Gavan Tredoux. Mechanising nondeterministic programming logics in higher-order logic. Master's thesis, Department of Mathematics, University of Cape Town, March 1993.

[98] A. M. Turing. Intelligent machinery. *Machine Intelligence*, 5, 1969. 1948 National Physical Laboratory Report.

[99] M. Utting and K. Whitwell. Ergo user manual. Technical Report 93-19, Software Verification Research Centre, The University of Queensland, February 1994.

[100] J. von Wright. The lattice of data refinement. *Acta Informatica*, 31, 1994.

[101] J. von Wright. Verifying modular programs in HOL. Technical Report 324, University of Cambridge Computer Laboratory, January 1994.

[102] J. von Wright. *A Mechanised Calculus of Refinement in HOL*, January 27, 1994.

[103] Joakim von Wright. Extending window inference. In *Theorem Proving in Higher-Order Logics*, number 1479 in LNCS, 1998.