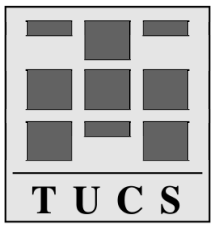
Mechanised Formal Reasoning About Modular Programs

by

Linas Laibinis



Turku Centre for Computer Science

TUCS Dissertations

No 24, April 2000

Mechanised Formal Reasoning About Modular Programs

Linas Laibinis

To be presented – with the permission of the Faculty of Mathematics and Natural Sciences at Åbo Akademi University – for public criticism in Auditorium 3102 at the Department of Computer Science at Åbo Akademi University, on April 14th, 2000, at 12 noon.

Department of Computer Science
Åbo Akademi University

Acknowledgements

I owe my deepest gratitude to my supervisor — Professor Joakim von Wright. Despite his busy schedule, he has always managed to find time for discussions on various aspects of my research. Without his continuous encouragement and friendly support combined with invaluable expert advice, this thesis would have never been finished.

Dr. David Carrington from the University of Queensland, Australia, and Prof. Thomas Melham from the University of Glasgow, United Kingdom have kindly accepted the role of reviewers and helped me by providing valuable feedback on the draft of this dissertation. I wish to thank for their time and efforts.

Many people have influenced my scientific growth by showing how high-quality research is to be done. I am especially grateful to Prof. Ralph Back, Åbo Akademi University, Dr. Michael Butler, currently at the University of Southampton, United Kingdom, Dr. John Harrison, currently at Intel Corporation, USA, Dr. Jim Grundy, currently at The Australian National University, and Dr. Emil Sekerinski, currently at McMaster University, Canada.

I would like to thank the Department of Computer Science at Åbo Akademi University and Turku Centre for Computer Science for the generous financial support and excellent work conditions provided during my studies. The financial support of the final year of my studies provided by the Academy of Finland is also gratefully acknowledged.

I wish to express my gratitude to all my colleagues at Turku Centre for Computer Science, and especially Marina Waldén, Mauno Rönkkö, Martin Büchi and Philipp Heuberger, for creating friendly and inspiring working atmosphere.

I would like to thank all professors and lecturers of my alma mater, Vilnius University. I am especially grateful to my first scientific advisor Dr. Vytautas Čyras and the Head of the Department of Computer Science Dr. Antanas Mitašiūnas for encouraging my scientific aspirations.

It is time to thank my friends whose company made my study years more cheerful. Warmest hugs to Rimvydas, Anna and Leonid, Elena and Vladimir. Thank You for Your friendship and all those enjoyable moments that we have spent together.

Finally, I wish to thank my parents for their love and support. Thank You for always believing in me, in good times and in worse. This dissertation is dedicated to You.

Linas Laibinis
Turku, March 10th, 2000

Contents

Chapter 1

Introduction

1.1 Abstract of the Thesis

This dissertation presents new techniques for the mechanised development of provably correct programs using the stepwise refinement paradigm. Program development techniques are described in a completely formal way, i.e., mathematically defined theories are used to justify the correctness of program implementations with respect to their specifications. The mathematical basis of our work is the refinement calculus theory developed by Back and von Wright[4, 15].

Tool support for program refinement can increase our confidence in our refinement theory and also in the correctness of our program derivations. Our work is performed in the HOL theorem prover[43] which mechanises a classical higher-order logic. We use this expressive logic to model our specification language. In addition, the Refinement Calculator tool[24, 26] built on the top of the HOL system provides a graphical environment for program development under window inference – a style of formal reasoning which is known to be especially well-suited for program refinement proofs. We present most of the contributions of this dissertation in the form of extensions of the Refinement Calculator tool.

The emphasis of this work is on mechanised formal reasoning about modular units of programs such as procedures and functions in the refinement calculus framework. We extend the mechanisation of the refinement calculus in the HOL system with new constructs for procedure and function calls. That allows us to actually derive a number of refinement and correctness properties that were just postulated before. We also show how our method for modelling procedures can be integrated into the Refinement Calculator

tool and used for program derivations.

The techniques presented also rely on other mathematical properties of the program model defined in the refinement calculus theory. We show how abstract lattice-theoretical properties of the program model can be used in program development, and also how program context information can be introduced and taken into account in program refinement proofs. These contributions are presented as the extensions of the Refinement Calculator as well. We also investigate the relationship between correctness assertions and specification statements of the refinement calculus, and show how our results can be applied to the derivation of proof rules for correctness of procedure calls.

1.2 Motivation

"Buggy" software It is no secret that most software contain errors. One has become used to the fact that every major software system that is released is virtually guaranteed to contain so called "bugs" and is followed by subsequent releases where some (most obvious) of them are fixed. Occasional collapses of software systems are often considered as an inevitable evil that must be endured. In a society which becomes more and more computerised and, therefore, dependent on the quality of computer software, the fact that erroneous software is so commonplace is worrying.

Why is such a situation tolerated? The reason is that it is impossible to actually create perfect software of the size and complexity desired, using the current technology of testing to detect errors. The tester usually treats a piece of software as a "black box" which produces some results/responses for given inputs/circumstances. The responses are then compared to the expected ones. However, even with some knowledge of the internal program structure, it is very difficult in many cases to exhaustively test all possible paths through the software, or all combinations of circumstances in which the software will be expected to function. The core of the problem remains, as expressed by Dijkstra[34]: "Program testing can be used to show the presence of bugs, but never to show their absence!".

Use of formal methods In order to cope with the problem of erroneous software, *formal methods* were proposed as a way of development of error-free software. The idea of formal methods is to prove mathematically that the software being developed will function as expected. The "expectations" are written in the form of a formal *specification* – a precise mathematical

description of the system. The proof then shows that the specification and the program, two forms of representing the same system, are consistent with each other. Thus, if the specification is complete and correct, then the program is guaranteed to perform correctly as well.

Formal methods use logic and mathematics to represent systems or languages. Therefore, to write a formal specification of a system requires much more precision than to give it an informal description. However, this very difficulty encourages a clearer and more meticulous analysis of the system.

Of course, one obvious limitation of this approach should be recognized. Deriving a complete and correct specification for a problem from the vague and nuanced words of a description in English (or any other language) is a difficult and uncertain process in itself. If the formal specification arrived at is not what was truly intended, then the entire proof activity may be considered as an exercise of redundancy. However, the main reason why it is so difficult to eradicate all errors in a large software system is an explosion of complexity. A specification as an abstract mathematical description of a system is much more concise which makes it easier to avoid errors.

Program verification vs. program refinement There are two different formal approaches for developing correct software – *program verification* and *program refinement*. The program verification approach allows us to prove that a given program has the desired properties, i.e., is consistent with its specification. Typically, programs are written in some predefined programming notation which has a precise semantics. The program verification task then amounts to proving that a given specification and a given program are logically related. For example, Hoare's classical approach (when one proves that a given program is correct with respect to a given pre/postcondition pair) falls into this category.

In refinement-based approaches one does not have a program to prove properties about. Instead, an executable program is derived from its specification by a series of transformations called *refinements*. The result of each refinement step is logically proved to be consistent with the initial specification. A specification language used in refinement-based approaches usually represents both imperative programming statements and non-executable specification constructs. Specification constructs abstract implementation details, and therefore contain nondeterminism that should be resolved during the program development. A refinement step in the program development then corresponds to preserving functional correctness while decreasing nondeterminism. Ultimately, an executable program which is provably correct

with respect to the given high-level specification is derived. The refinement calculus theory[4, 15] which serves as the mathematical basis of the thesis is an example of a refinement-based approach for program development.

Need for tool support Application of formal methods to sample programs has shown that for even moderately sized programs, the proofs are often very long and involved, and full of complex details. This raises the possibility of errors occurring in the proof itself, and brings its credibility into question.

Assistance may be provided by a tool which records and maintains the proof as it is constructed step by step, and ensures its soundness. The tool then becomes an agent which mechanically verifies correctness of the proof. Such a tool can provide the high accuracy needed in complex mathematical proofs, while handling the increasing amount of detail at the same time. Of course, the tool itself should satisfy basic requirements – be reliable and secure, i.e., perform only sound logical inferences. Moreover, the logic of the tool should be expressive enough to represent all statements of a program specification language of our model.

The Higher Order Logic[43] (HOL) proof assistant is an example of such a reliable and expressive tool. It is an interactive theorem-proving environment for higher-order logic, based on the LCF[78] approach to mechanical theorem proving developed by Milner. The HOL system is "open", i.e., the user can extend its functionality by constructing programs which automate whatever theorem-proving strategy he/she prefers. This programmability makes it especially suitable to serve as the basis for creation of new experimental tools.

1.3 Outline of the Thesis

Chapters 2 and 3 contain the material which constitutes the background of this thesis. Chapter 2 describes the refinement calculus theory which forms the mathematical basis of our work. We explain here the main notions of the refinement calculus such as the stepwise refinement paradigm, the weakest precondition semantics, program correctness etc. We also introduce the constructs of a program specification language that we use throughout the thesis, and give their semantic interpretations.

Chapter 3 reviews mechanical tools and techniques that are exploited in our work. It briefly describes the HOL theorem prover, its logic and implemented styles of formal reasoning. The window inference style of rea-

soning (implemented as the HOL window Library) turns out to be especially convenient when developing programs in a stepwise refinement fashion. The mechanisation of the refinement calculus is presented as a theory of the HOL system. Finally, we give a short description of the Refinement Calculator – a program refinement tool built on the top of the HOL system and its window Library.

In chapters 4–6 we present our contributions in the form of extensions of the Refinement Calculator. They cover different mathematical aspects of the program development by program refinement. Chapter 4 explains how abstract lattice-theoretical properties can be used when reasoning about programs. Once proved in a HOL theory, the properties of abstract lattices can later be automatically instantiated and used in different concrete domains that have been proved to be lattices. Chapter 5 describes formally the notion of program context and shows how it can be used in program refinement. In this chapter we also present two approaches for propagating of context information from one program place to another.

The main part of the thesis is devoted to mechanised formal reasoning about procedures and procedure calls. In Chapter 6 we show how procedures and procedure calls can be modelled in the refinement calculus, and explain the implementation of the approach in the HOL theory of the refinement calculus and in the Refinement Calculator tool. Chapter 7 illustrates the ideas presented with a bigger example. Using context information for proving refinement or correctness properties of procedures and procedure calls is discussed in Chapter 8. Chapter 9 shows how we can deal with recursive procedures and their calls. In Chapter 10 we discuss the use of procedure variables in our implementation. Reasoning about procedures defined by correctness assertions is described in Chapter 11, while Chapter 12 presents a new approach for modelling functional procedures and their calls.

Finally, Chapter 13 concludes with a summary of the thesis and discusses possible future extensions of the work presented.

Some of the work presented in this thesis was earlier published in the proceedings of scientific conferences or as technical reports of Turku Centre for Computer Science. The list of publications is as follows:

1. L. Laibinis.
   Using Lattice Theory in Higher Order Logic.
   In J. von Wright, J. Grundy, J. Harrison (Eds.), Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics, August 1996, Turku, Finland. Springer-Verlag, Lecture Notes in Computer Science 1125, pg. 315–330, 1996.

2. L. Laibinis and J.von Wright.
Context handling in the Refinement Calculus framework.
In U. M. Haveraaen, O. Owe (Eds.), Proceedings of the 8th Nordic Workshop on Programming Theory, December 1996, Oslo, Norway. Research Report 248, Department of Informatics, University of Oslo.

3. L. Laibinis and J.von Wright.
What's in a Specification?
In J.Grundy, M.Schwenke, T.Vickers (Eds.), Proceedings of International Refinement Workshop & Formal Methods Pacific'98, September 1998, Canberra, Australia. Springer-Verlag, pg. 180–192, 1998.

4. L. Laibinis and J.von Wright.
Functional Procedures in Higher-Order Logic.
TUCS technical report No.252, 1999.

5. L. Laibinis.
Mechanising Procedures in HOL.
TUCS technical report No.253, 1999.

Chapter 2

The Refinement Calculus theory

2.1 Introduction

Program Specification and Refinement The purpose of using formal methods for program development is to prove mathematically that programs behave in certain well-defined ways. However, in order to be able to reason formally about programs, one needs to have a starting point. Such a starting point is a *specification* – a precise high-level mathematical description of the intended behaviour of a system. A specification usually contains only essential properties or requirements that a system should satisfy, and therefore allows for different implementations.

The *stepwise refinement* method for program construction allows derivation of an executable program from its specification by a series of small transformations (refinements). Each refinement is proved to be correct with respect to the initial specification, i.e., it preserves the semantic system properties encoded in the specification. In programming theory two notions of correctness have been used: *partial correctness* and *total correctness*. Partial correctness means that every result that the program yields is consistent with what is specified. However, partial correctness admits the possibility of not producing any result at all, in case when the program does not terminate. Thus, any non-terminating program is still partially correct. Compared with partial correctness, total correctness is a stronger requirement. It signifies that every run of the program will in fact terminate and the obtained result is consistent with what is specified.

An initial specification abstracts implementation details and, therefore,

is typically highly nondeterministic. As such, it may not be executable. However, the nondeterminism contained in a specification leaves an implementor freedom to make his/her design decisions. A refinement step then usually corresponds to a certain implementation decision in the "top-down" style of system development. Hence, in general, refinement makes a program more deterministic. Another reason for refinement could be making a program more efficient while preserving correctness. In both cases refinement as a process (of making a program more deterministic or more efficient) intuitively can be understood as a program improvement. The ultimate goal for this process is to develop a detailed executable program.

2.2 The Refinement Calculus

The notion of refinement constitutes the basis of several formalisms for program development and verification[4, 86, 84]. In our thesis the refinement calculus theory developed by Ralph Back and Joakim von Wright[4, 6, 12, 15] is used. This theory presents a framework for development of programs that are totally correct by construction. Lattice theory and higher-order logic together form the mathematical basis for the calculus. This allows us to prove correctness of programs and to calculate program refinements in a rigorous, mathematically precise manner.

The focus of the theory is imperative state based programs. The language used to express programs and specifications is essentially Dijkstra's language of guarded commands, with some extensions.

The refinement calculus is based on Dijkstra's weakest precondition semantics for programs[35]. The meaning of different program statements is defined by means of predicates over the program variables (the program state). Given a statement and a predicate describing the intended result (a postcondition), the weakest precondition semantics defines the weakest initial predicate (a precondition) that guarantees that the intended result is achieved. Therefore, program statements are modelled as functions that map postconditions to preconditions.

Mathematically, the weakest precondition semantics can be defined as a function *wp* which takes two arguments – a statement and a postcondition. The semantic meaning of a statement S for a given postcondition p is the value $wp(S, p)$. The *wp* semantics of programs is compositional in the sense that the semantics of program constructs composed from several "atomic" statements is defined via *wp* of composing statements. For example, the weakest precondition of sequential composition of statements S_1 and S_2

(denoted as $S_1; S_2$) is defined as follows:

$$wp(S_1; S_2, p) \;=\; wp(S_1, wp(S_2, p))$$

The refinement calculus theory extends Dijkstra's work by introducing a *refinement relation* between programs. The refinement relation between programs S and S' is defined using their weakest precondition semantics: we say that a program (a statement) S is refined by S', written $S \;\sqsubseteq\; S'$, iff whenever S establishes a certain postcondition, so does S'[1]:

$$\forall q.\; wp(S, q) \;\Rightarrow\; wp(S', q)$$

The refinement relation is a preorder, i.e., a reflexive and transitive relation. Transitivity of the relation justifies program development in a linear fashion by a series of refinement steps: verifying refinements

$$S_1 \;\sqsubseteq\; S_2 \;\sqsubseteq\; ... \;\sqsubseteq\; S_n$$

we, in fact, by transitivity establish the refinement $S_1 \sqsubseteq S_n$.

Dijkstra's language of guarded commands originally contained only executable program constructs. An introduction of unimplementable specification statements into the language have led to a partial relaxation of this requirement[4]. However, it enriched the language with the useful abstraction mechanism: constructing a program as a mixture of abstract and executable statements makes it possible to conduct program development from an abstract specification to an executable program within a single framework.

The total correctness of programs (or program statements) may be expressed by logical formulae called *Hoare triples*, each containing a *precondition* p that may be assumed to hold in the initial state, program (statement) S, and *postcondition* q that is required to hold in the final state. We denote such a Hoare triple as $\{p\}\; S\; \{q\}$. The program S is said to be *correct* with respect to this Hoare triple specification, if the final state computed by the program satisfies q whenever the initial state satisfies p.

Program refinement can then be understood as modifying the program while preserving correctness. Alternatively, we say that statement S is refined by statement S', if S' is correct with respect to any pre-postcondition pair (p, q) whenever S is correct with respect to (p, q).

[1] The implication in the definition really should be read as "everywhere implies" (see Section 2.4).

Since programs tend to become very large and complex, it is usually too difficult to prove the correctness of a whole program directly. Instead, one can construct correct programs by a sequence of small incremental refinements. Each refinement is focused on some small component of the program and essentially means replacing this component with another component in a way that improves the original program.

This can be done without compromising correctness because all statement constructors of the refinement calculus language are *monotonic* with respect to refinement. For example, we can show that the following holds:

$$S_1 \sqsubseteq S_1' \ \wedge \ S_2 \sqsubseteq S_2' \ \ \Rightarrow \ \ S_1; S_2 \sqsubseteq S_1'; S_2'$$

i.e., sequential composition is monotonic in both arguments.

Thus one can refine the whole program by focusing on a small component and refining it in isolation from the context.

2.3 Underlying Logic

The logical basis of the refinement calculus is a classical *higher-order logic*[3, 43]. This is an extension of the simply typed lambda calculus originally developed by Church[31]. In this logic all variables are given types, and quantification is over the values of a type. Being "higher-order logic" means that quantification is allowed over predicates and functions of any order. Polymorphic types containing type variables are also permitted in the logic. All this makes it possible to reason logically about such higher-order entities as predicates, relations, predicate transformers etc. which is necessary when dealing with program correctness and refinement.

The logic can be extended in two ways, namely, by the definition of new types and type constructors, and by the definition of new constants (including new functions and predicates). However, the syntax of the higher-order logic is fixed. A type can be either a type variable or a type operator applied to n other well-defined types. If the operator arity n equals 0, then we have a constant type (like truth values or integer numbers). The function type operator $\rightarrow$ is considered basic and often included explicitly describing the type syntax. The latter can be then expressed with the following grammar:

$$\tau \ \ ::= \ \ \alpha \mid op^n(\tau_1, ..., \tau_n) \mid \tau_1 \rightarrow \tau_2$$

where α is a type variable and op^n is some type operator with arity n.

A term (expression) can be either a variable, a constant symbol, a function application or an abstraction. This can be expressed with the following

grammar:

$$t \quad ::= \quad x \mid c \mid t.t' \mid (\lambda x \bullet t)$$

where x is a variable and c is a constant.

In the refinement calculus, the type structure of higher-order logic is extended with *product types*, which stand for Cartesian product of types. New product types are formed using a binary infix type operator $\times$ which is associated to the right. Therefore, a type $\Sigma_1 \times \Sigma_2 \times \Sigma_3$ actually stands for $\Sigma_1 \times (\Sigma_2 \times \Sigma_3)$.

We also use special *paired (tupled) abstraction* for products. This is syntactic sugaring:

$$(\lambda u : \Sigma,\, v : \Gamma \bullet\ t) \quad \widehat{=} \quad (\lambda w : \Sigma \times \Gamma \bullet\ t[u, v/\mathit{fst}.\, w, \mathit{snd}.\, w])$$

where *fst* and *snd* are the corresponding projection functions returning the first and the second components of a tupled variable. The paired abstraction introduces names for the components of the product, which then can be used inside the term. For example, a lambda abstraction

$$(\lambda(x : \mathit{Int}, y : \mathit{Int}) \bullet\ x + y)$$

actually stands for

$$(\lambda w : \mathit{Int} \times \mathit{Int} \bullet\ \mathit{fst}.\, w\ +\ \mathit{snd}.\, w)$$

2.4 State Predicates, Relations, and Predicate Transformers

In this section we quickly review mathematical structures and notation that we use throughout the thesis.

In the refinement calculus the program state is modelled as a polymorphic type Σ or Γ which for concrete programs can be instantiated in different ways. In this thesis we consider a state to be a tuple (i.e., of product type) where every component corresponds to a program variable. In the recent textbook[15] Back and von Wright use a more abstract axiomatic model of the program state and program variables. However, modelling the program state as a tuple can be shown to be a special case of the approach presented in the book.

State functions are just functions of type $\Sigma \to \Gamma$ (for a given state they yield the new state that may be of a different type). The predicates over a state space (type) Σ are functions from Σ to *Bool*, denoted by $\mathcal{P}\Sigma$. The *state*

relations from Σ to Γ are functions from Σ to a predicate (set of values) over Γ, denoted by $\Sigma \leftrightarrow \Gamma$. The *predicate transformers* from Σ to Γ are functions mapping predicates over Γ to predicates over Σ, denoted by $\Sigma \mapsto \Gamma$ (note the reversion of the direction), or by $Ptran(\Sigma)$ in the case of $\Sigma \mapsto \Sigma$.

Function application is denoted by a dot, for example $f.x$. Function composition is denoted by infix operator $\circ$ and is defined in the following way:

$$(f \circ g).x \;\; \widehat{=} \;\; f.(g.x)$$

Since a state predicate p corresponds to the set of states that p maps to the boolean value T, we find it justified to use set notation for operations on predicates (intersection, union, subset). These operations are pointwise extensions of the corresponding operations on booleans. For example, the intersection operation is defined as a lifted conjunction:

$$p \cap q \;\; \widehat{=} \;\; (\lambda \sigma \bullet \; p.\sigma \; \wedge \; q.\sigma)$$

The entailment (subset) ordering $p \subseteq q$ on predicates $p, q : \mathcal{P}\Sigma$ is defined as the universal implication on booleans, i.e.,

$$p \subseteq q \;\; \widehat{=} \;\; (\forall \sigma : \Sigma \bullet \; p.\sigma \; \Rightarrow \; q.\sigma)$$

The predicates *true* and *false* over Σ map every $\sigma : \Sigma$ to the boolean values T and F, respectively. Also, $\sigma \in p$ and $p.\sigma$ say the same thing (that p holds in state σ).

The *refinement ordering* $S \sqsubseteq S'$, read *S is refined by S'*, on statements $S, S' : \Sigma \mapsto \Gamma$ is defined by universal entailment:

$$S \sqsubseteq S' \;\; \widehat{=} \;\; (\forall q : \mathcal{P}\Gamma \bullet \; S.q \; \subseteq \; S'.q)$$

A predicate transformer $S : \Sigma \mapsto \Gamma$ is said to be *monotonic* if for all predicates p and q, $p \subseteq q$ implies $S.p \subseteq S.q$.

A predicate transformer $S : \Sigma \mapsto \Gamma$ is called *conjunctive* if it satisfies the following property:

$$S.(\cap i \mid i \in I \bullet \; q_i) \;\; = \;\; (\cap i \mid i \in I \bullet \; S.q_i)$$

for an arbitrary nonempty collection $\{q_i \mid i \in I\}$ of predicates.

Notation The result of substituting a term t for all free occurrences of a variable x in a term s (with suitable renaming of bound variables to avoid variable capture) is denoted by s^x_t or $s[x/t]$. We allow the *bounded quantification* format $(\forall x \mid s \bullet t)$ as equivalent to $(\forall x \bullet s \Rightarrow t)$. The same format is also used for intersection and union over sets. Thus $(\cap i \,|\, i \in I \bullet t_i)$ is the intersection of all t_i where i ranges over the set I.

2.5 Language of Program Statements

The refinement calculus specification language we use in the thesis has the following syntax:

$$\begin{array}{rcl} S & ::= & \mathbf{skip} \mid \mathbf{abort} \mid \mathbf{magic} \mid \{p\} \mid [p] \mid \langle f \rangle \mid \{P\} \mid [P] \mid \\ & & S_1; S_2 \mid S_1 \sqcup S_2 \mid S_1 \sqcap S_2 \mid \mathbf{if}\ g\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi} \mid \\ & & \mathbf{do}\ g \rightarrow S\ \mathbf{od} \mid \mathbf{block}\, p\, S \mid S_1 \| S_2 \end{array}$$

Here p is a state predicate, f is a state function and P is a state relation. The syntax presented above is not closed; new derived constructs can be added later. Below we give the semantic definitions of the language constructs presented above.

Statements from Σ to Γ are identified with monotonic predicate transformers in $\Sigma \mapsto \Gamma$. They are defined according to their weakest precondition semantics, i.e., application of statement S to postcondition q, written as $S.\,q$, corresponds to $wp(S, q)$. Statements of this kind may be concrete, i.e., executable, or abstract, i.e., specifications. The refinement calculus includes all standard program statements, such as assignments, conditionals, and loops.

The statement **abort** does not guarantee any outcome or termination, therefore, it maps every postcondition to *false*. The statement **magic** is *miraculous*, since it is always guaranteed to establish any postcondition. The statement **skip** leaves the state unchanged. The definitions of these statements are as follows:

$$\mathbf{abort}.\,q \mathrel{\widehat{=}} \mathit{false} \qquad \mathbf{magic}.\,q \mathrel{\widehat{=}} \mathit{true} \qquad \mathbf{skip}.\,q \mathrel{\widehat{=}} q$$

The *assertion* statement $\{p\}$ indicates that the predicate p is known to hold at a certain point in the program. The assertion $\{p\}$ behaves as **abort** if p does not hold, and as **skip** otherwise. Formally, it is defined as follows:

$$\{p\}.\,q \mathrel{\widehat{=}} p \cap q$$

As an example, let us consider the assertion $\{\lambda(x, y) \bullet x > y\}$. The program state here consists of two integer components (variables) x and y. The assertion then states that the value of x should be greater than the value of y at the particular place of a program where this assertion occurs. As we show later, assertion predicates can provide additional context information for program refinements.

In our model program variables are defined as state projection functions, i.e., they indicate positions in the state tuple. In order to obtain the

value of a program variable in the current state, one has to apply the program variable (as a projection function) to this state, extracting in this way the corresponding state component. For example, the assertion statement $\{\lambda(x, y) \bullet\ x > y\}$ presented above actually stands for

$$\{\lambda\, s : Int \times Int \bullet\ x.\, s > y.\, s\}$$

where x and y are defined as the corresponding projection functions *fst* and *snd*. However, for the sake of simplicity, in our examples we continue to use tupled λ-abstraction of the program state.

The *assumption* (or *guard*) statement $[p]$ indicates that the predicate p is assumed (but not known) to hold at a certain point in the program. The assumption $[p]$ behaves as **magic** if p does not hold, and as **skip** otherwise. Formally, it is defined in the following way:

$$[p].\, q \ \ \widehat{=}\ \ p \subseteq q$$

The *functional update* statement $\langle f \rangle$ where f is a state function of type $\Sigma \to \Gamma$ describes a functional state change and is defined as follows:

$$\langle f \rangle.\, q.\, \sigma \ \ \widehat{=}\ \ q.\, (f.\, \sigma)$$

Working with concrete programs, we often use the commonly accepted assignment syntax for functional updates as well. For example, in the program state consisting of two program variables x and y the assignment $x := e$ (where e is some expression over program variables) corresponds to the functional update $\langle \lambda(x, y).\, (e, y) \rangle$.

The language supports two kinds of non-deterministic updates which, in fact, represent specification statements. Given a relation $P : \Sigma \leftrightarrow \Gamma$, the *angelic update* $\{P\} : \Sigma \mapsto \Gamma$, and the *demonic update* $[P] : \Sigma \mapsto \Gamma$ are defined by

$$\begin{aligned} \{P\}.\, q.\, \sigma \ &\ \widehat{=}\ \ (\exists\, \gamma : \Gamma \bullet\ P.\, \sigma.\, \gamma\ \wedge\ q.\, \gamma) \\ [P].\, q.\, \sigma \ &\ \widehat{=}\ \ (\forall \gamma : \Gamma \bullet\ P.\, \sigma.\, \gamma\ \Rightarrow\ q.\, \gamma) \end{aligned}$$

When started in a state σ, both $\{P\}$ and $[P]$ choose, if possible, a new state γ such that $P.\, \sigma.\, \gamma$ holds. The difference between them is that $\{P\}$ chooses "angelically", i.e., trying at the same time to establish the desired postcondition q, while $[P]$ chooses "demonically", i.e., trying to avoid establishing q, if possible. If no such state exists, then $\{P\}$ aborts, whereas $[P]$ behaves as **magic**.

For demonic updates, we also use more a common syntax of nondeterministic assignment $x := x'.P$ where x is a list of state variables that can be changed during execution of this statement. The relation P describes relationship between new values of x (denoted x') and the values of program variables before execution. For example, the nondeterministic assignment $x := x'.x' \geq y$ states that the variable x can get any value which is greater than or equal to the value of y. The variables that are not mentioned in the left hand side of a (nondeterministic) assignment statement are assumed to stay unchanged. Therefore, in the program state consisting of two variables x and y the nondeterministic assignment $x := x'.\, x' \geq y$ is actually a shorthand for $x, y := x', y'.\, (x' \geq y) \wedge (y' = y)$.

The sequential composition of statements $S_1 : \Sigma \mapsto \Gamma$ and $S_2 : \Gamma \mapsto \Delta$ is modeled by their functional composition, for $q : \mathcal{P}\Delta$,

$$(S_1; S_2).\, q \;\; \widehat{=} \;\; S_1.\, (S_2.\, q)$$

Conjunction $\sqcap$ and disjunction $\sqcup$ of predicate transformers are defined pointwise:

$$\begin{array}{lcl} (\sqcap i \in I \bullet S_i).\, q & \widehat{=} & (\cap i \in I \bullet S_i.\, q) \\ (\sqcup i \in I \bullet S_i).\, q & \widehat{=} & (\cup i \in I \bullet S_i.\, q) \end{array}$$

Both conjunction and disjunction of predicate transformers model nondeterministic choice among executions of S_i. Conjunction models *demonic* nondeterministic choice in the sense that nondeterminism is uncontrollable and each alternative must establish the postcondition. Disjunction, dually, models *angelic* nondeterministic choice, where choice is free and aimed at establishing the postcondition. For the special case $I = \{1, 2\}$, we have binary choice operators denoted as $S_1 \sqcap S_2$ and $S_1 \sqcup S_2$.

The predicate transformers (of the given type $\Sigma \mapsto \Gamma$) ordered by the refinement relation form a complete lattice. Angelic and demonic choices are the join and meet operations on lattice elements. The top element in this lattice is **magic** and the bottom element is **abort**.

The *conditional* statement is defined as the demonic choice of guarded alternatives:

$$\textbf{if } g \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \;\; \widehat{=} \;\; [g]; S_1 \;\sqcap\; [\neg g]; S_2$$

Iteration is defined as the least fixpoint of a function on predicate transformers with respect to the refinement ordering:

$$\textbf{do } g \rightarrow S \textbf{ od} \;\; \widehat{=} \;\; (\mu X \bullet \textbf{if } g \textbf{ then } S; X \textbf{ else skip fi})$$

Here μ denotes the least fixpoint operator.

According to the theorem of Knaster-Tarski[100], a monotonic function has a unique least fixpoint in a complete lattice. Predicate transformers form a complete lattice, and, therefore, the least fixpoint of monotonic functions on predicate transformers always exist and is unique. The least fixpoint operator μ is defined according to its characterisation given in the Knaster-Tarski theorem:

$$\mu f \;\widehat{=}\; (\sqcap x \mid f.x \sqsubseteq x \bullet x)$$

where f is a monotonic function on predicate transformers.

The definition presented above is difficult to use in practice. Instead, we use the following characteristic properties of the least fixpoint operator:

$$\begin{array}{rl} f.(\mu f) = \mu f & \mu \textit{ folding } (\textit{unfolding}) \\ f.x \sqsubseteq x \;\Rightarrow\; \mu f \sqsubseteq x & \mu \textit{ induction} \end{array}$$

The statement $\mathbf{block}\, p\, S$ introduces a block with a new local state component initialised according to the predicate p and a block body S working on the extended state. The new state component is added as the first component of the state. A block statement is defined in the following way:

$$\mathbf{block}\, p\, S \;\widehat{=}\; \mathbf{begin}\, p;\; S;\; \mathbf{end}$$

where the block beginning and end operators are modeled by demonic and functional updates respectively:

$$\begin{array}{rcl} \mathbf{begin}\, p & \widehat{=} & [\lambda u \bullet\; \lambda(x, u') \bullet\; p.(x, u) \wedge (u = u')]; \\ \mathbf{end} & \widehat{=} & \langle \lambda(x, u) \bullet\; u \rangle \end{array}$$

The **begin** operator introduces a new state component and initialises it according to the predicate p. Global variables remain unchanged as indicated by the conjunct $u = u'$. The **end** operator ends the block by removing the local component from the state. For blocks, we also use the following syntax – $|[\mathbf{var}\; x \mid p \bullet\; S]|$. Here x is an explicit list of local variables that are introduced by the block.

Finally, the parallel composition of two statements $S_1 \| S_2$ of the type $\Sigma_1 \times \Sigma_2 \mapsto \Gamma_1 \times \Gamma_2$ models parallel execution of the statements $S_1 : \Sigma_1 \mapsto \Gamma_1$ and $S_2 : \Sigma_2 \mapsto \Gamma_2$ on disjoint state spaces. It is defined in the following way[14]:

$$(S_1 \parallel S_2).q.(\sigma_1, \sigma_2) \;\widehat{=}\; (\exists q_1\; q_2 \bullet\; (q_1 \times q_2 \subseteq q) \;\wedge\; S_1.q_1.\sigma_1 \;\wedge\; S_2.q_2.\sigma_2)$$

where $\times$ denotes the product operation on predicates defined as follows:

$$(q_1 \times q_2).(\sigma_1, \sigma_2) \;\widehat{=}\; q_1.\sigma_1 \;\wedge\; q_2.\sigma_2$$

Intuitively, this definition means that $S_1 \| S_2$ establishes postcondition $q : Pred(\Gamma_1 \times \Gamma_2)$ in initial state (σ_1, σ_2), if there exists a subset $q_1 \times q_2$ of q such that S_1 establishes q_1 in σ_1 and S_2 establishes q_2 in σ_2 independently.

2.6 Data Refinement

Data refinement is a special case of refinement where abstract data structures are replaced with more concrete ones. Typically, "more concrete" means more easily or efficiently implemented. Data refinement is formally defined in terms of ordinary refinement as follows. For statements $S : Ptran(\Sigma)$ and $S' : Ptran(\Sigma')$, let $R : \Sigma' \leftrightarrow \Sigma$ be a relation between the state spaces Σ and Σ'. The relation R is called an *abstraction* relation. According to [17], the statement S is said to be data refined by S' via relation R, denoted $S \sqsubseteq_R S'$, if

$$\{R\}; S \;\sqsubseteq\; S'; \{R\}$$

Alternative equivalent characterisations of data refinement (for monotonic predicate transformers) using the inverse relation R^{-1} are then

$$S; [R^{-1}] \;\sqsubseteq\; [R^{-1}]; S' \qquad S \;\sqsubseteq\; [R^{-1}]; S'; \{R\} \qquad \{R\}; S; [R^{-1}] \;\sqsubseteq\; S'$$

Data refinement defined in this way is called *forward data refinement*.

The calculational approach to data refinement[38, 108] allows us to avoid inventing the new program in a data refinement step. The calculational rules for different programming constructs directly yield a new concrete program for a given abstract program and an abstraction relation. In the thesis we use techniques of calculational data refinement for transformation of blocks where abstract local variables are replaced with concrete ones. In other words, we are interested in refinement of the form:

$$|[\,\mathbf{var}\; a \mid ini.\; S\,]| \;\sqsubseteq\; |[\,\mathbf{var}\; c \mid \mathcal{DI}_R(ini).\; \mathcal{D}_R(S)\,]|$$

Here $\mathcal{DI}_R$ and $\mathcal{D}_R$ are functions for calculation of the new initialisation predicate and the new body of a concrete block. They are defined in the following way:

$$\begin{aligned} \mathcal{DI}_R\,(ini) &\;\widehat{=}\; (\lambda\sigma' \bullet \exists\sigma \bullet\; R.\sigma'.\sigma \;\wedge\; ini.\sigma') \\ \mathcal{D}_R\,(S) &\;\widehat{=}\; \{R\}; S; [R^{-1}] \end{aligned}$$

In order to calculate the expression $\mathcal{D}_R(S)$, the refinement calculus provides a number of special calculational rules for different statements and constructs of the specification language.

2.7 History

The origins of stepwise refinement method were presented in the works of Dijkstra and Wirth[34, 106], and in the program transformation approach of Gerhart[40] and Burstall and Darlington[23].

The stepwise refinement method[34, 106] was formalised into the refinement calculus by Back in his thesis[4], using the weakest precondition semantics proposed by Dijkstra[35]. The basic ideas of the refinement calculus such as an introduction of the program refinement relation, the emphasis on total correctness, modelling program statements as predicate transformers, using specifications as primitive program statements etc. were formulated in Back's seminal work.

Morgan[82, 81, 83] and Morris[86] have extended the Back's original formulation with different specification statements. Morgan has proposed miraculous statements (assumptions in the refinement calculus) in [82]. Morgan together with Gardiner[82, 38, 39] have studied data refinement techniques (the idea of data refinement was originally introduced by Hoare[56]).

In his influential book Morgan[84] has presented a more practical approach for the refinement calculus with pre-post specification statements, a simple set of refinement rules and checklists, and syntactical treatment of the program state and variables.

In later works Back and von Wright[11, 12, 13, 15] have reengineered the mathematical basis for the refinement calculus, studying lattice-theoretical properties of program refinement in higher-order logic. This have made the theory much more general and simple at the same time.

In recent years the refinement calculus theory has been applied to new areas such as stepwise derivation of parallel and reactive programs[7, 10], object-oriented programs[89, 101], and probabilistic programs[85].

Chapter 3

Mechanisation of the Refinement Calculus

3.1 Introduction

Proofs of program refinement, even for moderately sized programs, can become long and involved, full of complex details. The terms that one has to work with in proofs are usually quite big. Therefore, they can be difficult to manage and error-prone. Effects of even simple typing errors can propagate throughout the development causing annoying "backtracking" when noticed. In the case of more realistic programs, proofs are obviously much longer and, therefore, the possibility of errors is even bigger.

This situation naturally calls for some kind of automation. Assistance may be provided by a tool which records and maintains the proof as it is constructed step by step. Such a tool can provide high accuracy needed in complex detailed mathematical proofs. This way, for example, errors resulting from bad typing can be effectively eliminated. Furthermore, mechanised logics cannot rely upon any "hand-waving" over matters of syntax or semantics. Thus, a higher level of precision is needed which encourages a clearer analysis of the system. Finally, the use of tools facilitates the practice of formal methods by increasing their scalability since the automation provided allows us to handle the increasing detail, and, therefore, larger systems can be addressed.

Tool support for program refinement can increase our confidence in the soundness of our refinement theory and in the correctness of our program derivations. However, a refinement tool should satisfy a number of requirements. It should be expressive enough to represent all statements in our

specification and programming language, it must allow us to use standard results in classical mathematics, and it should be flexible enough to support program development using the stepwise refinement paradigm. The Higher Order Logic (HOL) theorem prover[41, 43] is such a mechanical proof assistant. It is an interactive theorem proving environment for higher-order logic, based on the LCF approach for general theorem proving developed by R.Milner[78].

In this chapter we give brief description of the HOL theorem prover and its most important features, and explain how it could be extended to be an effective and convenient tool for derivation of provably correct programs.

3.2 The HOL Proof Assistant

Why HOL? The HOL proof assistant mechanizes a higher-order logic, and provides an environment for defining theories and proving statements about them. HOL is secure in the sense that only true theorems can be proven, and this security is ensured at each point that a theorem is constructed.

HOL is called a proof assistant because it does not attempt to automatically prove theorems but rather provides an environment to the user to enable him/her to prove theorems. It is, however, powerfully programmable. The user is free to construct programs which automate whatever proving strategy he/she prefers.

HOL has been applied in many areas. The first and still very popular area is hardware verification, where it has been used to verify correctness of several microprocessors. In the area of software, HOL has been applied to Hoare logic[42], Lamport's Temporal Logic of Actions (TLA)[68], Chandy and Misra's UNITY language[2], Hoare's CSP[28], Milner's CCS and π-calculus[76, 79].

HOL is one of the oldest and most mature mechanical proof assistants available. Many other proof assistants have been introduced more recently that in some ways surpass HOL, but HOL has one of the largest user communities and history of experience. Moreover, the extensibility of the HOL logic and the programmability of its proving environment makes the HOL system a good choice when implementing your own mechanical tool. We therefore consider it most suitable for this work.

HOL Logic The logic of the HOL system is a higher-order logic which was briefly described in Section 2.3. Recall that this logic can be extended

in two ways: by the definition of new types and type constructors, and by the definition of new constants.

The HOL logic is based on eight rules of inference and five axioms. These are the core of the logical system. Each rule is sound, so one can only derive true results from applying them to true theorems. As the HOL system is built up, each new inference rule consists of calls to previously defined ones. Therefore the HOL proof system is fundamentally sound, in that only true results can be proven.

The HOL system provides the user a logic that can be easily extended. These extensions are organized into units called *theories*. A HOL theory is closely related to the familiar notion of theories in mathematics and logics, thus theories consists of a number of constants, definitions and axioms. Furthermore, theories usually contain theorems that have been proven (within HOL) using the axioms and previously proven theorems. Once created, a theory can be saved into a theory file on the disk and be loaded and extended later on.

In the implementation of the HOL logic, there is also the possibility to introduce new axioms. However, in this case the user bears complete responsibility for possible inconsistencies which may be introduced. In our use of HOL system, we restrict ourselves to never using the ability to assert new axioms. This style of using HOL is called "conservative extension". In a conservative extension, the security of HOL is not compromised, and hence the basic soundness of HOL is maintained.

HOL Meta Language When the HOL system is started, it presents to the user an interactive programming environment using the programming language SML, the *Meta Language* of HOL. Terms and theorems of the HOL logic are represented by the corresponding SML data types **term** and **thm**. SML functions are provided to construct and deconstruct HOL terms. Theorems, however, can only be created by means of a HOL proof, by using inference rules provided by the HOL system. Thus the security of HOL is maintained by implementing **thm** as an abstract data type in SML.

Additional rules, called *derived rules of inference*, can be written as new SML functions. Each rule typically takes a number of theorems and/or terms as arguments and produces a theorem as a result. This method of producing new theorems by calling functions is called *forward proof*.

The HOL system also supports *backward* proof, where one sets up a goal to be proved, and then breaks that goal into a number of subgoals, each of which can be reduced further, until every subgoal is resolved, at which point

the original goal is established as a theorem. At each step, the operation that is applied is called in HOL a *tactic*, which is a SML function of a particular type. The effect of of applying a tactic is to replace a current goal with a set of subgoals which if proven are sufficient to prove the original goal. Intuitively, a tactic can be understood as the inversion of an inference rule.

Functions in SML are provided to create new types, make new definitions, prove new theorems, and store the results into theories on disk. These may then be used to support further extensions. In this incremental way a large system may be constructed.

Formal Embeddings of Languages In order to reason formally about systems or languages in a theorem prover, one should represent or "embed" them into the underlying logic of the tool. There are two styles of embedding used in the HOL system: *shallow embedding* and *deep embedding*[21].

In a shallow embedding, each language construct is introduced as a separate HOL constant which defines a function directly denoting the construct's semantic meaning. Thus, terms of the embedded language are identified with corresponding terms of the HOL logic. However, the interpretation of the abstract syntax of a language is outside of the logic. That makes it impossible to reason about such syntactic notions as, for example, substitution.

In a deep embedding, a language is introduced into the HOL logic as a new HOL type. The new type inductively defines the given abstract syntax of a language to be embedded. The semantics of a language is defined as a separate function interpreting each syntactic language phrase in a structural way. This allows us to reason within HOL about the semantics of purely syntactic manipulations.

In the choice between deep and shallow embedding, there is a trade-off between expressiveness and ease of use. A deep embedding may allow more meta-properties to be stated and proved about a language. However, it is much easier to work with the semantics of shallowly embedded languages since one can can work on the semantics directly. That is especially true in cases when one needs to extend a language incrementally. In a shallow embedding this merely amounts to defining a new semantic abbreviation and proving the necessary properties about it. In a deep embedding, however, one needs to redefine the underlying HOL type and its interpretation (semantics) function as well as to re-prove all old results.

3.3 The Refinement Calculus as a Theory of the HOL Theorem Prover

The expressiveness of the HOL logic makes it a good choice for formalising program refinement. The weakest precondition semantics is easily modelled in higher-order logic. Program statements as predicate transformers can be defined as constants in the HOL system. As a result, it is possible to safely define the predicate transformer semantics of our programming language as a conservative extension of higher-order logic. Overall, the HOL logic has sufficient abstraction mechanisms to specify both complex refinement problems and the data-types needed to solve them.

The formalisation of the refinement calculus in higher order logic was implemented as a HOL theory by von Wright[108]. This mechanised theory forms the basis for our work.

The HOL theory of refinement is a shallow embedding. The choice of a shallow embedding was made for the following practical reasons. Shallow embedding makes the embedded programming language strongly typed by inheriting the HOL type system rather than constructing its own. It also allows easy reuse of existing HOL theories describing numbers, arrays, lists, and other data types for the types of program variables. Finally, the embedded programming language can be easily redefined or extended with new constructs which is an important factor while developing an experimental theory.

In this theory the program state is modelled as a polymorphic type variable α or β. For any given program, this type is instantiated to a product where each component corresponds to some program variable. If, for example, a program works on two natural number variables `x` and `y` and a boolean type variable `b`, then the state space has the type `num#num#bool` where # is the HOL product type constructor. The names of program variables are handled using a `let`-construction[1]. Thus, in our example the term representing the program is of the form

```
let x = FST in let y = FST o SND in let b = SND o SND in ...
```

Therefore, program variables are projection functions; they indicate positions in the state tuple. The `Typewriter` font we are using here indicates that a particular expression (term, type, theorem, inference rule etc.) is associated with the HOL system. We use this font convention throughout the thesis.

[1] In the HOL system the term `let x = y in t` stands for the functional application (λx.t) y.

Mathematical structures used in the theory include state predicates (functions $\alpha \rightarrow bool$), state relations (functions $\alpha \rightarrow \beta \rightarrow bool$), state functions (functions $\alpha \rightarrow \beta$) and predicate transformers (functions $(\beta \rightarrow bool) \rightarrow (\alpha \rightarrow bool)$). The latter are used to model program statements as functions that map postconditions to preconditions.

Here we only show HOL definitions that are used later in the thesis. We start with operations on predicates which are defined by lifting the corresponding operations on truth values.

```
⊢def true = (λv. T)
⊢def false = (λv. F)
⊢def ∀p q. p imp q = (λv. p v ⇒ q v)
⊢def ∀p q. p andd q = (λv. p v ∧ q v)
⊢def ∀p q. p or q = (λv. p v ∨ q v)
⊢def ∀p. not p = (λv. ¬(p v))
```

Greatest lower bound (generalized conjunction) and least upper bound (generalized disjunction) over sets of predicates are defined as follows:

```
⊢def ∀P. glb P = (λs. ∀p. P p ⇒ p s)
⊢def ∀P. lub P = (λs. ∃p. P p ∧ p s)
```

Universal implication (subset) relation on predicates is defined in the following way:

```
⊢def ∀p q. p implies q = (∀v. p v ⇒ q v)
```

Each statement of our language is defined in this theory according to its weakest precondition semantics given in the previous chapter. For example, the assignment statement has the following definition:

```
⊢def ∀f q. assign f q =  (λs. q(f s))
```

where `f` is a state function of type $\alpha \rightarrow \beta$, `q` is a postcondition and `s` is an initial state.

Other commands are defined similarly.

```
⊢def ∀q. skip q = q
⊢def ∀q. abort q = false
⊢def ∀q. magic q = true
⊢def ∀p. assert p q = p andd q
⊢def ∀b q. guard b q = b imp q
⊢def ∀P q. nondass P q = (λv. ∀v'. P v v' ⇒ q v')
⊢def ∀c1 c2 q. (c1 seq c2) q = c1 (c2 q)
⊢def ∀C q. Dch C q = glb (λp. ∃c. C c ∧ (p = c q))
⊢def ∀C q. Ach C q = lub (λp. ∃c. C c ∧ (p = c q))
⊢def ∀g c1 c2 q. cond g c1 c2 q = (g andd c1 q) or (not g andd c2 q)
⊢def ∀g c. do g c = mu (λx. cond g (c seq x) skip)
⊢def ∀p c q. block p c q = (λu. ∀x. p (x,u) ⇒ c (λv. q (SND v)) (x,u))
```

Here `nondass` is the nondeterministic assignment (demonic update) statement, `seq` is the infix operator denoting sequential composition of two statements, and `Dch` and `Ach` are the demonic and angelic choice operators defined over sets of statements.

The least fixpoint operator `mu` is defined according to its characterisation given in Section 2.5:

```
⊢def ∀f. mu f = Dch (λc. monotonic c ∧ (f c ref c))
```

In this definition we explicitly state that we consider only monotonic predicate transformers.

We also use the following characteristic properties of this operator:

```
⊢ ∀f. regular f ⇒ (f (mu f) = mu f)
⊢ ∀f c. monotonic c ∧ (f c ref c) ⇒ mu f ref c
```

where function regularity is defined in the following way:

```
⊢def regular f =
       monotonic c ∧ monotonic c′ ∧ c ref c′ ⇒
       f c  ref  f c′
```

The regularity property means that the restriction of function `f` to monotonic predicate transformers should be monotonic with respect to the refinement relation.

In our proofs, we use the fact that all statements of our language are monotonic and conjunctive. In HOL, monotonicity and conjunctivity properties are defined as follows:

```
⊢def monotonic c    =  (∀p q. p implies q  ⇒  (c p) implies (c q))
⊢def conjunctive c  =  (∀P. c (glb P) =  glb(λq. ∃p. P p ∧ (q = c p)))
```

However, since the HOL type that we are using for modelling statements also includes non-monotonic and non-conjunctive predicate transformers, assumptions about statement monotonicity and/or conjunctivity must be stated explicitly in HOL definitions and theorems.

The refinement relation on predicate transformers is defined in the following way:

```
⊢def ∀c1 c2. c1 ref c2 = (∀q. c1 q implies c2 q)
```

The refinement relation is then easily proved to be a partial order. A large number of useful refinement rules can be proved directly from the definitions. For example,

```
⊢ ∀p p'. p implies p'  ⇒  (assert p) ref (assert p')
```

This theorem states that the assertion statement is monotonic with respect to its predicate parameter.

A correctness assertion (correctness triple) for some statement `c` is defined as follows:

```
⊢def correct p c q  =  p implies c q
```

where `p` is a precondition and `q` is a postcondition.

Parallel execution of two statements on disjoint state spaces has the following definition:

```
⊢def ∀c1 c2 q s1 s2.
     par c1 c2 q (s1,s2) =
       (∃q1 q2.
          (∀s1' s2'. q1 s1' ∧ q2 s2' ⇒ q (s1',s2')) ∧
          c1 q1 s1 ∧ c2 q2 s2)
```

Note that the program state in the definition is modelled as a pair $(\mathtt{s1}, \mathtt{s2})$.

In the thesis we make use of a special kind of parallel execution called *lifting*. Lifting models parallel execution of the form $skip \| c$, where c is the statement operating on the second part (projection) of the state. The lift operation is defined in the following way [14]:

```
⊢def lift c q u = c (λv. q (FST u,v)) (SND u)
```

where `q` is a postcondition, `u` is an initial state, and `FST` and `SND` are the HOL operators returning correspondingly the first and the second component of a tuple . The relationship between this definition and the definition of parallel execution is proved as the following theorem:

```
⊢ ∀c. monotonic c ⇒ (lift c = par skip c)
```

The syntax used in the formalisation is sometimes hard to read. Therefore, presenting HOL terms we try as much as possible to use the syntax described earlier rather than the actual syntax described in corresponding HOL definitions. For example, we write

```
⊢ {p};skip ref skip;{p}
```

rather than

```
⊢ (assert p seq skip) ref (skip seq assert p)
```

3.4 Using Window Inference

Supporting program refinement proofs The HOL formalisation of the refinement calculus provides us with the basis for creating a formal framework for program development. However, some additional support for refinement proofs is needed. The problem is that the proof styles supported by the HOL system – forward proof and backward proof – are not convenient for proving program developments in a stepwise refinement fashion.

The stepwise refinement method can be seen as a combination of both proof styles. Externally, a refinement process is a program development in a linear way: $S_1 \sqsubseteq S_2 \sqsubseteq ... \sqsubseteq S_n$ which can be easily implemented as a forward proof. However, the proof of each particular refinement step $S_i \sqsubseteq S_{i+1}$ usually reduces to a refinement proof on some subcomponent T of program S_i:

$$\frac{T \sqsubseteq T'}{S_i[T] \sqsubseteq S_i[T']}$$

which in turn can be reduced to the proof of some property on state predicates or relations. Here the backward proof style is more suitable.

Grundy[49] in his PhD thesis proposed using the HOL Window library (implemented by him) to support refinement proofs. The HOL window library implements a new proof style based on *window inference*. However, he used a simpler relational semantics to model program statements. von Wright in [108] showed how the HOL Window library can also be used for reasoning about programs modelled in the weakest precondition semantics. Below we give a brief introduction to window inference and explain why it is well-suited for refinement proofs.

General idea of window inference Window inference is based on the idea of proofs by contextual transformation proposed by Robinson and Staples[94] and was extended and implemented as a HOL library by Grundy[48].

Window inference is a style of reasoning where the user may transform an expression by restricting attention to a subexpression (which is called "opening a window") and transforming it. A transformation of a subexpression is a transformation of the whole expression provided certain context-dependent monotonicity conditions hold. Also, while transforming a subexpression, the user can use assumptions that are based on the context of the subexpression.

In the window inference style of proof, a user starts with an expression s and transforms it to t such that $s\ R\ t$ holds for some relation R, thus

creating a proof of the theorem $\vdash s\ R\ t$. The relation must be preorder. The number of intermediate steps in the transformation of s to t does not matter, since R is transitive. Each such intermediate transformation step can be presented as an inference rule of the following form:

$$\frac{H \cup H' \ \vdash \ e \ \ R' \ \ e'}{H \ \vdash \ s_k[e] \ \ R \ \ s_k[e']}$$

Note that the relation in a subderivation (R') can be different from the relation of the main derivation (R). H' here denotes a list of additional assumptions of a subderivation which are based on the context in which the subexpression e occurs in the main expression $s_k[e]$. This proof decomposition process can be continued by focusing on some subexpression of e and, as a result, starting a new subderivation and so on.

It is easy to see that the window inference style of proof directly corresponds to the way in which refinement steps are carried out. In refinement proofs the relation to be preserved is program refinement. Since it is a preorder, the refinement relation can be used in the window inference system. The usual technique for proving program refinement is to focus on some program component and to refine it using the fact that the context is monotonic with respect to the refinement relation. That is exactly how the window inference mechanism works.

The idea of using the contextual transformation method for program refinement is not new. For example, a similar approach using refinement diagrams was proposed by Back[8]. However, his approach is more restrictive since it does not allow use of different relations in subderivations.

HOL Window Inference As indicated above, the window inference style of reasoning was implemented as a HOL library by Grundy. Within the HOL window inference system, reasoning is conducted with a stack of windows. Each window has a *focus* (the expression to be transformed), a set of formulae Γ that can be assumed true in the context of the window (the assumptions), and a relation R that must be preserved. It can also have a set of goals (conjectures or proof obligations).

By default, the window inference system supports three relations – equality, forward implication and backward implication, but the user can add new relations to the system. New relations are added by providing theorems about their reflexivity and transitivity.

In order to transform the focus in the window inference system, the user must provide special ML functions called *transformation rules*. A transformation rule takes the focus s and the current relation R as arguments and

returns the theorem that $s\,R\,t$ holds for some t that has been computed by the rule. The system then automatically transforms the focus s to t.

When opening a window on some subterm of the current focus, the system uses the information that the focus is monotonic in the position where the subterm occurs. This information must be provided by general inference rules for monotonicity. These rules are called *window rules*, and the system keeps them in a database together with information about applicability conditions. The user does not have to know names or other details of these rules since the choice of a suitable rule is done automatically. When needed, the system chains several window rules on behalf of the user, allowing to focus deep inside a term in a single window opening.

After opening the window on a subterm of the focus, the system starts a subderivation transforming the subterm while preserving some relation. After finishing the subderivation (closing the window), the system uses the monotonicity inference rule to prove the theorem for transforming the focus of the main window. Finally, the system transforms the focus according to the proved theorem.

Refinement under window inference In order to use the HOL refinement calculus formalisation together with the window inference library, one needs to do a number of steps. This is explained in great detail by von Wright in [108]. First, one needs to prove that the refinement relation is a preorder. Second, a number of window rules have to be added. The two rules presented below are given as examples. The first rule shows how we can refine the left statement of the sequential composition of two statements by first opening a window on it.

$$\frac{\vdash S_1 \sqsubseteq S_1'}{\vdash S_1; S_2 \sqsubseteq S_1'; S_2}$$

The second rule allows us to focus and transform the relation of the demonic update (nondeterministic assignment) statement.

$$\frac{\vdash \forall s\ s' \bullet R.\,s.\,s' \Leftarrow Q.\,s.\,s'}{\vdash [R] \sqsubseteq [Q]}$$

Note that in this rule the relation to be preserved in the subderivation is different from the one in the main derivation.

All window rules are based on the appropriate HOL theorems expressing monotonicity properties of programming language constructs. For the rules given above, the following theorems are proved beforehand:

```
⊢ ∀c c1 c2. c1 ref c2 ⇒ c1;c ref c2;c
⊢ ∀P Q. (∀s s'. Q s s' ⇒ R s s') ⇒ (nondass P) ref (nondass Q)
```

Furthermore, a number of theory-specific transformation rules have to be added to the window inference system. These rules are used to transform a given focus. For example, the transformation rule `Merge` can be used to merge two assignment statements into one. The rule expects the focus to be of the form $\mathtt{x} := \mathtt{e}; \mathtt{y} := \mathtt{e}'$ where `x` and `y` are lists of program variables. As a result, the rule produces a theorem of the form $\mathtt{x} := \mathtt{e}; \mathtt{y} := \mathtt{e}' \ \mathtt{ref}\ \mathtt{z} := \mathtt{e}''$ and then uses the window inference system to transform the focus according to the proved theorem.

3.5 The Refinement Calculator Tool

Need for graphical interface The HOL system is very useful as a proof assistant when working with general concepts of program refinement. However, the standard command line interface of HOL is not very convenient for transformational reasoning about programs in the style of window inference.

In window inference, the user indicates what transformation is desired and provides information describing the path to the subterm to be transformed. The path indicates the position of the subterm in the abstract syntax tree of the current focus. The fact that subterms have to be identified by their paths is a major inconvenience. Computing paths can be tedious for an ordinary user, especially when terms are large and contain a mixture of prefix and postfix operators. By providing a graphical user interface (GUI), it is possible to automate this process, allowing the user to select data by simply pointing and clicking with the mouse.

The Refinement Calculator The Refinement Calculator tool[70, 24] was developed in order to support the derivation of provably correct programs within the refinement calculus framework. This tool provides a graphical user interface for transformational reasoning in the style of window inference. It consists of a number of layers built on top of the HOL system and its window Library. These layers include the HOL formalisation of the refinement calculus, transformation rules for program refinement and an X Windows based graphical user interface.

The latter layer has its own name – TkWinHOL[70]. TkWinHOL serves as a graphical frontend to the HOL window Library. The GUI provided improves the usability of the HOL window Library and the HOL theory of the refinement calculus considerably by providing visually-based access

to subterms and context assumptions as well as a programming language syntax, and menus and dialogue boxes for the application of transformations. TkWinHOL is a tool in its own right and can be used to develop proofs under window inference. It was, however, designed with the intension of being used as a basis for the Refinement Calculator.

While developing the Refinement Calculator, the emphasis has been on making the tool easy to customise for the needs of particular HOL theories (like program refinement or lattice theory). In such specific theories one usually wants to use higher-level notation (concrete syntax) for the interaction with HOL. This is achieved in the Refinement Calculator by adding a theory specific parser and pretty-printer. That makes it possible, when developing refinement proofs, to use a more common programming language syntax rather than the syntax used in the underlying HOL theory.

The Refinement Calculator tool has been already applied for derivation of simple programs[25], program data refinement[95], development of reactive systems[71].

The basic appearance of the Refinement Calculator When the Refinement Calculator is started, three windows appear on the screen. The first one is a simple text editor that offers a small set of menu commands for file and edit operations. The second window (the HOL window) is similar to a terminal window. Commands can be typed directly (or selected by a mouse) to the HOL window and executed. The text string is sent verbatim to the HOL process running in the background and the HOL reply is presented in the HOL window.

The most important is the third window called the *focus window*. This window displays the status of the current window stack. As seen in Figure 3.5, the window is divided into parts. At the top of the window there is a menubar from which the user can select operations that transform the current focus. Below the menubar there is a field that presents the relation that is currently preserved by the window inference system.

The top subwindow presents a pretty-printed version of the current focus. Every time a transformation is applied to the current (syntactic) focus, the interface sends a command to SML which performs the transformation on the semantic level (using window inference) and then computes (using the pretty-printer) the syntactic form of the new focus. The new focus is returned to the interface and presented on the screen.

If the user selects some subexpression of the current focus with the mouse and presses a special window opening button, then the system computes the

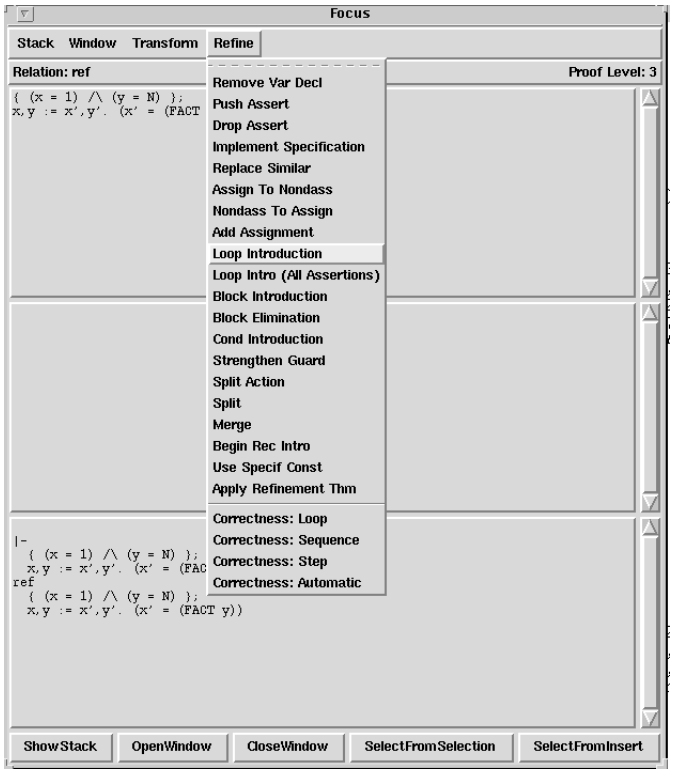


Figure 3.1: The focus window of the Refinement Calculator

path to the subexpression and executes the appropriate window opening command, and a subderivation starts. By closing a window the user ends the subderivation and the focus is transformed according to the appropriate monotonicity rule.

The middle subwindow presents context assumptions that can be used when transforming the current focus. What assumptions should be presented are determined by appropriate window rules. While opening a window on some subexpression, context assumptions (if any) are automatically generated and added into this window.

Some transformation rules are conditional – they can be applied provided that certain conditions hold. Such conditions in the window inference system are called *conjectures* or *proof obligations*; they should be discharged (proved to be true) before the end of a derivation. Proof obligations of the current focus are also presented in the middle subwindow. Clicking on a proof

obligation and selecting the Establish command from the Window menu, the user can start a subderivation transforming the proof obligation into truth under the backward implication relation.

Finally, the bottom subwindow contains the theorem that was actually proved in the current derivation. The theorem is of the form `s R t` where `R` is the relation to be preserved.

3.6 Extensions of the Refinement Calculator

The Refinement Calculator was developed to be easily extended. New functionality can be added by loadable libraries called *extensions*. An extension includes HOL theories, SML code for new transformation and window rules, and descriptions of new menu choices and their bindings to particular HOL commands.

Though the Refinement Calculator was primarily designed to support program refinement proofs, it can also be used as a plain TkWinHOL (the graphical interface to window Library) to support any logical transformations in the style of window inference. The extensions that are not directly related to program refinement are General Logic and Lattice. The General Logic extension adds general logic transformation rules for work with boolean expressions, and the Lattice extension allows to use abstract lattice properties in various concrete domains that have been proved to be lattices.

The current refinement-oriented extensions are:

1. Refinement, basic program refinement,
2. Context, handling context information in programs,
3. Dataref, calculational data refinement of blocks,
4. Correctness, verification of total correctness conditions,
5. Procedure, working with procedures and procedure calls.

The Context and Procedure extensions are explained in detail in Chapters 5 and 6. Here we briefly describe the other three extensions.

The Refinement extension provides basic transformations for program refinement such as merging/splitting of assignment statements, introduction/elimination of a block, introduction of a while-loop etc. The complete list of transformation rules may be seen in Figure 3.1. The detailed descriptions of these rules can be found in [26].

The data refinement extension Dataref[95] supports transformation of local blocks using calculational data refinement approach briefly described in Section 2.6. The transformation is implemented using a function $\mathcal{D}_R$ such that

$$|[\mathbf{var}\ a \mid ini.\ S\ ]| \sqsubseteq |[\mathbf{var}\ c \mid (\exists a.\ R \wedge ini).\ \mathcal{D}_R(S)\ ]|$$

where the abstract statement S refers to the variables v, a, and the concrete statement $\mathcal{D}_R(S)$ to v, c. Here variables v are the variables that are not affected by the transformation. $R(a, c, v)$ is an abstraction relation, relating the abstract (i.e., (a, v)) and concrete (i.e., (c, v)) variables. The function $\mathcal{D}_R$ is defined recursively over the structure of programming language notation.

Using the Correctness extension, the user can prove correctness goals, i.e., that certain program (or program fragment) is correct with respect to a given precondition and a postcondition. The user can start a separate correctness derivation, or to prove a correctness goal generated as a proof obligation (for example, after a loop introduction). The relation to be preserved in a derivation is the backward implication. The extension presents a number of transformation rules for reducing a correctness goal. After several reduction steps the goal is usually transformed into a boolean expression which can be transformed to truth using common logical transformations.

3.7 Conclusions

In this chapter we have described in very general terms the hierarchical infrastructure needed to support mechanical program development in a stepwise refinement fashion. This infrastructure is used as the background of our work presented in this thesis. In the next three chapters we present our implementations of three extensions of the Refinement Calculator (Lattice, Context, and Procedure) in more detail. These extensions demonstrate a variety of different mathematical problems that one has to cope with in order to develop provably correct programs.

Independently of the Refinement Calculator tool described in this chapter, a refinement tool called PRT[29, 30] has been developed by a group at the University of Queensland. PRT is built on top of the Ergo[102] theorem prover which also supports the window inference style of reasoning. The underlying logic of the PRT tool is a purpose-built modal logic. Program statements, predicates, program variables and logic variables are treated as separate syntactic classes in this logic. In some aspects PRT surpasses the Refinement Calculator, like

- better visualisation and management of a proof tree;

- automatic selection of a subset of transformation rules that are applicable to the current focus;
- the use of meta-variables in refinement proofs.

However, the main advantage of the Refinement Calculator comparing to PRT is its better reliability. This is related to the fact that its underlying logic (higher-order logic) can be incrementally extended in a secure way, i.e., by making conservative extensions. The predicate transformer semantics of the programming language used in the Refinement Calculator is described as such a definitional (conservative) extension of higher-order logic. In the modal logic used in PRT, the only way to achieve the same effect is by assertion of new axioms. Therefore, using the Refinement Calculator for refinement proofs gives us a higher degree of confidence in their soundness.

Chapter 4

The Lattice Extension

4.1 Introduction

In this chapter we describe the implementation of the Lattice extension of the Refinement Calculator. This extension provides the necessary infrastructure for using properties of abstract lattices when working with concrete domains that have been proved to be instances of lattices.

What is the motivation behind this work? Working with formalised concepts of program refinement [12, 108], we have encountered structures that are instances of lattices on different abstraction levels of the theory. In this chapter we show how it is possible to create a single abstract theory of lattices in HOL, which can then be instantiated in different ways and used efficiently when reasoning within these structures. Thus we prove properties of lattices once and for all in the abstract theory. When one has an instance of a lattice (i.e., when a certain structure is shown to satisfy the defining properties of lattices), then the theorems are easily shown to hold for this instance as well. The basic principles of abstract theories are taken from Gunter's work[50] on abstract group theory in HOL.

In this chapter we also explain how lattice properties can be used in derivations using the window inference style of formal reasoning[48]. We extend window inference with new transformation and window rules for working with lattices. The transformation rules for lattices allow us to introduce and eliminate lattice constructs in the style of natural deduction, while the window rules for lattices allow us to focus on some subcomponent and do local transformations using context monotonicity properties of lattice constructs.

Furthermore, we show how our transformation rules and other infras-

tructure for transformational reasoning work together with the Refinement Calculator tool. The implementation is extensible; users can add new instances of lattices and all the existing transformation rules are then available for the added structures.

4.2 Formalising Lattice Theory in HOL

We start by considering what lattices are and how they can be formalized in HOL. The basic ideas of formalising lattice theory in HOL are described in more detail by von Wright[107]. We reuse most of the basic definitions and some of the theorems presented in this work. Our additions include some new properties of general meets and joins, and also of fixpoint constructs proved as HOL theorems. In the second part of this section, we present our implementation of inference rules for lattices that allow us to use abstract lattice properties conveniently in the style of natural deduction.

4.2.1 Doing Algebra in HOL

Theories of algebra generally assume an underlying set of anonymous elements and operators that work on these elements. Examples of algebraic theories are the theories of groups and lattices. Group theory in HOL is described by Gunter [50], with the underlying set represented by its characteristic function and n-ary operators represented by n-ary (curried) functions. The same approach can be used for other theories of algebra as well.

Consider the theory of posets (partially ordered sets) as an example. Assume that $(A, \sqsubseteq)$ is a poset, where the elements of A belong to an unspecified type α. The partial order $\sqsubseteq$ is formalised as a function $po : \alpha \to \alpha \to bool$ with the following interpretation: if x and y are elements of A then $po\ x\ y$ holds if and only if $x \sqsubseteq y$. In HOL, we define the predicate `POSET` so that `POSET(A,po)` holds if and only if `po` satisfies the properties of reflexivity, antisymmetry and transitivity on A. In HOL the definitional theorem is as follows:

```
POSET_DEF =
  ⊢def POSET(A,po) =
       (∀x. A x ⇒po x x) ∧
       (∀x y. A x ∧ A y ⇒(po x y ∧ po y x ⇒ (x=y))) ∧
       (∀x y z. A x ∧ A y ∧ A z ⇒(po x y ∧ po y z ⇒ po x z))
```

Theorems proved in the theory of posets will generally have an assumption stating that the set under consideration is a poset.

From now on we consistently work with structures where the underlying set is a whole type, i.e., *universal set* $U = (\lambda x : \alpha \bullet T)$. Everything we do could also be done with subsets of U, but this would add membership conditions to almost all definitions and theorems which would make them (even) harder to read.

4.2.2 What is a Lattice?

We show now how basic definitions and properties of lattices are formalised in HOL. We first recall some basic concepts of lattices. Then we explain how to define lattices, meets, joins etc. in the HOL system.

We assume that the reader is familiar with the concepts of partial orders, meets (greatest lower bounds) and joins (least upper bounds).

A partially ordered set $(A, \sqsubseteq)$ is a *lattice* if the meet (greatest lower bound) $x \sqcap y$ and the join (least upper bound) $x \sqcup y$ exist for arbitrary elements x and y in A. If every subset B of the set A has the meet $\sqcap B$ and the join $\sqcup B$ in A we say that $(A, \sqsubseteq)$ is a *complete lattice*. The least (bottom) element of a complete lattice is denoted $\bot$ and the greatest (top) element is denoted $\top$. If $(A, \sqsubseteq)$ is a complete lattice where the following conditions hold for arbitrary $x \in A$ and $B \subseteq A$:

$$x \sqcap (\sqcup B) = (\sqcup y \in B \bullet x \sqcap y)$$
$$x \sqcup (\sqcap B) = (\sqcap y \in B \bullet x \sqcup y)$$

we say that $(A, \sqsubseteq)$ is *infinitely distributive*. Finally, a *(complete) boolean lattice* is an infinitely distributive lattice where every element x has an unique inverse x^{-1} satisfying

$$x \sqcap x^{-1} = \bot \quad \text{and} \quad x \sqcup x^{-1} = \top$$

4.2.3 Definitions

We define lattices in the same way as we defined posets above; the defining theorem states that in order to be a lattice, a set must be a poset and every pair of elements must have a greatest lower bound (meet) and a least upper bound (join):

```
LAT_DEF =
  ⊢def LAT(U,po) =
         POSET(U,po) ∧
         (∀a b. (∃m. po m a ∧ po m b ∧
                   (∀m'. po m' a ∧ po m' b ⇒ po m' m)) ∧
                (∃j. po a j ∧ po b j ∧
                   (∀j'. po a j' ∧ po b j' ⇒ po j j'))
```

The (binary) meet and join operators can now be defined using Hilbert's choice operator ε. The defining theorem for a binary meet is as follows:

```
meet2_DEF =
  ⊢def meet2(U,po) a b =
         (εm. po m a ∧ po m b ∧
           (∀m'. po m' a ∧ po m' b ⇒ po m' m))
```

and a binary join, `join2`, is defined dually. Note that the first argument of the constants `POSET`, `LAT` and `meet2` is a pair `(U,po)` which indicates the lattice under consideration. Note also that the universal set `U` in the definitions is redundant in the sense that it can easily be inferred from the type of the ordering `po`. We left it here only for the sake of clarity.

4.2.4 Basic Properties

The definitions of `meet2` and `join2` are hard to work with in practice, due to the occurrences of the choice operator ε. We want to reason about lattices in the ordinary mathematical way, relying on the characteristic properties of meets and joins: idempotency, commutativity, associativity and absorption, as well as on the identities relating meets and joins to the partial order. Once these properties are proved, we can reason about lattices much as we do in ordinary mathematics.

The proofs of the characteristic properties of the meet and join operators are quite straightforward. We show as examples the theorems about idempotence and commutativity of a binary meet (other properties are defined similarly):

```
meet2_idemp =
   LAT(U,po) ⊢ meet2(U,po) a a = a

meet2_comm =
   LAT(U,po) ⊢ meet2(U,po) a b = meet2(U,po) b a
```

4.2.5 Complete and Boolean Lattices

We define complete lattices, infinitely distributive lattices and boolean lattices using the same principles as above. We show only definitions of complete, distributive and boolean lattices, general meets and bottom elements (general joins and top elements are defined dually):

```
CLAT_DEF =
  ⊢def CLAT(U,po) =
         LAT(U,po) ∧
         (∀B. (∃m. (∀x. B x ⇒ po m x) ∧
                   (∀m'. (∀x. B x ⇒ po m' x) ⇒ po m' m)) ∧
              (∃j. (∀x. B x ⇒ po x j) ∧
                   (∀j'. (∀x. B x ⇒ po x j') ⇒ po j j')))

meet_DEF =
  ⊢def meet(U,po) B =
         (εm. (∀x. B x ⇒ po m x) ∧
            (∀m'. (∀x. B x ⇒ po m' x) ⇒ po m' m))

bot_DEF =
  ⊢def bot(U,po) = (εb. ∀x. po b x)
```

Now we can define infinitely distributive and boolean lattices:

```
DLAT_DEF =
  ⊢def DLAT(U,po) =
         CLAT(U,po) ∧
            (∀a. ∀B.
              (meet2(U,po) a (join(U,po) B) =
                 join(U,po) (λx.∃y. B y ∧ (x = meet2(U,po) a y))) ∧
              (join2(U,po) a (meet(U,po) B) =
                 meet(U,po) (λx.∃y. B y ∧ (x = join2(U,po) a y))))


BLAT_DEF =
  ⊢def BLAT(U,po) =
         DLAT(U,po) ∧
           (∀a. ∃a'. (meet2(U,po) a a' = bot(U,po)) ∧
                     (join2(U,po) a a' = top(U,po)))
```

Inverses are defined in the obvious way (the definition of `BLAT` guarantees that inverses exist in boolean lattices). The HOL-proof that inverses are unique is quite complicated; it involves both the basic properties of meets and joins and the infinite distributivity properties.

4.2.6 Fixpoints

Monotonicity of a function over a complete lattice guarantees the existence of (least and greatest) *fixpoints* which can be very useful for defining recursion and iteration. Recall that the fixpoints of a function $f : \alpha \to \alpha$ are the solutions of the equation $f.x = x$.

The Knaster-Tarski theorem[100] gives the following explicit constructions of the least (μf) and the greatest (νf) fixpoints:

$$\begin{aligned} \mu f &= (\sqcap x \in A \mid f\,x \sqsubseteq x \bullet x) \\ \nu f &= (\sqcup x \in A \mid x \sqsubseteq f\,x \bullet x) \end{aligned}$$

where $(A, \sqsubseteq)$ is a complete lattice and f is a monotonic function on A.

First we define the monotonicity property of functions on lattice:

```
Lmono_DEF =
  ⊢def Lmono(U,po) f =
        (∀x y. po x y  ⇒ po (f x) (f y))
```

Then we define the least fixpoint using the characterisation given above (the definition of the greatest fixpoint is dual):

```
lfix_DEF =
  ⊢def lfix(U,po) f =
        meet (U,po) (λx. po (f x) x)
```

Using these definition we can prove basic properties of fixpoints. Each such theorem will have assumptions stating that the type under consideration is a complete lattice and the function is monotonic on this lattice.

4.2.7 Inference Rules for Lattices

Abstract lattices introduce special constructs to operate with – binary meets and joins, general meets and joins, tops and bottoms. It is convenient to work with them using the style of natural deduction, with special inference rules for introduction and elimination of different lattice constructs. Such rules are useful when the aim is to prove a theorem of the form $\vdash\ t \sqsubseteq t'$ by stepwise transformational reasoning, i.e., by proving first $\vdash\ t \sqsubseteq t_1$, then $\vdash\ t_1 \sqsubseteq t_2$ etc. up to $\vdash\ t_n \sqsubseteq t'$.

Our inference rules for introducing and eliminating lattice constructs differ slightly from the traditional rules for logical connectives. This is because we want to express the rules as properties of the lattice ordering. The names "introduction" and "elimination" here refer to the fact that the specific construct is introduced (or eliminated) when we move from the left to the right hand side of the ordering. The advantages of this approach will become clear in Section 4 where we consider reasoning about lattices using the window Library of the HOL system.

Below we present the inference rules for the basic lattice constructs[15]. For a binary meet we have the following rules:

$$\frac{\Phi \vdash s \sqsubseteq t \quad \Phi' \vdash s \sqsubseteq t'}{\Phi \cup \Phi' \vdash s \sqsubseteq t \sqcap t'} \qquad (\textit{binary } \sqcap \textit{ introduction})$$

$$\vdash t \sqcap t' \sqsubseteq t \qquad \vdash t \sqcap t' \sqsubseteq t' \qquad (\textit{binary } \sqcap \textit{ elimination})$$

For a general meet the rules are as follows:

$$\frac{\Phi, v \in I \vdash s \sqsubseteq t}{\Phi \vdash s \sqsubseteq (\sqcap v \in I \bullet t)} \qquad (\textit{general } \sqcap \textit{ introduction})$$
$\bullet$ v not free in s, I or Φ

$$t' \in I \vdash (\sqcap v \in I \bullet t) \sqsubseteq t[v/t'] \qquad (\textit{general } \sqcap \textit{ elimination})$$
$\bullet$ t' is free for v in t

The rules for joins are dual.

For the least fixpoint we have the following introduction and elimination rules:

$$\frac{\forall i \bullet C_i \sqsubseteq f(\sqcup j \mid j < i \bullet C_j)}{(\sqcup i \in Nat \bullet C_i) \sqsubseteq \mu f} \qquad (\mu \textit{ introduction})$$

$$\frac{f\, x \sqsubseteq x}{\mu f \sqsubseteq x} \qquad (\mu \textit{ elimination})$$

where $C_1, C_2, ...$ is a sequence of lattice elements and f is a monotonic function on complete lattice. The first rule is very useful for recursion introduction on concrete domains that turn out to be complete lattices. We show later that, for example, the do-loop introduction rule can be derived from a specialisation of this rule. The rules for the greatest fixpoint are dual.
For bottom and top the rules are as follows:

$$\Phi \vdash \bot \sqsubseteq t \qquad (\textit{bottom elimination})$$

$$\Phi \vdash t \sqsubseteq \top \qquad (\textit{top introduction})$$

In addition to these rules, there are inference rules expressing monotonicity properties of lattice constructs. For example, the rule stating that a binary meet is monotonic in its left argument is:

$$\frac{\Phi \vdash t \sqsubseteq t'}{\Phi \vdash t \sqcap s \sqsubseteq t' \sqcap s} \qquad (\textit{left monotonicity of binary } \sqcap)$$

Similar rules exist for the right argument of a binary meet as well as for both arguments of a binary join.

For general meets ($\sqcap v \in I \bullet s$), we have monotonicity in the body s and antimonotonicity in the index set I of the argument (as a set, I belongs to a powerset lattice ordered by set inclusion):

$$\frac{\Phi,\ v \in I \vdash s \sqsubseteq s'}{\Phi \vdash (\sqcap v \in I \bullet s) \sqsubseteq (\sqcap v \in I \bullet s')} \quad (\textit{monotonicity of body of } \sqcap)$$
$$\bullet\ v \text{ not free in } s,\ I \text{ or } \Phi$$

$$\frac{\Phi \vdash I \supseteq I'}{\Phi \vdash (\sqcap v \in I \bullet s) \sqsubseteq (\sqcap v \in I' \bullet s)} \quad (\textit{antimonotonicity of index set of } \sqcap)$$

For general joins ($\sqcup v \in I \bullet s$), we have monotonicity in both the body s and the index set I of the the argument.

All these rules are implemented in HOL as SML functions taking as arguments a term (representing the left hand side of the ordering in the conclusion expression) and one or several theorems (hypotheses of the inference rule) and returning the theorem in the conclusion of the rule. For example, the inference rule for the left monotonicity of the binary meet operator is implemented as the SML function `MEET2_MONO_LEFT` which takes a term of the form $\mathtt{t} \sqcap \mathtt{s}$ and a theorem of the form $\vdash\ \mathtt{t} \sqsubseteq \mathtt{t}'$ and produces the new theorem of the form $\vdash \mathtt{t} \sqcap \mathtt{s} \sqsubseteq \mathtt{t}' \sqcap \mathtt{s}$.

In Section 4.4 we show how the rules presented here can be used in transformational reasoning in the style of window inference.

4.3 Using Lattice Properties in Various Domains

We now consider concrete examples of lattices encountered in various domains and show how abstract lattices (as they were formalised in the preceding section) can be instantiated in the HOL system.

4.3.1 Concrete Instances of Lattices

We can encounter concrete lattices in various contexts. We start with the most commonplace domain in classical logic - truth values.

The truth values with implication as the ordering form a complete, boolean and totally ordered lattice. Other logical connectives can be treated as lattice operations as well (conjunction is a binary meet, disjunction is a binary join, and negation is an inverse). The constants F and T are the bottom and top elements of the lattice. The *bounded* universal quantification ($\forall v \in I \bullet t$) stands for a general meet, and *bounded* existential quantification

$(\exists v \in I \bullet t)$ for a general join. Thus the truth values form a very specific and restricted lattice structure with many useful properties.

Pointwise extension is a general method by which operations on a type α can be lifted to operations on functions from some type β to α. If we introduce an ordering on this new type $\beta \to \alpha$ by pointwise extension, i.e.,

$$f \sqsubseteq_{\beta\to\alpha} g \;=\; (\forall x : \beta \bullet\; f.x \;\sqsubseteq_\alpha\; g.x)$$

then the new type $\beta \to \alpha$ inherits the property of being a lattice, as well as completeness, distributivity, and the property of being a boolean lattice. The lattice operations in $\beta \to \alpha$ are defined in terms of the corresponding operations on α by pointwise extension.

As we know, *bool* is a complete boolean lattice. Using pointwise extension, we derive that $\alpha \to$ *bool* (predicates or subsets of α) is a complete boolean lattice, for arbitrary type α. The ordering in this lattice is defined by pointwise extension:

$$p \;\subseteq\; q \;=\; (\forall x : \alpha \bullet\; p.x \;\Rightarrow\; q.x)$$

Other lattice operations are also lifted to the new lattice by pointwise extension. For example, binary meet in the predicate lattice is defined as

$$p \;\cap\; q \;=\; (\lambda x : \alpha \bullet\; p.x \;\wedge\; q.x)$$

Doing one more step, we derive that relations (as functions $\beta \to \alpha \to$ *bool*) form a complete boolean lattice as well.

As the last example of concrete lattices, we take a lattice that is not constructed by pointwise extension. Consider finite sequences of natural numbers (or finite sequences of some other ordered type). The ordering is defined as a lexicographic ordering:

$$\begin{aligned} s \sqsubseteq s' \;=\;& (s = <>) \\ & \vee\; ((s' \neq <>) \wedge (hd.s \;\leq\; hd.s')) \\ & \vee ((s' \neq <>) \wedge (hd.s \;=\; hd.s') \wedge (tail.s \;\sqsubseteq\; tail.s')) \end{aligned}$$

It is easy to show that this yields a lattice but not a complete lattice because it is unbounded from the top, and, therefore, general joins need not exist.

4.3.2 Refinement Calculus Theory

Predicates are only one of many interesting lattice structures, others include imperative programs ordered by program refinement. In particular, we consider lattice structures found in the refinement calculus theory.

Recall that the program state in the theory is modelled as a polymorphic type α. Above, we showed that state predicates (functions $\alpha \rightarrow bool$) and state relations (functions $\beta \rightarrow \alpha \rightarrow bool$) are complete boolean lattices. Using pointwise extension for state predicates once again results in the type $(\beta \rightarrow bool) \rightarrow (\alpha \rightarrow bool)$ (predicate transformers) which is also a complete boolean lattice. The ordering on this lattice is program refinement ordering defined in the usual way:

$$S \sqsubseteq T = (\forall q : \beta \rightarrow bool \bullet S.q \subseteq T.q)$$

Note that we have constructed a pointwise extension hierarchy of types starting from *bool*. At all levels of this hierarchy we have complete boolean lattices and we can use all properties of abstract lattices of this kind that were mentioned in Section 4.2.

4.3.3 Instantiation of the Abstract Lattice Theory

In Section 4.2 we showed how the HOL theory of abstract lattices can be created. Just above we presented several concrete domains which turned out to be lattices of some kind. How can the properties of abstract lattices be used in the concrete domains?

In order to instantiate the abstract lattice theory, one needs a partial order on a type satisfying the defining property of lattices (complete lattices etc.). As our example, we take predicates defined as functions of the type $\alpha \rightarrow bool$.

The partial order on predicates is the universal implication order (lifted from the booleans). Recall that the universal implication is defined as the infix constant `implies` in our HOL theory (see Section 3.3):

```
implies_DEF =
   ⊢def p implies q = (∀s. p s ⇒ q s)
```

The operations on predicates `and`, `or` and `not` are also defined by lifting the corresponding operations on truth values.

We can now use the HOL system to prove that the predicates of the given type $\alpha \rightarrow bool$ are in fact a lattice:

```
pred_lat =
   ⊢ LAT(U,implies)
```

According to the definition of `LAT`, one must prove that `implies` is a partial order, and both a meet and a join exist for every pair of lattice elements. Choosing `and` as the meet operator and `or` as the join operator, we can easily prove this theorem.

The predicates are, of course, also a boolean lattice. To prove this, we first prove that they form a complete lattice with with `glb` as the meet operator and `lub` as the join operator:

```
pred_clat =
   ⊢ CLAT(U,implies)
```

The greatest lower bound `glb` and the least upper bound `lub` operations on sets of predicates are defined as follows (see Section 3.3):

```
glb_DEF =
   ⊢def glb P = (λs. ∀p. P p ⇒ p s)

lub_DEF =
   ⊢def lub P = (λs. ∃p. P p ∧ p s)
```

The proof in HOL that the predicates satisfy the infinite distributivity property is tedious but not difficult. Finally, the predicates can be proved to be a boolean lattice with `not` as the inverse operator.

The next step would be instantiation and specialisation of inference rules and general theorems for lattices. But this is not necessary because these actions are done automatically in the inference rules and special rewrite rules for lattices, using a database of preproved lattices and lattice instantiations. The implementation details are described in Section 4.5.

4.4 Using Window Inference

In this section we show how window inference can be used when working with lattices. Window inference and its HOL implementation were described in Section 3.4.

4.4.1 Transformation and Window Rules for Lattices

The ordering relation on lattices is a partial order and therefore also a pre-order. Thus, the lattice ordering can be used as the relation to be preserved in the window inference system.

As explained in Section 3.4, in order to transform the focus in the window inference system, one must provide special SML functions called transformation rules. We have written a transformation rule for each introduction and elimination rule presented in Section 4.2.7. Recall that the conclusion parts of our inference rules are of the form $t \sqsubseteq t'$ where the relation is the lattice ordering. If the focus expression can be matched to the left hand side of the conclusion expression, then the conclusion of an inference rule is the required theorem for transforming the focus.

If the conclusion of an inference rule has hypotheses, then these hypotheses become proof obligations. If some hypothesis can be matched to one of the contextual assumptions, then it is discharged automatically. All necessary instantiations of the inference rules of an abstract lattice are done automatically as well.

Window rules of the window inference system allow us to open a window on some subterm of the current focus and transform it, using the fact that the focus is monotonic in the position where the subterm occurs. Because the lattice constructs are monotonic (or antimonotonic) in their arguments (as explained in Section 4.2.7), we have written one window rule for each argument position of each lattice operator.

For example, the window rule for opening a window on the left argument of a binary meet uses the fact that a binary meet is monotonic in its left argument (see the monotonicity rule for a binary meet in Section 4.2.7). Similar rules are implemented for the right argument of a binary meet as well as for both arguments of a binary join.

Similarly, we have implemented two window rules for a general meet $(\sqcap v \in I \bullet s)$ (one for the index set I and one for the body expression s) and also two rules for a general join.

4.4.2 Basic Rewrites

Because equality is the smallest preorder relation, it is always possible to transform the current focus using equational theorems (provided the left hand side of the equation matches the current focus). For this purpose the command `REWRITE_WIN` is used in the window inference system. It takes a window stack and a list of equational theorems and produces the new window stack where the focus expression is transformed (rewritten) according to the equational theorems provided.

It is convenient to have a similar rewrite rule for working with lattices as well. We have implemented such a rule (called `LAT_REWRITE_WIN`) which does all necessary instantiations in order to make an equational theorem ex-

pressing some basic property of the abstract lattice applicable to the current focus. `LAT_REWRITE_WIN` with the empty list as an argument transforms the focus using a list of trivial properties of abstract lattices (such as $s \sqcup s = s$ and $s \sqcap \top = s$).

4.4.3 Example of a Derivation

Here we show a simple example of a derivation in the window inference system using general lattice properties. As concrete domain, we take the predicate lattice instantiated in the way described in Section 4.3.3 (we assume that the relation `implies` has been added as a relation that is supported in the window inference system). We set up the window stack with `p or` (`q and r`) as the focus, `r implies p` as the assumption and `implies` as the relation to be preserved. The HOL window Library prints this stack as follows:

```
         ! r implies p
implies * p or (q and r)
```

Here the first line is the contextual assumption (marked with the exclamation mark) and the second line contains the relation to be preserved and the current focus (separated by the asterisk).

Let us now focus on the right subterm of the focus. The window inference system allows this because `or` (the binary join operator) is monotonic in its right argument and we have added the corresponding window rule for the abstract case to the system. All necessary instantiations to the concrete level of predicates are done automatically.

To open a window on a subterm in the window inference system, one has to use the command `OPEN_WIN` which takes as an argument the path describing the position of the desired subterm within the focus. A path is a list made up of the HOL constructors `RATOR`, `RAND` and `BODY` which indicate how one should navigate in the HOL abstract syntax tree to reach the subterm. Intuitively `RATOR` means "take the operator ($\mathtt{t}_1$) of a function application $\mathtt{t}_1\,\mathtt{t}_2$", `RAND` means "take the operand ($\mathtt{t}_2$) of a function application $\mathtt{t}_1\,\mathtt{t}_2$", and `BODY` means "take the body ($\mathtt{t}$) of an abstraction ($\lambda \mathtt{v}.\,\mathtt{t}$)".

The right subterm of the focus can be reached as the operand of the function application (`or p`) (`q and r`) in the internal HOL representation:

```
- DO(OPEN_WIN [RAND]);
         ! r implies p
implies * q and r
```

In the same way we can open a window on `r` using the fact that a binary meet is monotonic in its right argument as well. Then the contextual

assumption can be used to transform the focus. This is done with the command `TRANSFORM_WIN`. This command takes a theorem of the form $s\ R\ t$, where R is the transformation relation, and, if the left hand side expression in the theorem matches the current focus, transforms the focus accordingly:

```
- DO(OPEN_WIN [RAND]);
- DO(TRANSFORM_WIN (ASSUME (--‘r implies p‘--)));
- DO(CLOSE_WIN);
         ! r implies p
 implies * q and p
```

Next, we simplify the focus using the transformation rule for binary meet elimination ($a \sqcap b \ \sqsubseteq b$):

```
- DO(MEET2_ELIM_WIN2);
         ! r implies p
 implies * p
```

Closing the window with the command `CLOSE_WIN` gives us the transformed version of our initial expression:

```
- DO(CLOSE_WIN);
         ! r implies p
 implies * p or p
```

In the final step, the default rewrite applies the idempotence property of binary join ($a \sqcup a \ = \ a$):

```
- DO(LAT_REWRITE_WIN[]);
         ! r implies p
 implies * p
```

With the command `WIN_THM`, we can now retrieve the theorem that has been proved in the derivation:

```
- WIN_THM();
val it =
  r implies p |- p or (q and r) implies p  : thm
```

Note that our steps in this example are independent of what concrete lattice we are working with. If we had some other lattice instead of the predicate one, our actions would be absolutely the same, and the end result would have been another instance of the theorem $r \sqsubseteq p \ \vdash p \sqcup (q \sqcap r) \sqsubseteq p$.

4.5 Implementation Issues

In the previous section we showed how to use lattice properties in derivations working with window Library in HOL. However, working with subexpressions (selecting subexpression by path, regrouping expressions by explicit application of commutativity and associativity, etc.) is very tedious with the standard command line interface to HOL.

4.5.1 The Refinement Calculator

We have added general lattice transformations to the system and made them available in a separate menu. When a user chooses a menu alternative, the appropriate transformation rule is applied to the current focus. If the rule requires arguments, then a dialog window pops up and the user can type in them. It is also possible to indicate (by pointing and clicking) that a transformation should be applied to a specific subterm of the current focus, rather than to the whole focus.

It should be pointed out that the lattice transformations can also be used in ordinary proofs, since the booleans with forward or backward implication as the ordering are a complete boolean lattices. Thus transforming a boolean expression to truth while preserving backward implication is a special case of preserving a lattice ordering.

4.5.2 A Database of Concrete Lattices

The window inference system stores information about supported relations and window rules in a database. Similarly, our lattice tool has a small database with information about preproved lattices and lattice instantiations. By default, there are theorems that show (among others) that *bool*, predicates and predicate transformers are complete boolean lattices. In all situations, when properties of such concrete lattices are used, corresponding assumptions about being (complete, boolean) lattice are automatically discharged. The database also contains information about lattice instantiations, i.e., theorems of the form $< operator > = < lattice\ construct >$. For example,

$$\vdash \wedge \ = \ \texttt{meet2}(\texttt{U}, \Rightarrow)$$

is the binary meet instantiation for the lattice *bool* with the forward implication ordering.

Before a lattice transformation is applied to the focus, all concrete lattice operators (i.e., instantiations of a meet, a join etc.) are rewritten into ab-

stract form. After the transformation is done, the converse rewriting takes place. Thus one is not forced always to use the abstract form of lattice constructs in order to transform expressions according to some lattice property.

The user can add further lattice structures to the database, by supplying the appropriate theorems (i.e., theorems stating that the structure is a lattice, that a specific operator is a meet etc).

4.5.3 Pretty Printer

We have extended the parser and the pretty-printer to allow traditional mathematical syntax for binary and general meets and joins: $a \sqcap b$, $a \sqcup b$, $(\sqcap v \in I \bullet s)$ and $(\sqcup v \in I \bullet s)$ rather than the the current HOL syntax which is hard to read. For example, the HOL syntax for general meet is

$$\texttt{meet}\ (\texttt{U}, \texttt{po})\ (\lambda \texttt{x}.\ \exists \texttt{v}.\ \texttt{I v} \ \wedge\ (\texttt{x} = \texttt{s}))$$

where `po` is the ordering.

With pretty-printing we can still open subwindows on subexpressions of the focus and transform them using applicable window rules. The pretty-printer automatically translates the path to the subexpression on the screen to the actual path in abstract HOL syntax.

4.5.4 Example of a Refinement

Let us try a simple refinement example using the Refinement Calculator. Suppose we are refining the program containing as a subcomponent the nondeterministic assignment statement "$\texttt{x} := \texttt{x}'.\ \texttt{x}' = \texttt{0} \vee \texttt{x}' = \texttt{1}$" ($x$ is assigned either 0 or 1).

Recall that in the refinement calculus theory a nondeterministic assignment is defined as a predicate transformer in the form

$$(\lambda q : \beta \to \mathit{bool} \bullet\ \lambda \sigma : \alpha \bullet\ \forall \gamma : \beta \bullet\ R.\sigma.\gamma\ \Rightarrow\ q.\gamma)$$

where q is a postcondition, σ an initial state, γ a final state and R is a relation on initial and final states.

We have proved that a nondeterministic assignment in general is equivalent to a general meet on predicate transformers of the form

$$(\sqcap (\sigma, \gamma) \in R \bullet\ (\lambda q \bullet\ \lambda \sigma' \bullet\ (\sigma' = \sigma)\ \Rightarrow\ q.\gamma)\ ).$$

Intuitively, this means a general meet on all such statements (predicate transformers) which started in any state $\sigma \in \mathit{Dom}\, R$ reach a state γ such that $R.\sigma.\gamma$ holds.

Therefore, focusing on the body of our nondeterministic assignment

$$\mathtt{x} := \mathtt{x}'.\ \mathtt{x}' = 0 \ \vee \ \mathtt{x}' = 1$$

we in fact focus on the index set of a general meet (of a special form). Because a general meet is antimonotonic in its index set (see Section 4.2.7), we arrive at the following starting expression for the subderivation:

$$\lambda(\mathtt{x}, \mathtt{x}').\ \mathtt{x}' = 0 \ \vee \ \mathtt{x}' = 1$$

The relation to be preserved is `implied_by` - the ordering reverse to the `implies` ordering on predicates. The easiest way to refine such an expression is to focus on the body of the lambda expression, thus moving from the predicate to the boolean level. In this case the following focus is constructed:

$$\mathtt{x}' = 0 \ \vee \ \mathtt{x}' = 1$$

The relation to be preserved now is $\Leftarrow$ (backward implication) [1]. $(bool, \Leftarrow)$ is a complete boolean lattice dual to the $(bool, \Rightarrow)$ lattice. Because of duality $\vee$(disjunction) is binary meet in the lattice $(bool, \Leftarrow)$. Therefore, we can apply binary meet elimination rule to simplify the current expression to the left disjunct. This results in the following focus:

$$\mathtt{x}' = 0$$

Closing windows yields the following refinement of the initial subcomponent:

$$\mathtt{x} := \mathtt{x}'.\ \mathtt{x}' = 0.$$

It is easy to show that such a nondeterministic assignment is equivalent to the ordinary assignment `x:=0`. Applying the appropriate transformation rule we finish our derivation. The final result is shown as the theorem (in the pretty-printed form):

$$\vdash \ (\mathtt{x} := \mathtt{x}'.\, \mathtt{x}' = 0 \ \vee \ \mathtt{x}' = 1) \ \mathtt{ref} \ (\mathtt{x} := 0).$$

The following structured derivation[9] describes the structure of the proof.

[1] We could open a window on the boolean expression in one step. The window inference system would then chain the necessary window rules on our behalf.

$$
\begin{array}{ll}
\vdash & x := x'.\ x' = 0 \ \vee \ x' = 1 \\
\sqsubseteq & \{\text{focusing on the body expression}\} \\
& \quad \bullet \ (\lambda(x, x') \bullet \ x' = 0 \ \vee \ x' = 1) \\
& \quad \supseteq \ \{\text{focusing inside of the abstraction}\} \\
& \quad\quad \bullet \ x' = 0 \ \vee \ x' = 1 \\
& \quad\quad \Leftarrow \ \{\text{applying meet elimination}\} \\
& \quad\quad\quad x' = 0 \\
& \quad\quad (\lambda(x, x') \bullet \ x' = 0) \\
& x := x'.\ x' = 0 \\
= & \ \{\text{transforming the nondeterministic assignment}\} \\
& x := 0
\end{array}
$$

Indentations (marked by •) indicate subderivations needed to justify certain steps in the derivation.

This example shows how the nondeterministic assignment statement can be refined using lattice properties. The body of the nondeterministic assignment is very simple in the example but the same principles could be applied for more complex cases as well.

4.6 Conclusions

The work presented in this chapter can be seen as an example of implementation of abstract theories in the HOL system. Rather than changing the logic or the system or adding extra layers of syntax, we have created a general theory of lattices and provided ways of instantiating this theory to concrete lattices. This means that the same underlying theory can be used for transformational reasoning in situations that on the surface seem to have very little in common, such as backward proof (transforming boolean terms under backward implication) and program development (transforming programs under a correctness preserving refinement relation).

Similar formalisations of abstract theories were implemented in HOL by Gunter[51] and Windley[105]. Regensburger[92] has formalised the theory of complete partial orders (cpo's) in Isabelle using type classes. His work is a genuine technical advance over prior work by Agerholm[1] on cpo's in HOL. However, Isabelle's type classes allow a type to be a lattice only in

one way (with one ordering relation). Our approach is more flexible in this sense.

Recently Kammüller[60, 61] has presented a method of implementing abstract algebraic structures in Isabelle using so called *dependent types*[73, 72]. He constructs dependent types as Isabelle/HOL sets and uses them to represent modular structures (such as abstract theories) by semantical embedding. Furthermore, patterns of algebraic structures are represented using the very recent concept of record types in Isabelle[88]. The main advantage of Kammüller's approach comparing to ours is that the algebras themselves remain first class citizens of the logic. Therefore, it is possible to define operations on abstract theories as functions of higher-order logic.

The system described in this chapter depends heavily on the Window Inference library of the HOL system. The Refinement Calculator tool provides a good interface and numerous instances of lattices, but it is also possible to use our system with only the standard HOL system. A moderate amount of pretty-printing and parsing is still needed to make the system easy to use for someone who is not familiar with the HOL system.

Throughout the rest of the Thesis we focus on programs formalised within the refinement calculus framework. We use the properties of abstract lattices to reason about programs in this formalisation.

Chapter 5

The Context Extension

5.1 Introduction

In this chapter we describe two approaches for context handling in the refinement calculus framework. They show how information relevant for total correctness can be transported from one place of a program to another and then used for refinement of program components. Both approaches have been formalised in the HOL theorem proving system and integrated as a separate extension of the Refinement Calculator.

Programs can be very large and complex. Therefore, it is usually very difficult to prove refinement of the whole program directly. Instead, one can refine a program by focusing on some small subcomponent and replacing it with another which is a refinement of the first one. Such transformations are possible whenever the context of the selected component is monotonic with respect to the refinement relation.

In practice, we usually do not need refinements that are correct in any context; it is sufficient if the refinement of a subcomponent is correct in the specific context that it occurs in. Such refinements can be handled by introducing *context information* into the program. There are two dual approaches for handling context information in the refinement calculus - by *propagating context assertions* and by *propagating context assumptions*.

By propagating context assertions in a program we collect information about the program text surrounding a certain part of a program. Context information is accumulated in the assertion predicate and can be used for refinement of the following program subcomponents. The dual approach with context assumptions allows us to assume facts that are expected to be true in a certain place of our program. Using this information we can make

program refinements that are correct provided our introduced assumption holds. This proviso can then be discharged by propagating the assumption backwards taking the preceding context into account.

In this chapter we describe how both approaches for handling context information have been implemented in the Refinement Calculator. We have extended the Refinement Calculator with special transformation and window rules for working with program context. The methods of context handling and their formalisation in the HOL theorem prover are described in the first half of the chapter, while the second half explains how they can be used for program derivation within the Refinement Calculator tool.

5.2 Handling Context Information

The general problem of using context information can be described as follows. Assume that we are working with a program statement C containing a specific substatement S. We write this as $C[S]$ and refer to C as the *context* of S. Since statement constructors are monotonic, we can always replace S by another statement S' that refines S, and then we have that $C[S] \sqsubseteq C[S']$ holds.

However, it may be possible to replace S with a statement S' that does not refine S in isolation, but where $C[S] \sqsubseteq C[S']$ does hold. We can do that in the following way. We can find a (reasonably strong) predicate p such that $C[S] = C[\{p\}; S]$. Intuitively this means that we are collecting information about the context of statement S in the assert statement $\{p\}$, so that we may replace S by any command that is a refinement of $\{p\}; S$. A derivation then has the following structure[15]:

$$
\begin{array}{ll}
 & C[S] \\
= & \{\text{introduce context assertion}\} \\
 & C[\{p\}; S] \\
\sqsubseteq & \{\text{prove } \{p\};\ S \sqsubseteq S', \text{ monotonicity}\} \\
 & C[S']
\end{array}
$$

A refinement of the form $\{p\}; S \sqsubseteq S'$ is called a *refinement in context* p. It means that we only have to prove refinement between S and S' for those states that satisfy the condition p. This is weaker than $S \sqsubseteq S'$, which requires refinement between S and S' for every initial state.

Assertions as the statements carrying context information and a number of rules for handling assertions were introduced by Back in his original

formulation of the refinement calculus[4].

5.2.1 Context Assertions

For refinement in context to work in practice, we need to find a way of introducing assertions at different points of a program. We can always introduce an assertion $\{true\}$ anywhere in a program because the predicate $true$ holds independently of a context (formally, $\{true\} = \mathsf{skip}$). We then push (propagate) the introduced assertion step by step towards the point where we want to collect context information. Each propagation step adds new context information about the statement propagated through into the assertion predicate. This accumulated information can then be used for refinement of the selected subcomponent as described above. After the refinement is done the assertion can be discharged (because $\{p\} \sqsubseteq \mathsf{skip}$ always holds).

The following simple derivation illustrates the idea.

$$
\begin{array}{ll}
 & x := 0;\ \mathsf{if}\ z = 1\ \mathsf{then}\ y := x + z\ \mathsf{else\ skip\ fi} \\
\sqsubseteq & \{\text{introduction of context assertion}\} \\
 & \{true\};\ x := 0;\ \mathsf{if}\ z = 1\ \mathsf{then}\ y := x + z\ \mathsf{else\ skip\ fi} \\
\sqsubseteq & \{\text{assertion propagation through assignment statement}\} \\
 & x := 0;\ \{x = 0\};\ \mathsf{if}\ z = 1\ \mathsf{then}\ y := x + z\ \mathsf{else\ skip\ fi} \\
\sqsubseteq & \{\text{assertion propagation into conditional statement}\} \\
 & x := 0;\ \mathsf{if}\ z = 1\ \mathsf{then}\ \{x = 0 \wedge z = 1\};\ y := x + z\ \mathsf{else\ skip\ fi} \\
\sqsubseteq & \{\text{refinement in context}\} \\
 & x := 0;\ \mathsf{if}\ z = 1\ \mathsf{then}\ \{x = 0 \wedge z = 1\};\ y := 1\ \mathsf{else\ skip\ fi} \\
\sqsubseteq & \{\text{elimination of assertion}\} \\
 & x := 0;\ \mathsf{if}\ z = 1\ \mathsf{then}\ y := 1\ \mathsf{else\ skip\ fi}
\end{array}
$$

Note that some of the steps here really preserve equality. However, since refinement is the relation we are interested, we write $\sqsubseteq$ even where we could write $=$.

The approach of propagating context assertions relies on special rules for propagating assertions through various statements of the language. The rules have the form $\{p\}; S \sqsubseteq S; \{q\}$ where p and q are predicates and $\sqsubseteq$ is refinement relation.

Lemma 5.1. *Context assertions can be propagated forward according to the following rules*:

$$
\begin{array}{rcl}
\{p\};\ \mathsf{skip} & \sqsubseteq & \mathsf{skip}\ ;\ \{p\} \\
\{p\}\ ;\{q\} & \sqsubseteq & \{q\};\ \{p \wedge q\} \\
\{p\};\ [q] & \sqsubseteq & [q];\ \{p \wedge q\} \\
\{p\};\ x := e & \sqsubseteq & x := e;\ \{\exists x_0 \bullet\ p[x_0/x]\ \wedge\ (x = e[x_0/x])\} \\
\{p\};\ x := x'.Q & \sqsubseteq & x := x'.Q;\ \{\exists x_0 \bullet\ p[x_0/x]\ \wedge\ Q[x_0, x/x, x']\} \\
\{p\};\ \mathsf{if}\ g\ \mathsf{then}\ S\ \mathsf{else}\ S'\ \mathsf{fi} & \sqsubseteq & \mathsf{if}\ g\ \mathsf{then}\ \{p \wedge g\}; S\ \mathsf{else}\ \{p \wedge \neg g\}; S'\ \mathsf{fi} \\
\mathsf{if}\ g\ \mathsf{then}\ S; \{q\}\ \mathsf{else}\ S'; \{q'\}\ \mathsf{fi} & \sqsubseteq & \mathsf{if}\ g\ \mathsf{then}\ S\ \mathsf{else}\ S'\ \mathsf{fi}\ ;\ \{q \vee q'\} \\
\{p\};\ |[\ \mathsf{var}\ x | q \bullet\ S\ ]| & \sqsubseteq & |[\ \mathsf{var}\ x | q \bullet\ \{p\ \wedge\ q\}; S\ ]| \\
|[\ \mathsf{var}\ x | q \bullet\ S; \{r\}\ ]| & \sqsubseteq & |[\ \mathsf{var}\ x | q \bullet\ S\ ]|;\ \{\exists x \bullet\ r\}
\end{array}
$$

Proof The detailed derivation of these rules is presented by Back and von Wright in [15]. □

Note that each rule deals with assertion propagation through some particular statement of the refinement calculus theory. Conditional (if-then-else) and block statements have two separate rules - for propagation from the outside (in front) into the statement and from inside out of (past) the statement.

All rules in Lemma 5.1 have been proved as theorems of the HOL theory of the refinement calculus. Also, we have proved that these rules are "sharp", i.e., the assertions on the right hand side of the rules are the strongest possible. Intuitively this means that the assertion predicates accumulate the maximum possible amount of context information.

This maximal amount of context information is closely related to the concept of a *strongest postcondition*. In fact, for terminating statements both notions amounts to the same thing. For nonterminating (aborting) statements the maximal amount of propagated context information is always the predicate *false*, while the strongest postcondition is undefined.[1]

5.2.2 Loops

For loops, a loop invariant can be added as a context assertion. In addition to the invariant, the guard of the loop gives us information; it always holds

[1]This is explained by the fact that a strongest postcondition was originally defined in the partial correctness framework[35].

when the body is executed and it cannot hold after the loop has terminated. This is formulated in the following lemma.

Lemma 5.2. *Assume that statement S is correct with respect to pre-postcondition pair $(g \wedge p,\ p)$. Then context assertion can be added to while-loop as follows*:

$$\{p\};\ \mathsf{do}\ g \rightarrow S\ \mathsf{od} \quad \sqsubseteq \quad \mathsf{do}\ g \rightarrow \{p \wedge g\};\ S\ \mathsf{od};\ \{\neg g \wedge p\}$$

This rule works only for a loop invariant, so it cannot be used to propagate an arbitrary context assertion through a loop. The assertion added into the body of the loop can provide useful context information when the loop body is refined.

Unfortunately, the context propagation rule presented in Lemma 5.2 is not sharp, i.e., the predicate of the context assertion after a loop is not necessarily the strongest possible. The problem of calculating of maximum possible context information after a loop was investigated earlier by de Bakker[18] and Back[6]. However, their proposed solution of calculating it as the disjunction of strongest postconditions over all loop approximations is impossible to use in practice.

Working with the Refinement Calculator, do-loops are usually introduced using the `Loop Introduction` transformation rule. At that point the user is asked to supply (besides other things) the loop guard g and the loop invariant Inv. Both the loop guard and the invariant are then used to provide context information (in the form of context assertions) and to formulate proof obligations necessary to prove correctness of the loop introduction.

Since it is desirable to collect as much context information as possible via propagation, let us to consider how our results can be improved using the given loop invariant Inv and loop guard g. First of all, Lemma 5.2 can be rewritten as an inference rule of the following form:

$$\frac{\{g \ \wedge\ Inv\}\ c\ \{Inv\}}{\{Inv\};\ \mathsf{do}\ g\ \rightarrow\ c\ \mathsf{od}\ \ \sqsubseteq\ \ \mathsf{do}\ g\ \rightarrow\ c\ \mathsf{od};\ \{\neg g\ \wedge\ Inv\}} \tag{5.1}$$

The hypothesis (correctness triple) of this inference rule expresses the fact that Inv is so called a *(strong) invariant* of a loop.

Stronger results can be obtained if we ignore initial states from which the loop body does not terminate (since they lead to the trivial refinement $abort \sqsubseteq abort$). This is expressed in the following inference rule:

$$\frac{\{(c\ true)\ \wedge\ g\ \wedge\ Inv\}\ c\ \{Inv\}}{\{Inv\};\ \mathsf{do}\ g\ \rightarrow\ c\ \mathsf{od}\ \ \sqsubseteq\ \ \mathsf{do}\ g\ \rightarrow\ c\ \mathsf{od};\ \{\neg g\ \wedge\ Inv\}}$$

Here, we only require that Inv is a *weak invariant* of a loop. However, it can be difficult to calculate the predicate $c\,true$ so it is often omitted.

In practice, the most typical case of losing context information using the context propagation rules above is when the given loop invariant is not strong enough. Let us illustrate this by a simple example. Suppose we are working with the following program fragment:

$$
\begin{array}{l}
\{(x = 10) \ \wedge \ (y = 0) \ \wedge \ (i = 0)\}; \\
\mathsf{do}\ i < 20 \ \rightarrow \\
\quad y \ := \ y \ + \ a[i]; \\
\quad i \ := \ i \ + \ 1 \\
\mathsf{od}
\end{array}
$$

The purpose of the loop in the example is to calculate the sum of all elements of the subarray $a[0..19]$ and store the value in the variable y. The invariant sufficient to prove correctness of this loop is

$$(y \ = \ (\sum j |\, j < i \bullet \ a[j])) \ \wedge \ (i \leq 20)$$

The inference rule (5.1) yields the context assertion

$$(y \ = \ (\sum j |\, j < i \bullet \ a[j])) \ \wedge \ (i = 20)$$

propagated right after the loop. However, this context assertion does not tell us anything about the possible value of the variable x. Since this variable is not used in the loop body, it is obvious that x should retain the value it had before execution of the loop. Therefore, the context assertion after the loop should include the conjunct $x = 10$.

This means that the predicate $x = 10$ itself is an invariant of the loop. It is known that if $Inv1$ and $Inv2$ are invariants of the loop, so is $Inv1 \ \wedge \ Inv2$. The conjunction of all loop invariants would give us the strongest loop invariant. Therefore, the predicate $x = 10$ would be part of the strongest invariant of the loop. However, calculation of the strongest possible invariant of a loop can be a very difficult task in general.

As an alternative, we consider the problem of propagating additional context information through a do-loop in cases when the current loop invariant is not sufficiently strong. In other words, under what conditions is the refinement

$$\{p\};\ \mathsf{do}\ g \ \rightarrow \ c\ \mathsf{od} \quad \sqsubseteq \quad \mathsf{do}\ g \ \rightarrow \ c\ \mathsf{od};\ \{p\}$$

true.

Let us consider two different cases. In the first case, the predicate can be propagated through a do-loop, if it is implied by the current loop invariant:

$$\frac{(\{g \ \wedge \ Inv\}\ c\ \{Inv\}) \ \wedge \ (Inv \ \Rightarrow \ p)}{\{p\};\ \mathsf{do}\ g \ \rightarrow \ c\ \mathsf{od} \ \sqsubseteq \ \ \mathsf{do}\ g \ \rightarrow \ c\ \mathsf{od};\ \{p\}} \tag{5.2}$$

This rule is the direct consequence of (5.1), using the fact that an assertion statement is monotonic in its predicate parameter.

In the second case, the predicate is either a loop invariant itself or the part of a stronger (weak) invariant of a do-loop. This can be formulated as the following inference rule:

$$\frac{\{(c\ true) \ \wedge \ g \ \wedge \ Inv \ \wedge \ p)\}\ c\ \{p\}}{\{p\};\ \mathsf{do}\ g \ \rightarrow \ c\ \mathsf{od} \ \ \sqsubseteq \ \ \mathsf{do}\ g \ \rightarrow \ c\ \mathsf{od};\ \{p\}} \tag{5.3}$$

Once again, $c\ true$ can be omitted.

This rule actually means that the predicate $p \wedge Inv$ is a (weak) invariant of a loop (though p itself might not be a (weak) invariant), and, therefore, the predicate p can be propagated through a do-loop according to the rule (5.2).

The rules (5.2) and (5.3) present two ways to obtain sharper results by propagating context information through a do-loop. The rule (5.3) is especially useful since it allows us to propagate any predicate that is part of the strongest loop invariant. For example, any predicate on program variables that are unaffected by loop execution can be propagated, according to this rule.

5.2.3 Refinement in Context

Once context information has been accumulated in the assertion predicate using the context propagation rules presented above, it can be used for program refinement. The general form of refinement in context is $C[\{p\}; S] \sqsubseteq C[S']$. We now present rules for the situations where S has a certain syntactic form.

The rules are given as inference rules. The hypothesis part of each inference rule is of the form $p \vdash e\ R\ e'$ and the conclusion part is of the form $\vdash \{p\};\ S[e] \ \sqsubseteq \ \{p\};\ S[e']$. Intuitively, we can understand these rules in the following way. Suppose we have a statement S containing an expression e as a subcomponent. In addition, we know that the statement S occurs in the context p (from the fact that the assertion $\{p\}$ appears just before S). If in a separate subderivation we can prove, using the assumption p, that is e is related to e' by some (reflexive and transitive) relation R, then we can refine

our statement $S[e]$ to the statement $S[e']$. All the inference rules presented below are based on the corresponding HOL theorems proved beforehand.

For the assignment statement, we have the following rule:

$$\frac{p \vdash e = e'}{\vdash \{p\};\ x := e \ \sqsubseteq\ \{p\};\ x := e'}$$

Thus, we can rewrite the right hand side of an assignment statement using the available context information. The corresponding HOL theorem is as follows:

```
⊢ (∀s. p s ⇒ (e s = e' s)) ⇒ ({p};(assign e) ref {p};(assign e'))
```

The rule for the nondeterministic assignment statement is

$$\frac{p \vdash Q \supseteq R}{\vdash \{p\};\ x := x'.Q \ \sqsubseteq\ \{p\};\ x := x'.R}$$

Thus, we can change the relation of a nondeterministic assignment if we can prove that the new relation is stronger than the old one, assuming the accumulated context information.

The rule for the conditional statement is

$$\frac{p \vdash g = g'}{\vdash \{p\};\ \mathsf{if}\ g\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \ \sqsubseteq\ \{p\};\ \mathsf{if}\ g'\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}}$$

Using this rule, we can change the guard of a conditional statement. We do not need additional rules for refining the bodies S_1 and S_2 of a conditional statement, since the same results can be achieved by propagating context information inside the statement using the rules given in the Section 5.2.1.

The rule for the block statement is

$$\frac{p \vdash q \supseteq q'}{\vdash \{p\};\ |[\,\mathsf{var}\ x \mid q.\ S\,]| \ \sqsubseteq\ \{p\};\ |[\,\mathsf{var}\ x \mid q'.\ S\,]|}$$

Using this rule we can change the initialisation predicate of the block statement, taking context information into account.

Finally, the rule for the while-loop is

$$\frac{p \vdash g = g'}{\vdash \{p\};\ \mathsf{do}\ g \rightarrow S;\ \{p\}\ \mathsf{od} \ \sqsubseteq\ \{p\};\ \mathsf{do}\ g' \rightarrow S;\ \{p\}\ \mathsf{od}}$$

This rule states that we can rewrite the loop guard using context information p only if this predicate is an invariant of the loop.

5.3 Context Assumptions

The other (dual) approach for handling context information is based on using *assumption* statements (this approach was introduced by Morgan [83]). An assumption statement $[p]$ indicates that predicate p is assumed to be true at a certain point of the program.

We can always add context assumptions about facts we expect (hope, guess) to be true in any place of our program (formally, $\mathsf{skip} \sqsubseteq [p]$ always holds). This context information can then be used for refinement of the following program components. However, these refinements will be correct under the condition that our introduced assumption holds. We usually do not want to have such assumptions in the final version of our program so the context assumptions must be discharged. The only way to do this is to prove that they are in fact true, in the sense that the preceding context justifies them.

The following method for working with context assumptions is used. Suppose we want to refine some subcomponent S using the assumption that predicate p holds. We can then introduce an assume-assert pair using the following general rule:

$$S \sqsubseteq [p]; \{p\}; S$$

(here we really use the rule $\mathsf{skip} \sqsubseteq [p]; \{p\}$). Then we can refine the statement S using the context assertion $\{p\}$ as described above. Furthermore, we can propagate the assumption $[p]$ backwards using the rules described below. Each propagation step tends to weaken the assumption predicate because the context information about the statement propagated through is taken into account. If our introduced assumption is in fact legal, then after a number of propagation steps it will be transformed to $[true]$. The statement $[true]$ is equivalent to skip and can therefore be discharged.

Let us try the example from Section 3.1 using the context assumption approach.

$$
\begin{array}{ll}
 & x := 0;\ \mathsf{if}\ z = 1\ \mathsf{then}\ y := x + z\ \mathsf{else\ skip\ fi} \\
\sqsubseteq & \{\text{introduction of context assumption}\} \\
 & x := 0;\ \mathsf{if}\ z = 1\ \mathsf{then}\ [x = 0];\ \{x = 0\};\ y := x + z\ \mathsf{else\ skip\ fi} \\
\sqsubseteq & \{\text{refinement in context}\} \\
 & x := 0;\ \mathsf{if}\ z = 1\ \mathsf{then}\ [x = 0];\ y := z\ \mathsf{else\ skip\ fi} \\
\sqsubseteq & \{\text{assumption propagation out of the conditional statement}\}
\end{array}
$$

$$
\begin{array}{ll}
& x := 0;\ [x = 0\ \vee\ z \neq 1];\ \mathsf{if}\ z = 1\ \mathsf{then}\ y := z\ \mathsf{else\ skip\ fi} \\
\sqsubseteq & \{\text{assumption propagation through assignment statement}\} \\
& [true];\ x := 0;\ \mathsf{if}\ z = 1\ \mathsf{then}\ y := z\ \mathsf{else\ skip\ fi} \\
\sqsubseteq & \{\text{elimination of the assumption}\} \\
& x := 0;\ \mathsf{if}\ z = 1\ \mathsf{then}\ y := z\ \mathsf{else\ skip\ fi}
\end{array}
$$

Similarly, we could assume $z = 1$ at the same place of the program, use this assumption for refinement and then get the assumption discharged by propagating it backwards.

The approach of propagating context assumptions relies on special rules for propagating assumptions backwards through the various statements of the language. The rules have the form $S; [p] \sqsubseteq [p]; S$.

Lemma 5.3. *Context assumptions can be propagated backwards according to the following rules*:

$$
\begin{array}{rcl}
\mathsf{skip}; [q] & \sqsubseteq & [q]; \mathsf{skip} \\
\{p\}; [q] & \sqsubseteq & [\neg p \vee q]; \{p\} \\
[p]; [q] & \sqsubseteq & [p \vee q] \\
x := e;\ [q] & \sqsubseteq & [\ q[e/x]\ ];\ x := e \\
x := x'.Q;\ [q] & \sqsubseteq & [\forall x' \bullet\ Q \Rightarrow q[x'/x]\ ];\ x := x'.Q \\
\mathsf{if}\ g\ \mathsf{then}\ S\ \mathsf{else}\ S'\ \mathsf{fi}\ ;\ [q] & \sqsubseteq & \mathsf{if}\ g\ \mathsf{then}\ S; [q]\ \mathsf{else}\ S'; [q]\ \mathsf{fi} \\
\mathsf{if}\ g\ \mathsf{then}\ [q];\ S\ \mathsf{else}\ [q'];\ S'\ \mathsf{fi} & \sqsubseteq & [(g \wedge q) \vee (\neg g \wedge q')];\ \mathsf{if}\ g\ \mathsf{then}\ S\ \mathsf{else}\ S'\ \mathsf{fi} \\
|[\ \mathsf{var}\ x \mid p \bullet\ S\ ]|;\ [q] & \sqsubseteq & |[\ \mathsf{var}\ x \mid p \bullet\ S; [q]\ ]| \\
|[\ \mathsf{var}\ x | p \bullet\ [q]; S\ ]| & \sqsubseteq & [\forall x \bullet\ p \Rightarrow q];\ |[\ \mathsf{var}\ x | p \bullet\ S\ ]|
\end{array}
$$

Proof The detailed derivation of these rules is presented by Back and von Wright in [15]. □

Most of these rules are obtained immediately from Lemma 5.1 using the following general law for conjunctive predicate transformers:

$$\{p\};\ S\ \sqsubseteq\ S;\ \{q\} \quad \equiv \quad S;\ [q]\ \sqsubseteq\ [p];\ S$$

This law shows the duality between context assertions and context assumptions.

Exactly as the rules for assertion propagation, the rules of Lemma 5.3 have been proved as theorems in the HOL system and we have shown that they are sharp (i.e., the predicates in the propagated context assumptions are the weakest possible).

For loops, we can propagate a loop invariant backward. Thus, we have

$$\mathsf{do}\ g\ \rightarrow\ S\ \mathsf{od};\ [p] \quad \sqsubseteq \quad [p];\ \mathsf{do}\ g\ \rightarrow\ S\ \mathsf{od}$$

provided S is totally correct with respect to precondition $g \wedge p$ and postcondition p. Unfortunately, this rule does not seem very useful in practice since it does not give us anything new (by the definition, a loop invariant should be true before a do-loop anyway).

Propagation of assertions and assumptions easily leads to an explosion in size of the predicates involved. To achieve more efficiency and flexibility, we can combine both approaches and make assertions and assumptions meet "halfway", cancelling each other. Another way is to modify the predicates inside assertions and assumptions using monotonicity properties of these statements:

$$\begin{array}{l}(p \subseteq q)\ \Rightarrow\ \{p\} \sqsubseteq \{q\} \\ (p \supseteq q)\ \Rightarrow\ [\,p\,] \sqsubseteq [\,q\,]\end{array}$$

This makes it possible to rewrite context assertions and assumptions into a more manageable form.

5.4 Extending the Refinement Calculator

We have implemented the approach for handling context information described above as an extension to the Refinement Calculator tool. The extension includes six transformation rules for working with context assertions and assumptions as well as a number of window rules that allow us to refine subcomponents using the accumulated context information.

5.4.1 Transformation Rules for Context Handling

Typically, the work with context assertions (assumptions) within the Refinement Calculus framework can be divided into three separate steps: 1) introduction of a context assertion (assumption) at some place of our program, 2) propagation of the introduced context assertion (assumption) through program statements, and 3) elimination of the context assertion (assumption) after the refinement is done.

For context assertions these three steps are as follows:

- we introduce a context assertion $\{true\}$ (usually at the beginning of the program text);
- we propagate the context assertion forward, accumulating context information;
- we eliminate the context assertion when it is not needed anymore.

Dually, for context assumptions we have the following steps:

- we introduce a context assumption-assertion pair at some place of our program;
- we propagate the context assumption backwards taking into account the preceding context;
- we eliminate the context assumption when it has the form $[true]$.

All these steps are implemented as separate transformation rules. For context assertions, we have the transformation rules called `ADD_ASSERTION`, `PUSH_ASSERTION`, and `DROP_ASSERTION`, while, for context assumptions, the transformation rules are `ADD_ASSUMPTION`, `PULL_ASSUMPTION`, and `DROP_ASSUMPTION`, respectively. All transformation rules rely on corresponding HOL theorems. For example, the transformation rule for introducing context assumptions (`ADD_ASSUMPTION`) is derived from the following HOL theorem, which we have proved in the HOL refinement calculus theory:

```
⊢ ∀c p.  c ref [p];{p};c
```

where `c` is an arbitrary statement and `p` is a state predicate. The transformation rule just specialises this theorem according to the current focus (the statement `c`) and the context assumption (the predicate `p`) supplied by the user.

Similarly, the context assumption eliminating transformation rule `DROP_ASSUMPTION` relies on the following HOL theorem:

```
⊢ ∀c.  [true];c = c
```

The corresponding theorems for context assertions are dual.

The context propagation transformation rules for context assertions and context assumptions (`PUSH_ASSERT` and `PULL_ASSUMPTION` respectively) use the propagation rules outlined in Lemmas 5.1-5.3. Each rule is proved as a separate HOL theorem. For example, the theorems for propagation of context assertions and context assumptions into conditional (if-then-else) statement are as follows:

```
⊢ ∀p g c1 c2.  {p}; if g then c1 else c2 fi  ref
       if g then {p andd g};c1 else {p andd not g};c2 fi

⊢ ∀p g c1 c2.  if g then c1 else c2 fi; [p]  ref
       if g then c1;[p] else c2;[p] fi
```

The transformation rules `PUSH_ASSERT` and `PULL_ASSUMPTION` analyse the current focus, distinguishing two basic cases – propagation of the assertion (assumption) through a sequential composition (the focus of the form $\{p\}$; S or S; $[p]$) and propagation of the assertion (assumption) from inside of conditional and block statements. After this they try to find suitable instantiations for universally quantified variables in the appropriate propagation theorems. The result is a HOL theorem that can be used to transform the current focus according to one of the propagation rules of Lemmas 5.1-5.3.

5.4.2 Window Rules for Context Handling

In Section 5.2.3 we presented the rules for refinement in context for concrete statements. These rules were presented as inference rules. The hypothesis part of such a rule is of the form $p \vdash e\, R\, e'$ (where R is a reflexive and transitive relation) and the conclusion part is of the form $\vdash \{p\}; S[e] \sqsubseteq \{p\}; S[e']$. All these rules are implemented as window rules in the Refinement Calculator. Let us explain how they work.

If our current focus is the expression $\{p\}; S[e]$ and we want to open a subwindow on subexpression e, the system starts a subderivation with focus e, relation R and assumption p. When we finish the subderivation (after expression e has been transformed to some e' using the context information p), the system generates the theorem $\vdash \{p\}; S[e] \sqsubseteq \{p\}; S[e']$ and automatically transforms the current focus to $\{p\}; S[e']$. Thus, window rules allow us to refine program components using accumulated context information in a very convenient way.

5.4.3 Example

Let us try a simple program refinement example with the Refinement Calculator using the context assumption technique described above. Our initial program is entered as a starting point for the derivation (`ex` is the name of the derivation):

```
program ex var x,y:num.
```

```
x:=x'.x'=0 ∨ x'=1; if x>0 then x:=x-1 else skip fi; y:=x+1
```

Suppose we want to refine the last statement `y:=x+1` and we expect (guess) that the value of x should be 0 at this point of execution. Therefore, we focus on the last statement and introduce the context assumption by selecting the command `ADD_ASSUMPTION` from the pull-down menu and entering `x=0` in the dialogue box. This gives us the following focus:

```
[x=0]; {x=0}; y:=x+1
```

Now we can refine the assignment statement using the context assertion `x=0`. In order to do this, we click on the right-hand side of the assignment `y:=x+1`. The system then starts a subderivation using the window rule for context refinement of an assignment statement. The subderivation has the focus `x+1` and the assumption `x=0`, while equality is the relation to be preserved. Using the assumption and basic HOL arithmetics, we can rewrite the focus to `1`. Closing the subderivation, we have the following focus:

```
[x=0]; y:=1
```

Closing window once more time yields the following refinement of the initial program:

```
program ex var x,y:num.
  x:=x'.x'=0 ∨ x'=1;
  if x>0 then x:=x-1 else skip fi; [x=0]; y:=1
```

Now we need to get rid of the introduced context assumption [`x = 0`]. We try to do it by propagating it backwards (with the command `PULL_ASSUMPTION`). The first propagation gives us the focus as follows:

```
x:=x'.x'=0 ∨ x'=1;
if x>0 then x:=x-1;[x=0] else skip;[x=0] fi; y:=1
```

Focusing inside the conditional statement, we propagate the assumption backwards for every branch:

```
x:=x'.x'=0 ∨ x'=1;
if x>0 then [x-1=0];x:=x-1 else [x=0];skip fi; y:=1
```

Focusing on the whole conditional statement, we can now propagate the assumptions to the outside:

```
x:=x'.x'=0 ∨ x'=1;
[(x > 0 ∧ (x - 1 = 0)) ∨  (¬(x > 0) ∧ (x = 0))];
if x>0 then x:=x-1 else skip fi; y:=1
```

Opening a window on the assumption predicate (to do this, the system uses the appropriate window rule) and using basic HOL arithmetics, we can easily simplify it to $(\mathtt{x} = 1) \vee (\mathtt{x} = 0)$:

```
x:=x'.x'=0 ∨ x'=1;
[(x = 1) ∨ (x = 0)];
if x>0 then x:=x-1 else skip fi; y:=1
```

After one more propagation (through the nondeterministic assignment), the following focus is constructed:

```
[ !x'. (x' = 0) ∨ (x' = 1) ⇒ (x' = 1) ∨  (x' = 0) ];
x:=x'.x'=0 ∨ x'=1;
if x>0 then x:=x-1 else skip fi; y:=1
```

The assumption predicate is obviously true. Therefore, after simplification (rewriting using built-in theorems of HOL) we have on the screen:

```
[true];
x:=x'.x'=0 ∨ x'=1;
if x>0 then x:=x-1 else skip fi; y:=1
```

Finally, in the last step we eliminate the context assumption with the command `DROP_ASSUMPTION`. After closing window, we arrive at the resulting program:

```
program ex var x,y:num.
  x:=x'.x'=0 ∨ x'=1;
  if x>0 then x:=x-1 else skip fi; y:=1
```

5.5 Conclusions

In this chapter we have described the HOL mechanisation of two dual approaches for handling context information within the refinement calculus framework. They show how information relevant for total correctness and

refinement can be transported from one place of a program to another.

The idea of using state assertions as a part of the statement language was introduced in Back's original formulation of the refinement calculus [4]. The basic rules for propagating context assertions into subcomponents were also given there. The approach was extended and generalised considerably by Back and von Wright in [15], taking also into account the dual notion of context assumption statements. This dual way of handling context information was originally introduced by Morgan [83].

We have shown how context handling rules are implemented and used for program derivations within the Refinement Calculator tool. We reuse the standard feature of the window inference system that permits us to use context information while transforming a subterm. Storing accumulated context information in context assertions allows us to easily add this information while starting a subderivation on some program subcomponent. The use of context assumptions provides us with additional flexibility while handling context information in refinement proofs.

Handling context assumptions can also be useful in other situations. The Refinement Calculator has a tool for data refinement (the Dataref extension described in Section 3.5) where certain data refinement transformations introduce explicit assumption statements, rather than proof obligations. These assumptions can be propagated and discharged using the tool described here.

Grundy[49] has used window inference to handle refinement by using a simpler program model proposed by Hehner[53]. This approach treats programs (including specifications) as predicates and uses implication to model refinement. All reasoning is effectively carried out in standard predicate calculus. That makes it possible to directly use window hypotheses to propagate context information through a program. Our approach allows us to handle context information while working with program statements modelled as predicate transformers, without having to reduce everything to predicates.

Our approach of storing context information in assertion statements and then reusing the window inference system to do the rest is similar to the approach implemented by Nickson and Hayes[90] in the Ergo theorem prover developed in Queensland University. However, due to the syntactic nature of the modelled program state and variables, the information about types and names of program variables also needs to be stored in context in their approach. This is not necessary using our approach since strong typing is maintained by the HOL system and program variables are ordinary (higher-order) variables of the underlying logic.

Chapter 6

Modelling Procedures

6.1 Introduction

Procedures are basic units of modularisation which permit us to abstract a certain piece of code and give it a name. Flexibility of procedures is provided by parametrisation mechanisms that make it possible to adapt the procedure code in different places of the main program. Procedures are common in imperative programming languages, since they provide a convenient and simple way of writing efficient and clearly structured programs.

In this chapter we present an approach for modelling procedures (as they occur in imperative programs) in the refinement calculus theory. This allows for formal reasoning about procedures in the weakest precondition framework. We have implemented this approach in the mechanisation of the refinement calculus theory in the HOL system. This makes it possible to prove a number of correctness and refinement properties of procedures as HOL theorems. Finally, we show how our method for procedure handling can be integrated into the Refinement Calculator tool. For this purpose we have extended the Refinement Calculator with the necessary infrastructure for working with procedures.

6.2 Procedures in the Refinement Calculus

The general purpose of a procedure is to abstract a certain piece of code, giving it a name and then adapting it (through parameters) in different places of the program. Therefore, procedures can be seen as fragments of program code that can be used (and reused) in different contexts.

Let us forget for a while about procedure parametrisation mechanisms.

Then a procedure declaration is nothing more than a way of giving a name to a piece of a program, and a procedure call is just a way of using this name as a shorthand for the procedure body. In higher order logic this can be expressed using the *let* construct. In our case, a procedure P with no parameters can be expressed simply as

$$let\ P\ = < procedure\ body > \quad in \quad < main\ program >$$

Parametrisation provides flexibility for procedures because procedure execution can be modified by supplying different values for its parameters. In programming practice, four kinds of parameters are used – *value*, *result*, *value-result* and *reference* parameters.

For simplicity (and without losing generality), we restrict ourselves to value and reference parameters because value-result and result parameters can always be simulated as reference parameters. For the same reason we do not allow global variables inside the procedure body. Therefore, the only means of modifying the execution of a procedure is its parameters. Modelling procedures in this way, we can reason about them independently (separately) of the main program.

The procedure body is a fragment of program code, so it can be modelled as a predicate transformer in the refinement calculus formalisation. The program state that a procedure is working on is composed from the procedure parameters. A procedure call leads to execution of the procedure body on an "adapted" global state. The actual procedure parameters provide the necessary information for this adaption.

As mentioned before, a procedure works on its own state consisting of the value and reference parameters. Its execution can only affect the global state by changing variables that are present in the reference parameter list of a procedure call. Other variables of the global state should remain unchanged (no side effects possible). To achieve this effect, we rearrange the global state in such a manner that the procedure execution can be expressed as a parallel execution of skip (on the variables of the global state that are unaffected by the procedure execution) and the procedure body (on the state composed from the value and reference parameters). Since the value parameters are "new" variables they should be created (and initialised according to the concrete value expressions in the procedure call) by a block construct. So the call to a procedure `P` with value expressions `x` and reference parameters `y` can be unfolded in the following way:

$$
\begin{array}{l}
\textbf{call}\ P\ (\textbf{val}\ e;\ \textbf{var}\ y)\ = \\
\quad |[\textbf{var}\ x \mid x\ =\ e. \\
\qquad < \mathit{rearrange\ state} >;\ (\textbf{skip} \mid\mid S);\ < \mathit{restore\ state\ order} > \\
\quad]|
\end{array}
$$

Here x are the formal names for the value parameters inside the block and S is the body of the procedure P. In the case when a procedure does not have value parameters, unfolding of the procedure call does not produce a block around the adapted procedure body.

6.3 Implementing Procedures in the HOL System

In this section we show how we can extend the existing formalisation of the refinement calculus in HOL with procedures. We explain how a procedure and a procedure call can be defined in the HOL system, and what basic properties of monotonicity, correctness and refinement can be proved on the basis of these definitions.

6.3.1 Defining Procedures

As explained before, a procedure declaration can be modelled using the *let* construct in higher order logic. There is a corresponding `LET` constant in the HOL system. However, in our system `LET` can be used for different purposes. For example, variables are also declared by using the `LET` construct which associates the variable name with the corresponding state projection function. In order to distinguish a procedure declaration from other applications of `LET`, we copy the definition of `LET` giving it the new name – `PLET`.

$\vdash_{def}$ `PLET = (λf x. f x)`

So `PLET` takes two arguments – the first one is a function taking a predicate transformer (the procedure body) and returning a predicate transformer (the program which can contain procedure calls), and the second one is the body of a procedure we are defining. The type of the procedure state is of the form `T1#T2` where `T1` and `T2` are the types representing the tuples of value and result parameters respectively. In the case when a procedure does not have value or reference parameters, the HOL type `one`[1] is used

[1]The HOL type `one` contains only one object — the constant `one`.

to indicate this. Therefore, the type of the procedure body unambiguously determines the number and the types of the procedure parameters.

Recursive procedures can be defined using the fixpoint operator μ. In this case the body of the procedure P is of the form (μP. c[P]) where c[P] is a program fragment containing recursive calls to P. We explain handling of recursive procedures in detail in Chapter 9.

Though the PLET construct is used to define only one procedure at a time, its structure allows for nesting. The result of an application of PLET is a predicate transformer (a program), so it can be used instead of the main program in another PLET application. Therefore, by repeated application of PLET we can define as many procedures as we need. Then the following nested structure of PLET definitions is constructed:

```
PLET (λP1.
    PLET (λP2.
       PLET (λP3. ...)
       <P3 body>)
    <P2 body>)
<P1 body>
```

The nesting level of each procedure determines the part of the program where the procedure is "visible", i.e., can be called. The outermost procedure has the biggest scope.

Therefore, procedure bodies can contain calls to other procedures if the latter ones are in scope. This means that the order of procedure definitions is very important when determining procedure dependencies. However, no circular procedure calls are possible in our formalisation.

6.3.2 Adaption and Procedure Call

Before defining a procedure call, we need to define the adaption operator. This operator adapts a statement c operating on the state Σ to the bigger state Γ. Adaption works in the following way: the current state Γ is rearranged to be of the form $\Sigma' \times \Sigma$, where Σ' is a state composed from the state components that should not be affected by execution of c, then the lifted statement lift c is executed on $\Sigma' \times \Sigma$, and, finally, the state order is restored back. Adaption is controlled by the state functions (state reorderings) f and g which are inverses of each other, i.e., $g \circ f = id = f \circ g$.

$\vdash_{def}$ `adapt f g c = (assign f); (lift c); (assign g)`

Recall that lifting models parallel execution of the form $skip \| c$, where c is a statement operating on the second part (projection) of the program state. The HOL definitions for lifting and parallel execution are given in Section 3.3.

The notion of adaption is not novel. It was introduced by vonWright in [109] and later used for reasoning about reactive systems in HOL[71]. Theoretical properties of adaption are studied in [16].

Now we are ready to define a procedure call. It is defined as a block which introduces new local variables for each value parameter and initialises them with the value expressions of the procedure call. The body of the block contains the adapted procedure body. It is easy to see that this definition directly corresponds to the intuitive interpretation of a procedure call given on p.75.

$\vdash_{def}$ `call c f g V = block (λv. FST v = V (SND v)) (adapt f g c)`

The additional argument `V` is an initialisation function for the local variables of the block, i.e., the value parameters of a procedure. It takes a global state as argument and returns a tuple of the value parameters initialised according to the value expressions of a procedure call.

Let us try a trivial example of a small program with a procedure. The following program just assigns `2` to the first state component (variable `z`) by calling the procedure `p1`. The procedure `p1` has one value parameter `x` and one reference parameter `y`. In Pascal-like notation it can be written as follows:

```
program test
  procedure p1(val x : num; var y : num) =
        y := x
in
  var z, u : num.
  u := 0; p1(u + 2, z)
```

The internal representation of this program in our formalisation would look as follows:

```
PLET
 (λp1. VLET (λz. VLET (λu.
    assign (λs. z s, 0) seq
    call p1
         (λs. SND(SND s), FST s, FST(SND s))
         (λs. FST(SND s), SND(SND s), FST s)
         (λs. u s + 2))
       SND) FST)

 (VLET (λx. VLET (λy.
    assign (λs. x s, x s)) SND) FST)
```

or, using the simpler `let` syntax, the same term could be presented as:

```
let p1 = (let x = FST in (let y = SND in assign (λs. x s, x s)))
 in
    (let z = FST in
      (let u = SND in
        assign (λs. z s, 0) seq
        call p1
           (λs. SND(SND s), FST s, FST(SND s))
           (λs. FST(SND s), SND(SND s), FST s)
           (λs. u s + 2))
    )
```

Here we see how the `LET` construct is used for two different purposes. `VLET`s associate variable names with the appropriate state projection functions. `PLET` connects the declared procedure `p1` and the program part containing calls to the procedure `p1` into one program.

The functions (λs. SND(SND s), FST s, FST(SND s)) and (λs. FST(SND s), SND(SND s), FST s) are state adaption functions which should be supplied to describe the procedure call. Using the tupled λ-abstraction of the program state, the first function can be rewritten as (λ(x, z, u). u, x, z) where x is a new local variable representing the value parameter of the procedure call. The function rearranges the program state in such a way that the procedure p1 can be executed on its second component (i.e., (x, z)) which consists of the value and reference parameters of the procedure call. The second function restores the order of program state components.

The state adaption functions do not look very pretty in this example, and in more realistic examples they tend to become even more ugly. In Section 6.4 we show how the Refinement Calculator tool can hide this internal representation by the use of a parser and a pretty-printer.

6.3.3 Basic Properties

We shall now show a number of properties of the PLET and call operators that we have proved as HOL theorems. We start with a few obvious ones. It is not surprising that the procedure call operator preserves monotonicity.

```
mono_call =
    ⊢ ∀c f g V. monotonic c ⇒ monotonic (call c f g V)
```

The procedure call operator is also monotonic with respect to refinement of the procedure body.

```
mono_call_body =
    ⊢ ∀f g c c′ V.  c ref c′ ⇒ call c f g V  ref  call c′ f g V
```

Another desirable property would be the possibility to refine a procedure body independently of the main program. In other words, we need a property that states that any refinement of the procedure body leads to a refinement of a program containing calls to that procedure as well. This property can be proved as the following theorem:

```
ref_proc =
   ⊢ ∀f c c′.
      regular f ⇒
      monotonic c ⇒
      monotonic c′ ⇒
      c ref c′ ⇒
      (PLET f c)  ref  (PLET f c′)
```

The regularity property can be automatically proved by decomposition using the fact that all program constructors of our language (such as sequential composition, the block statement, the conditional statement and so on) are regular if we consider them as functions on predicate transformers. We can prove that the procedure call operator itself is regular:

```
regular_call =
    ⊢ regular (λc. call c f g V)
```

This property is important for the automatic proof of regularity in cases when there are nested calls.

6.3.4 Correctness Proofs with Procedures

In order to prove some correctness property of a program containing procedure calls, we use Hoare logic to decompose the proof in the ordinary way.

When we get to the level when we need to prove a correctness assertion for a procedure call, the following theorem is used. It states that the correctness assertion for the procedure call can be reduced to a correctness assertion for the procedure body.

```
correct_call =
     ⊢ ∀f g c p q V.
       inverse f g ⇒
       (correct p (call c f g V) q  =
        (∀w. correct (λu. (p o SND o g) (w,u) ∧
                          ((FST o g) (w,u) = (V o SND o g) (w,u)))
                     c (λu. (q o SND o g) (w,u))
```

The assumption of this theorem requires that the adaption functions `f` and `g` should be inverses of each other.

This theorem shows that predicates `p` and `q` on the global state can be translated to predicates (`p o SND o g`) and (`q o SND o g`) respectively. The additional conjunct in the derived precondition of the procedure body expresses the requirement that the procedure state components corresponding to the value parameters should be initialised according to the value expressions (encoded in the function `V`) of a procedure call. The predicates (`p o SND o g`) and (`q o SND o g`) are defined on the state composed of the global variables that are not present in the reference parameter list of procedure call (`w`) and the variables of the internal procedure state (`u`). Since the variables `w` are not affected by the procedure execution, the correctness triple on the right hand side is true for any value of them.

Sometimes we are faced with the opposite task. We know that the procedure body satisfies a certain correctness assertion and we want to prove a correctness property of a procedure call with actual parameters. The following theorem shows how this correctness property can be derived:

```
correct_body_call =
      ⊢ ∀c f g V p q.
      inverse f g ∧ monotonic c ⇒
         (correct p c q ⇒
          correct (λu. (p o SND o f) (V u,u))
                  (call c f g V)
                  (λu. ∃x. (q o SND o f) (x,u)))
```

Recall that a procedure call can be unfolded as a block with the adapted procedure body. The local variables of this block correspond to value parameters initialised with the actual value expressions of the call. The predicates (`p o SND o f`) and (`q o SND o f`) are defined inside this block.

Note that the state components corresponding to the value parameters get existentionally quantified in the postcondition (λu. $\exists$x. (q o SND o f) (x, u)). The reason is that it is impossible to reconstruct the value parameter state from the global state after a procedure call. In syntactic proof rules for procedures, which allow us derive a correctness property of a procedure call from a correctness property of the procedure body, this problem is usually solved by prohibiting the use of value parameters in the postcondition. As an example of such a syntactic proof rule, Gries's rule[46] can be mentioned here.

This rule states that, if the procedure body is specified by the correctness triple $\{P\}\ B\ \{Q\}$, then the following correctness property holds for a procedure call:

$$\{P^{x,y}_{a,b}\ \wedge\ I\}\ \ p(a,b,c)\ \ \{Q^{y,z}_{b,c}\ \wedge\ I\}$$

where x are value parameters, y are value-result parameters, z are result parameters and I is an invariant expression on the variables that are not affected by the procedure execution.

Let us try a simple example to compare our theorem and the syntactic proof rule. Suppose we have a procedure P with the value parameter x and the reference parameter y specified by a correctness property:

```
correct (λ(x,y).x = x0) c (λ(x,y).y = x0*x0 + 1)
```

where x0 is a specification variable which allows us to refer to the initial value of the value parameter x in the postcondition.

Suppose we want to derive a correctness property of the procedure call P(z + 1, w) in the state (w, z). In our formalisation this procedure call corresponds to:

```
call P (λs. SND(SND s), FST s, FST(SND s))
       (λs. FST(SND s), SND(SND s), FST s)
       (λs. SND s + 1)
```

From now on, for readability purposes we use tupled λ-abstraction of a program state (instead of the actual implementation of program variables using VLETs). For example, the procedure call above can be presented as:

```
call P (λ(x,w,z). z, x, w)
       (λ(z,x,w). x, w, z)
       (λ(w,z). z + 1)
```

Specializing our correctness theorem with concrete instances for adaption functions f and g, a value parameter initialisation function V, a precondition p and a postcondition q, we get (after automatically discharging the

assumptions and simplifying) that the procedure call $P(z+1, w)$ is correct with respect to the precondition $(\lambda(w, z) \bullet z{+}1 \ = \ x0)$ and the postcondition $(\lambda(w, z) \bullet \ w \ = \ x0 * x0 \ + \ 1)$.

Applying Gries's syntactic proof rule yields the same result:

$$\{(z+1 \ = \ x0) \ \wedge \ I\} \ \ B \ \ \{(w \ = \ x0 * x0 \ + \ 1) \ \wedge \ I\}$$

where I is an invariant expression on the variables that are not affected by the procedure execution. Our definition of a procedure call guarantees that variables that are not present in the reference variable list stay unchanged during execution of a procedure call. Therefore, any invariant property on these variables is obviously preserved.

6.4 Extending the Refinement Calculator with Procedures

Our approach for procedure handling was integrated into the Refinement Calculator tool as one of its extensions. The procedure extension enhances the functionality of the tool by extending the parser and pretty-printer with new syntax for procedures and their calls, and by providing new transformation and window rules for working with procedures.

6.4.1 The Syntax

The Refinement Calculator contains a parser and a pretty-printer which allow users to interact with the system using the syntax they are accustomed to. We extended them allowing to use the commonly used syntax for procedures and procedure calls as well.

We adapt the following standard syntax when working with procedures. A procedure definition starts with the keyword `procedure` followed by the procedure name and the list of arguments. The keywords `val` and `var` indicate the beginning of the value and reference parameter lists respectively. All procedure definitions should be presented before the main program part. Therefore, the structure of the program that is entered at the beginning of a session can be, for example, as follows:

```
procedure P1 (val x,y:num var z:bool)
  <the body of the procedure P1>
procedure P2 (val x:num var z,w:num)
  <the body of the procedure P2>
...
procedure PN (...)

<main program>
```

The order of procedure definitions determines the scope of procedures. For example, the procedure `P2` can call the procedure `P1`, but not other way around. The body of the procedure `PN` can contain calls to any previously defined procedure.

The value parameter list can be missing. That means that the procedure in question does not have value parameters.

The syntax of a procedure call is standard – a procedure name followed by the list of actual parameters in parentheses. One obvious inconvenience of using our mechanisation of procedures is that it is necessary to supply concrete adaption functions for every procedure call. This problem is solved by extending the parser so that it automatically generates adaption functions using the actual definition of a procedure and the actual parameters of a procedure call. The pretty-printer then prints a procedure call using the usual syntax, hiding the internal representation of the HOL term.

6.4.2 Transformation Rules for Procedures

We have added three transformation rules for working with procedures. The first one introduces (declares) a new procedure into a program, and the others allow us to introduce or eliminate a procedure call.

Definition of a New Procedure

Starting our session we enter a program (or a specification) together with the procedures that we are planning to use. However, sometimes it is necessary to introduce a new procedure when the session is already on its way. Our first transformation rule (called `Define Procedure`) takes care of this. During this transformation the main program is replaced by

$$\texttt{PLET}\ (\lambda\, \texttt{New\_Proc\_Name}.\ <\texttt{main program}>)\ <\texttt{new procedure body}>$$

where `New_Proc_Name` is the name of the new procedure. Since the main program does not contain any calls to the procedure that has just

been introduced, such a replacement is always valid (formally, $\forall c\, t \bullet\ c\ =\ (\lambda x \bullet\ c).\, t$, if x does not occur in c).

The term

```
PLET (λ New_Proc_Name. < main program >) < new procedure body >
```

is embedded by the procedures that were defined previously. This means that the new procedure can use (call) all previously defined procedures.

Let us try a simple example to demonstrate what is actually going on. Suppose we are working with a program containing one procedure called `Inc`:

```
procedure Inc (val x:num var y:num)
   y := x+1
<main program>
```

which corresponds to the internal HOL representation:

```
PLET (λ Inc.<main program>) (assign(λ (x,y).x,x+1))
```

If we select the transformation rule Define Procedure from the Refine menu, we are asked to enter a new procedure. For example, if we want to define a procedure for calculating the square of a natural number, we can enter:

```
procedure Square (val x:num var y:num)
   y := x*x
```

This procedure becomes the input for the parser which generates the internal HOL representation for this particular procedure. If it contains calls to already defined procedures, all arguments for the HOL constant `call` (the procedure body, the adaption functions and the value parameter initialisation function) are automatically generated as well. The new procedure is put between the procedures defined earlier and the main program. In our example, after the transformation rule has been executed, the corresponding HOL theorem is proved and the internal HOL representation of the focus is changed to the following:

```
PLET (λ Inc.
     PLET (λ Square.<main program>) (assign(λ (x,y).x,x*x)))
 (assign(λ (x,y).x,x+1))
```

Unfolding of a Procedure Call

The second transformation rule (called `Unfold Procedure Call`) implements the "copy rule" for procedure calls. A procedure call is unfolded to a block containing the adapted procedure body where all formal parameters are replaced by the actual parameters. In some cases (for example, if the procedure does not have any value parameters), the block is automatically eliminated.

Let us try a simple example to see what happens. Suppose we have the following program on the screen:

```
procedure swap (var x,y:num)
  x,y:=y,x
procedure max (val x,y:num var z:num)
  if x < y then z:=y else z:=x fi

var z,w:num.
... swap(z,w); ... max(5,z+2,w) ...
```

Focusing on the call to the procedure `swap`, we can unfold it using the transformation rule `Unfold procedure call`. As a result, the procedure call in the focus is automatically replaced by the multiple assignment `z,w := w,z`.

Unfolding the call to the procedure `max` would result in the following block:

```
|[ var x,y|(x=5) ∧ (y=(z+2)). if x < y then w:=y else w:=x fi ]|
```

The local variables of the block correspond to the formal value parameters of the procedure `max` and are initialised according to the concrete value expressions of the procedure call. The formal reference parameter `z` got substituted by the actual variable `w`. Using context propagation rules and the techniques for refinement in context (see Chapter 5), we can replace the local variables of the block with their initial values given in the initialisation predicate. Then, since the local variables are not used anymore in the block body, the block can be eliminated using the `Block Elim` transformation rule. After these simplifications, the focus is as follows:

```
if 5 < (z+2) then w:=z+2 else w:=5 fi
```

In many cases, such a simplification can be done automatically. However, in cases when value parameters are changed inside of the procedure body, the block cannot be eliminated so simply. In such cases, the transformation rule just unfolds a procedure call to the corresponding block, leaving all further simplification to the user.

Introduction of a Procedure Call

The last transformation rule (called `Introduce Procedure Call`) does the opposite of the previous one. It tries to replace the current focus (the fragment of program code) by a procedure call supplied by the user. If there is a direct correspondence between the piece of code in the focus and the unfolded procedure call, then a replacement takes place. Otherwise, a replacement still takes place but an additional proof obligation is generated for the user. The proof obligation should be discharged at some point during the session.

For example, let us have the following program fragment in the focus:

```
if x ≥ y then z:=x else z:=y fi
```

and assume that we want to replace it with the procedure call `max(x, y, z)` (where `max` is the procedure from the previous example).

When we select the transformation `Introduce Procedure Call` from the menu, a dialog window pops up, and we are asked to supply the exact form of a procedure call. Then the transformation rule replaces the current focus with our procedure call `max(x, y, z)`. However, the additional proof obligation is generated:

```
?- if x ≥ y then z:=x else z:=y fi  ref
   if x < y then z:=y else z:=x fi
```

6.4.3 Procedure Refinement

The theorem `ref_proc` (Section 6.3.3) shows that `PLET` is monotonic with respect to refinement of its second argument (the body of a new procedure) provided certain conditions about regularity and monotonicity hold. Since all these conditions can be automatically discharged, we can add a new window rule which allows us to focus on the procedure body. After we finish our subderivation proving some refinement of the procedure body, the new window rule guarantees that our refinement is actually a refinement of the whole program.

Let us look at a simple example. Suppose we are working with a program containing the procedure `Swap`.

```
procedure Swap (var x,y:num)
   x,y:=y,x

<main program (with possible calls to Swap)>
```

Focusing on the body of the procedure `Swap`, we can refine the multiple assignment $\mathtt{x}, \mathtt{y} := \mathtt{y}, \mathtt{x}$ to the block $|[\mathtt{var\ t : num.\ t := x;\ x := y;\ y := t}]|$. Closing the window yields the new focus:

```
procedure Swap (var x,y:num)
   |[var t:num. t:=x; x:=y; y:=t ]|

<main program>
```

As a result, we automatically proved that a program containing the refined `Swap` procedure is a refinement of our initial program.

The `PLET` construct is also monotonic with respect to a refinement of the main program which is the function body of its first argument. This is quite obvious since, when refining the main program, we actually prove the refinement for any procedure body:

$$\vdash \forall \mathtt{x.\ f\ x\ ref\ f'\ x}$$

where `f` is the first argument of `PLET`. Therefore, we can add the second window rule which guarantees that a refinement of the main program leads to the refinement of the whole program as well.

6.5 Conclusions

In this chapter we presented an approach for modelling procedures (as they occur in imperative programs) in the weakest precondition framework. This approach was implemented in the mechanisation of the refinement calculus theory in the HOL system.

In the classical works of the refinement calculus [5, 80, 84] procedures are treated in a syntactical way, meaning that procedures are syntactically substituted in all places where calls to them occur. Therefore, all correctness and refinement rules for procedures are syntactical as well.

In their recent book[15] R.Back and J.von Wright presented a new axiomatic approach for modelling the program state and program variables. It allows a simple and elegant way to define procedures and procedure calls. A procedure is just a lambda abstraction taking program variables and returning a predicate transformer (the procedure body). A procedure call then corresponds basically to the application of this lambda abstraction to the actual parameters. The use of a lambda abstraction and a function application to model procedures makes this approach similar to ours. However, we use a function application (the *let* construct) to introduce a procedure into

a program while Back and Wright are using it to bind the formal and the actual parameters in a procedure call.

The approach presented in this chapter allows us to derive correctness and refinement properties of procedures and procedure calls rather that just postulate them. This constitutes the main contribution of this work. The properties of procedures are proved while we are still inside the logic (in contrast to the syntactical rules that are meta-logical).

The implementation of procedures as the extension of the Refinement Calculator tool allows us to hide the aspects of the mechanisation that make it hard to use in practice. The adaption functions needed to describe a procedure call are automatically generated during parsing, and most assumptions (such as monotonicity or regularity) are also automatically proved and discharged.

We should admit that the use of our implementation of procedures was illustrated only with small examples. In the next chapter we try to rectify this by showing a more realistic example of program derivation.

Chapter 7

Example: Array Sorting

7.1 Introduction

In this chapter we present an example of program derivation using the Refinement Calculator tool. The `Context`, `Correctness` and `Procedure` extensions of the tool are used in this derivation. Therefore, they should be loaded before the derivation is started.

Our goal is to derive a program for sorting arrays using the minimal element insertion algorithm, i.e., an array is sorted by first finding the minimal element of an unsorted array, swapping it with the first element and then repeating the process for the remaining part of the array.

Arrays[25] can be modelled by defining a new HOL type $(\alpha)array$ where α is a type variable. The following operations are defined for arrays:

$$\begin{aligned} asize: &\quad (\alpha)array \rightarrow num \\ lookup: &\quad (\alpha)array \rightarrow num \rightarrow \alpha \\ update: &\quad (\alpha)array \rightarrow num \rightarrow \alpha \rightarrow (\alpha)array \end{aligned}$$

The meaning of these operations is following. If `a` is an array, then `asize a` is the size of array `a`, `lookup a i` is the array element indexed by `i`, provided `i` $<$ `asize a`, and the operation `update a i x` returns the new array which is the same as `a` but with the value x in the position indexed by `i`, if `i` $<$ `asize a`. Arrays are indexed from 0 to (`asize a`) $-$ `1`.

7.2 Initial Specification

To formulate an initial specification of our program, we need definitions of a sorted subarray and a permutation of an array. They can be defined in

the following way:

```
⊢def sorted (a:(num)array) (r:num→bool) =
   (∀i. (r i) ⇒ i < (asize a)) ∧
   (∀i j. (r i) ∧ (r j) ∧ (i < j)) ⇒ (lookup a i ≤  lookup a j))

⊢def perm (a1:(α)array) (a2:(α)array) =
  (asize a1) = (asize a2) ∧
  (∃f. (injective f) ∧
       (∀i. i<(asize a1) ⇒ (f i)<(asize a1) ∧
                           (lookup a1 i) = (lookup a2 (f i))))
```

Here `sorted a r` stands for "the array `a` is sorted on the index set `r`"and `perm a1 a2` stands for "the array `a1` is a permutation of the array `a2`".

Now we are ready to enter our initial specification:

```
var a:(num)array.
  a:= a′. (sorted a′ (λi. i<(asize a))) ∧ (perm a a′)
```

7.3 Introducing Local Variables and a Do-Loop

We implement this specification using a loop that traverses the array starting at position 0. The array is traversed using an indexing variable `i : num`. Every iteration of the loop guarantees that the array slice $\mathtt{a}[0..\mathtt{i}-1]$ becomes sorted. This can be achieved in the following way: we find the minimal element in the subarray $\mathtt{a}[\mathtt{i}..(\mathtt{asize\ a})-1]$, swap it with the current $\mathtt{i}^{\mathtt{th}}$ element of the array, and, finally, increase the value of `i` by `1`. The loop terminates when `i` reaches the value $(\mathtt{asize\ a})-1$.[1]

Finding the minimal element in a subarray and swapping two array elements can be defined as relations in the HOL system. The definition of `min_in_subarray` relation is the following:

```
⊢def min_in_subarray (a:(num)array) (r:num→bool) k =
  (r k) ∧ (∀j. (r j) ⇒ (j < asize a) ∧ (lookup a k ≤ lookup a j))
```

Thus the term `min_in_subarray a r k` specifies that `k` is the index of the minimal element on the part of the array `a` specified by the index set `r`.

The relation `swap` is defined as follows:

[1] There is no need to execute the loop for the last element of the array (the case $\mathtt{i} = (\mathtt{asize\ a})-1$) since by then the element $\mathtt{a}[(\mathtt{asize\ a})-1]$ should contain the maximal element of the array.

```
⊢def swap (a:(α)array) i j a′ =
  ((asize a′ = asize a) ∧ (i < asize a) ∧ (j < asize a) ∧
   (lookup a i = lookup a′ j) ∧ (lookup a j = lookup a′ i) ∧
   (∀k. ¬(k = i) ∧ ¬(k = j) ⇒ (lookup a k = lookup a′ k)))
```

The term `swap a i j a′` states that the array `a′` is the same as the array `a` with $\mathtt{i}^{\mathtt{th}}$ and $\mathtt{j}^{\mathtt{th}}$ elements swapped.

Now we are prepared to formulate the guard, the body, the invariant and the variant function of our loop:

```
Guard: i < (asize a)−1
Body:  k:=k′.(min_in_subarray a (λii.(ii≥i) ∧ (ii < asize a)) k′);
       a:=a′.(swap a i k a′); i:=i+1
Invariant:
  (sorted a (λii. ii < i)) ∧ (perm a A) ∧
  ((asize a > 0) ⇒ i < asize a) ∧
  ((i>0) ⇒ min_in_subarray a (λii. ii≥(i−1) ∧ ii<asize a) (i−1))
Variant: (asize a) − i
```

The constant `A` here refers to the initial value of the array `a`. The first three conjuncts of the invariant property are quite obvious. The last conjunct expresses the very useful (for further correctness proofs) fact that (except for the first step) the last element of the sorted subarray $\mathtt{a}[\mathtt{0..i}-\mathtt{1}]$ also is the minimal element of the remaining part of the array $\mathtt{a}[\mathtt{i}-\mathtt{1..(asize\ a)}-\mathtt{1}]$.

Before transforming the specification into a do-loop using the transformation rule for loop introduction, we should introduce the variables `i` and `k`, and the constant `A` into our specification. The variables `i` and `k` are introduced as the local variables of a new block, and the constant `A` is introduced as a specification variable in the assertion statement `a = A`. We also initialize the variable `i` by adding the assignment statement `i := 0`.

```
var a:(num)array.
  |[ var i,k:num.
     {a=A}; i:=0;
     a,i,k:= a′,i′,k′. (sorted a′ (λi. i<(asize a))) ∧ (perm a a′)
  ]|
```

Focusing inside the block, we can propagate context information using the transformation rule `PUSH_ASSERTION` from the `Context` menu, and then use the accumulated context for rewriting the body of the nondeterministic assignment statement (see Figure 7.1).

As a result, all references to the initial value of the array `a` are replaced by `A`:

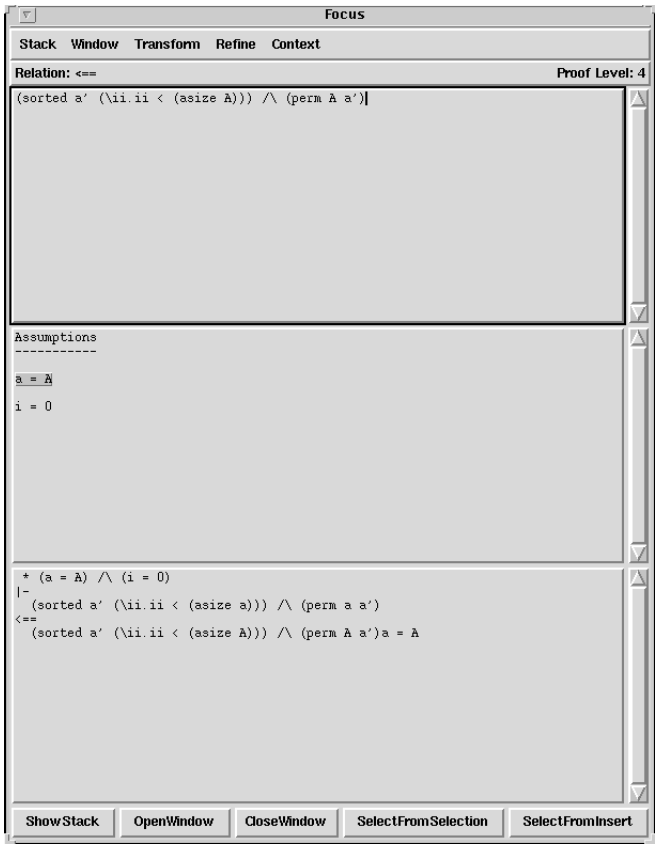


Figure 7.1: Rewriting the body of the nondeterministic assignment

```
{(a=A) ∧ (i=0)};
a,i,k:= a′,i′,k′. (sorted a′ (λi. i<(asize A))) ∧ (perm A a′)
```

Now applying the loop introduction rule (the menu choice `Loop Introduction`) with the guard, the body, the invariant, and the variant described above as arguments, yields the following focus:

```
do i < (asize a)−1 →
  k:=k′.(min_in_subarray a (λii.(ii≥i) ∧ (ii < asize a)) k′);
  a:=a′.(swap a i k a′);
  i:=i+1
od
```

This step generates a number of proof obligations. The proof obligations state respectively that the loop invariant should hold initially, the loop body

should preserve the invariant and decrease the variant, and the invariant and the negation of the guard should imply the postcondition. The second proof obligation is expressed as a correctness assertion (i.e., of the form `pre << loop body >> post`) for the loop body. In order to prove it, we should use transformation rules provided by the `Correctness` extension which allow us to reduce this assertion to a boolean term. Proof obligations are proved in separate subderivations, transforming boolean terms into truth preserving the backward implication relation.

For the sake of brevity, we omit the actual proofs of these proof obligations. We should mention, however, that these proofs require some additional properties of the constants `sorted`, `perm`, `swap`, and `min_in_subarray` proved beforehand. For example, we need the following theorem stating that by swapping loop elements we get a new array that is a permutation of the original:

```
⊢ ∀a a′ i j.
  (i < asize a) ∧ (j < asize a) ∧ (swap a i j a′) ⇒
  (perm a a′)
```

Closing windows gives us a refinement of our initial specification:

```
var a:(num)array.
  |[var i,k:num.
    i:=0;
    do i < (asize a)−1 →
      k:=k′.(min_in_subarray a (λii.(ii≥i) ∧ (ii < asize a)) k′);
      a:=a′.(swap a i k a′);
      i:=i+1
    od
  ]|
```

7.4 Introducing Procedures Min_in_subarray and Swap_in_array

In the next step we implement the relations `min_in_subarray` and `swap` as procedures and replace appropriate nondeterministic assignments that use these relations in the main program by corresponding procedure calls. Selecting the `Define Procedure` transformation rule, we can enter:

```
procedure Min_in_subarray (val a:(num)array; i:num var j:num)
  {i < (asize a)−1};
  j := j′.(min_in_subarray a (λii.(ii≥i) ∧ (ii < asize a)) j′);
```

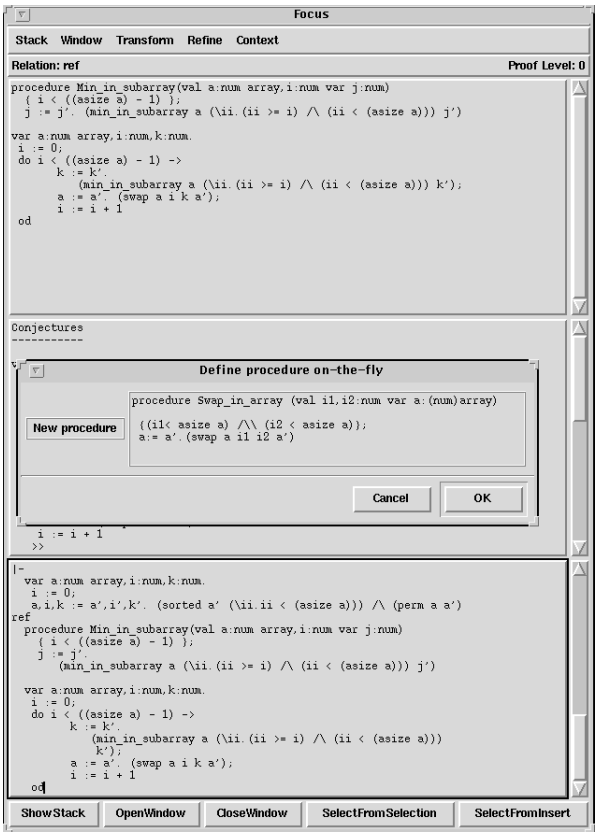


Figure 7.2: Definition of the procedure `Swap_in_array`

In the same way, the new procedure `Swap_in_array` is defined (see Figure 7.2):

```
procedure Swap_in_array (val i1,i2:num var a:(num)array)
  {(i1 < asize a) ∧ (i2 < asize a)}
  a:=a′.(swap a i1 i2 a′);
```

Since there is the direct correspondence between introduced procedures and the corresponding specification (nondeterministic assignment) statements in the loop body of the main program, we can replace them by focusing and selecting the `Introduce Procedure Call` transformation rule from the menu. The dialog window asks us to supply the exact form of a procedure call.

As a result, the following main program is yielded:

```
var a: (num)array.
  |[var i,k:num | i=0.
    while i < ((asize a)−1) do
      Min_in_subarray(a,i,k);
      Swap_in_array(i,k,a);
      i := i+1
    od
   ]|
```

By introducing the procedures `Min_in_subarray` and `Swap_in_array` in our program, we only postponed the task of actual implementation of the corresponding relations `min_in_subarray` and `swap`. In the following sections we show how we can refine the bodies of these procedures into executable code.

7.5 Refining the Procedure Min_in_subarray

Focusing on the body of the `Min_in_subarray` procedure yields the following focus:

```
{i < (asize a)−1};
j := j′.(min_in_subarray a (λii.(ii≥i) ∧ (ii < asize a)) j′)
```

We implement this using a loop which traverses the array starting from position `i + 1`. The array is traversed using an indexing variable `k : num`. The body of the loop ensures that the variable `j` contains the index of the minimal element in the subarray `a[i..k]`.

First of all, we introduce a block with a local variable `k` which is initialised to the value `i + 1` by an assignment statement.

```
|[var k:num.
  k:=i+1; {i < (asize a)−1};
  j,k := j′,k′.(min_in_subarray a (λii.(ii≥i) ∧ (ii < asize a)) j′)
]|
```

We then add an assignment statement which initialises the result parameter `j` to the value `i`, and propagate context information into an assertion before the nondeterministic assignment statement.

```
|[var k:num.
  j:= i; k:=i+1;
  {(i<(asize a)−1) ∧ (j=i) ∧ (k=i+1)};
  j,k := j′,k′.(min_in_subarray a (λii.(ii≥i) ∧ (ii < asize a)) j′)
]|
```

Now we can formulate the guard, the body, the invariant and the variant function of our loop:

```
Guard: k < asize a
Body:  if (lookup a k ≤ lookup a j) then j:=k else skip fi; k:=k+1
Invariant:
   (k ≤ asize a) ∧ (min_in_subarray a (λii. ii≥i ∧ ii<k) j)
Variant: (asize a) − k
```

Supplying these terms as the parameters for the `Loop Introduction` transformation rule results in the following focus:

```
|[var k:num.
  j:= i; k:=i+1;
  do (k < asize a) →
    if (lookup a k ≤ lookup a j) then j:=k else skip fi;
    k:=k+1
  od
]|
```

The corresponding proof obligations are generated for this loop as well but they can be easily proved in separate subderivations.

7.6 Refining the Procedure Swap_in_array

The `Swap_in_array` procedure can be easily implemented using the `update` operation for arrays. However, the `update` operation allows changing only one array element in a time. Therefore, in order to make a swap of two elements, we should save the value of one of them in a temporary variable. We introduce a local variable `t` for this purpose and initialise it to the value of array element indexed by the value parameter `i1`.

```
|[var t:num.
  t:= lookup a i1;
  {(i1 < asize a) ∧ (i2 < asize a)};
  a:=a′.(swap a i1 i2 a′);
]|
```

Next we introduce the specification variable `A` to refer to the initial value of the array `a` as an assertion statement and then propagate the assertion inside the block collecting context information. After rewriting the body of the nondeterministic assignment (specification) statement using the context information, the focus is transformed to:

```
|[var t:num.
  t:= lookup a i1;
  {(a=A) ∧ (i1 < asize A) ∧ (i2 < asize A) ∧ (t = lookup a i1)};
  a := a'.(swap A i1 i2 a');
]|
```

Now we can add the assignment `a := update a i(lookup a j)` before the specification statement since `a` is no longer used in the latter. Propagating the context information once again results in the following focus:

```
|[var t:num.
  t:= lookup a i1;
  a := update a i1 (lookup a i2);
  {(i1 < asize A) ∧ (i2 < asize A) ∧
    (t = lookup a i1) ∧ (a = update A (lookup A j))};
  a := a'.(swap A i1 i2 a');
]|
```

Focusing on the body of the specification statement, we get the context information added to the assumption list. The body expression can be transformed preserving the backward implication relation. This is based on the following antimonotonicity property of a specification statement

$$(\forall s\, s'.\ R.s.s' \ \Rightarrow\ Q.s.s') \ \Rightarrow\ [Q] \ \sqsubseteq\ [R]$$

which implemented in the Refinement Calculator as a window rule (see Section 3.4).

Using the theorem

```
⊢ ∀a a' a'' i j t.
  (i < asize a) ∧ (j < asize a) ∧
  (t = lookup a i) ∧ (a' = update a i (lookup a j)) ⇒
       ((a'' = update a' j t) ⇒ swap a i j a'')
```

we can transform the body of the specification statement to `a' = update a i2 t`. Such a specification statement can be rewritten as an ordinary assignment statement. Rewriting and closing windows gives us the refinement of the procedure `Swap_in_array`:

```
procedure Swap_in_array (val i1,i2:num var a:(num)array)
  |[var t:num.
      t:= lookup a i1;
      a := update a i1 (lookup a i2);
      a := update a i2 t;
  ]|
```

7.7 The Final Program

As a result of the derivation, we arrived at the executable program that is proved to be the implementation (refinement) of our initial specification.

```
  procedure Min_in_subarray (val a:(num)array; i:num var j:num)
    |[var k:num.
       j:= i; k:=i+1;
       do (k < asize a) →
         if (lookup a k ≤ lookup a j) then j:=k
         else skip fi;
         k:=k+1
       od
    ]|


  procedure Swap_in_array (val i1,i2:num var a:(num)array)
    |[var t:num;
      t := lookup a i1;
      a := update a i1 (lookup a i2);
      a := update a i2 t
    ]|


var a:(num)array.
|[var i,k:num.
    i := 0;
    do i < ((asize a)−1) →
      Min_in_subarray(a,i,k);
      Swap_in_array(i,k,a);
      i := i+1
    od
]|
```

7.8 Conclusions

In this chapter we have illustrated how our mechanisation of procedures can be quite effectively used as a structuring tool for top-down development of a simple program. The derivation of the final implementation was straightforward and went quite smoothly. The main bulk of the work was actually done proving a number of properties of the introduced HOL constants. This was needed to discharge the proof obligations generated by the transformation rule for loop introduction.

Of course, we avoided some complications, for example while introducing procedure calls. In our example there was the direct correspondence between

specification statements to be replaced and procedure calls. Otherwise, we would need to prove additional proof obligations of the form

$$?- \texttt{S ref call P}(...)$$

To do this, we would have to unfold the right hand side of the refinement assertion, remove (if necessary) a block statement etc.

In the following chapters we return to the general properties of our procedure model. Some interesting questions are still unanswered. How can we use context information when working with procedures and procedure calls? How can we deal with recursive procedures? Is it possible to assign a procedure as a value to a variable? Let us find out.

Chapter 8

Procedure Calls in Context

In Chapter 5 we described how context information can be transported from one program place to another and used for program refinement. We presented two sets of rules – one set includes rules for propagating context assertions or assumptions through different constructs of our language, and the other one includes rules for doing refinement in the context provided by context assertions.

In this chapter we show how we can extend these sets with rules for procedure calls. Since a procedure call can be rewritten (unfolded) into the adapted body of the corresponding procedure, the expansion can be done in a rather simple way reusing the rules for working with context that were already presented.

8.1 Taking Context Into Account

Recall that refinement in context is refinement of the form $\{p\}; S \sqsubseteq S'$ where p is a predicate containing context information. Intuitively it means that we prove refinement of the statement S only for those initial states that satisfy the predicate p. Since we use additional (context) information in the refinement step, we get as a result a wider class of possible refinements of S.

Suppose we have a procedure call $P(e, x)$ where P is the name of a procedure, e are the value expressions, and x are the reference parameters. A context assertion before a procedure call provides us with additional information about the initial state in which the procedure call is executed. We can use this information for rewriting the actual value parameters of a procedure. The following HOL theorem describes the way in which value

expressions can be rewritten:

```
⊢ ∀p V V′ c f g.
   (∀u. p u ⇒ (V u = V′ u)) ⇒
   {p};(call c f g V) = {p};(call c f g V′)
```

Recall that V is a state function which returns the value expressions of a procedure call as a tuple. The theorem then says that we can use the context assertion p for rewriting the state function V into the new state function V′.

The above theorem forms the basis for the corresponding window rule that we have added to the window inference system. This window rule allows us to focus on any value expression of a procedure call and use the predicate of a context assertion as an additional assumption while rewriting it (doing a subderivation which preserves the equality relation).

Let us consider a very simple example. Suppose we are working with the program fragment $\{\mathtt{x} = 5\}$; $\mathtt{P}(\mathtt{x} + 1, \mathtt{y})$. The exact definition of the procedure P has no importance for this example. Focusing on the value expression $\mathtt{x} + 1$, we can start a new subderivation with the assumption $\mathtt{x} = 5$ added in the Assumptions subwindow. The relation to be preserved in this subderivation is the equality relation. Selecting the Rewrite Once transformation from the Transform menu, we can rewrite the focus (using the assumption $\mathtt{x} = 5$) to $5 + 1$ which can be further simplified (rewritten) to just 6. Closing the window (the subderivation), we get our initial focus transformed (according to the theorem presented above) to $\{\mathtt{x} = 5\}$; $\mathtt{P}(6, \mathtt{y})$.

We showed in this section how context information can be used for rewriting a procedure call. The only thing we can do (retaining a procedure call in its place) is to rewrite the value expressions of a procedure call. However, there is always the alternative possibility to unfold a procedure call into a block with the adapted procedure body. Then we can use the previously defined rules for propagating context information inside the block, and then use this information to refine a program statement we interested in.

8.2 Propagating Context Information Through a Procedure Call

As explained earlier, a procedure call can be rewritten into a block containing the adapted procedure body. Therefore, propagation of context information through a procedure call actually means pushing a context assertion through the corresponding program fragment.

In chapter 5 we presented a number of sharp context propagation rules for different constructs of our language. However, we cannot give a similar rule for procedure calls. The problem is that in general the procedure body can contain loop(s), and, as we have explained in Chapter 5, there is no simple way to propagate the maximal amount of context information through a loop.

Here we consider alternative ways to propagate context information through a procedure call. First of all, we distinguish the special case when the predicate of a context assertion before a procedure call is indifferent to (i.e., not affected by) execution of a procedure call. In other words, it is described in terms of variables that are not present in the list of reference variables of the procedure call. Then, according to the following theorem, the context assertion can be propagated through the procedure call:

```
⊢ ∀c f g V p.
   conjunctive c ⇒
   inverse f g ⇒
   (∃p'. ∀w u. (p o SND o g)(w,u) = p' w) ⇒
      ({p};(call c f g V) ref (call c f g V);{p})
```

The internal state (w, u) is the state that the lifted procedure body ($skip \| c$) is executed on. It consists of the global variables that are not affected by procedure execution (w) and the actual procedure state containing the value and reference parameters (u). The function SND o g translates the internal procedure state into the initial (global) state. The theorem then states that if the predicate p defined on the global state can be expressed (redefined) in terms of the variables that are not affected by procedure execution (w) then it can be propagated through a procedure call.

The theorem above can also be expressed in the form of a syntactic inference rule:

$$\frac{}{\{p\};\ P(\textbf{val}\ x;\ \textbf{var}\ y) \quad \sqsubseteq \quad P(\textbf{val}\ x;\ \textbf{var}\ y);\ \{p\}}$$

- P is conjunctive, and

  no free variables in p appear in the reference variable list y

Since all statements of our programming language are conjunctive, we can always discharge the first side condition of this inference rule.

We have written a transformation rule which allows us to propagate context information in the cases covered by the theorem. The condition

```
(∃p'. ∀w u. (p o SND o g)(w,u) = p' w)
```

is discharged by automatically finding (if possible) a "witness" predicate $\mathtt{p}'$ defined on the part of a program state that is not affected by the procedure execution. Such a "witness" always exists if the predicate `p` does not mention the variables that are included in the reference parameter list. The other conditions are discharged automatically as well.

Let us consider a simple example. In Chapter 6 we have defined the procedure `Square` for calculating the square of a natural number:

```
procedure Square (val x:num var y:num)
  y := x*x
```

Suppose we have the following fragment of the main program working on a program state consisting of three variables (`x`,`y` and `z`) of the type `num` in the focus:

```
{x≤y}; Square(y+1,z)
```

According to the theorem above we can propagate the assertion through the procedure call. After the transformation we get:

```
Square(y+1,z); {x≤y}
```

Before the transformation, the predicate of the context assertion is $(\lambda(\mathtt{x},\mathtt{y},\mathtt{z})\bullet\ \mathtt{x} \geq \mathtt{y})$. The transformation rule automatically generates the "similar" predicate $(\lambda(\mathtt{x},\mathtt{y})\bullet\ \mathtt{x} \geq \mathtt{y})$ which is used to prove and discharge the corresponding proof obligation.

8.3 An Alternative Solution

Another way of propagating context information through a procedure call is by using a previously stored initial specification of the procedure body. The idea (originally proposed by M.Staples in [99]) is to keep the procedure body of the form $\{S_0 \sqsubseteq S\}; S$ where S_0 is the initial specification of the procedure body, and S is the current implementation of it. This format can easily be introduced for the initial specification S_0 since the refinement $S_0 \sqsubseteq \{S_0 \sqsubseteq S_0\}; S_0$ is trivially true. Every further refinement of the procedure body preserves this format since the following rule is valid:

$$\frac{S \sqsubseteq S'}{\{S_0 \sqsubseteq S\};\ S \sqsubseteq \{S_0 \sqsubseteq S'\};\ S'}$$

A context assertion that is propagated through the initial specification can be propagated through any of its implementation as well:

$$\frac{\{p\};\ S_0 \ \sqsubseteq\ S_0;\ \{q\}}{\{p\};\ \{S_0 \ \sqsubseteq\ S\};\ S \ \sqsubseteq\ \{S_0 \ \sqsubseteq\ S\};\ S;\ \{q\}}$$

Both inference rules can be easily proved as corresponding HOL theorems.

We can propagate context information through the initial specification of the procedure body in a sharp way using the sharp context propagation rules for assertion and nondeterministic assignment statements given in Chapter 5. However, the application of this rule for any proper refinement of the initial specification would not yield the maximal amount of context information since implementation decisions made by the developer in the refinement steps are not reflected in it.

8.4 Discussion

When we consider how to use or propagate context information when working with procedures, we face a kind of conflict of interests. Talking about procedures, we usually mean "abstraction" and "hiding of details". On the other hand, talking about using or calculating context information, we actually mean "complete disclosure". The rules for working with context proposed in this chapter reflect this "all or nothing" dilemma.

If we see a procedure as a "black box", then context information accumulated before a procedure call can be used only for rewriting the value expressions of the procedure call, and can be propagated through a procedure call only if it is indifferent to the procedure execution. If we want to collect the maximal amount of context information, we should disclose what is hiding behind a procedure call. In other words, we should unfold the procedure call.

The idea proposed by M.Staples to store the initial specification of a procedure in an assertion statement can be seen as a compromise solution (it is similar to so called the "grey box" approach[27] advocated for development of component systems). This allows calculation of the context information after a procedure call in an easy way no matter how difficult the actual procedure implementation is. However, the question of how easy is this approach to implement in practice still remains open.

Chapter 9

Recursive Procedures

Our approach for modelling procedures allows us to define a procedure which contains recursive calls to itself. The body of such a procedure is then of the form $(\mu X. ...)$ where μ is the least fixpoint operator. In this chapter we consider recursive procedures in much more detail concentrating on such issues as introduction and unfolding of a recursive procedure call, introduction of a recursive procedure from a nonrecursive specification, correctness of a recursive procedure call etc.

We have already encountered fixpoints when talking about lattice-theoretical properties of predicate transformers and other mathematical structures used in the refinement calculus theory (Chapter 4). In this chapter we show how these general lattice properties of fixpoints can be applied for special cases and used in practice.

9.1 Representation of a Recursive Procedure

In the refinement calculus theory recursion is modelled as the least fixpoint of the corresponding function defined on predicate transformers. According to the theorem of Knaster-Tarski[100], a monotonic function on a complete lattice has the unique least fixpoint. Predicate transformers form a complete lattice and, therefore, the least fixpoint of monotonic functions on predicate transformers always exist and is unique. Recall that all statements of our language are monotonic and all statement constructors are regular (i.e., monotonic as functions on monotonic predicate transformers). The regularity and monotonicity assumptions should be explicitly stated in the theorems but they can be checked and discharged automatically.

When we enter a procedure containing recursive calls to itself, the parser

recognizes the recursive structure of the procedure body and generates the least fixpoint operator μ around its internal representation.

For example, a program with a recursive procedure `fact`

```
procedure fact (val n:num var x:num)
  if n>1 then fact(n−1,x); x:=x*n else x:=1 fi
in
<main program>
```

is internally represented as

```
PLET
  (λfact. <main program>)
  (VLET (λn.
    VLET (λx.
     mu (λfact. cond (λvar. n var > 1)
                     (call fact ... seq assign (λvar. n var, x var * n var))
                     (assign (λvar. n var, 1))))
         SND) FST)
```

We are allowed to focus and refine the body of a recursive procedure because of the following general monotonicity property of the least fixpoint operator:

```
mu_submono ⊢ (∀x. (f x) ref (g x))  ⇒  (mu f) ref (mu g)
```

Focusing on the body of a recursive procedure, we in fact focus inside the expression `mu`$(\lambda$`X`....), so any refinement that we prove is true for an arbitrary value of `X`, i.e., is of the form ∀`x`. (`f x`) `ref` (`g x`).

9.2 An Unfolding of a Recursive Procedure Call

An unfolding of a procedure call is the same as replacing the procedure call with a block containing the adapted procedure body. The block introduces new local variables for every procedure value parameter and initialises them with the value expressions from the procedure call.

Since the body of a recursive procedure is of the form (μ`X`....), we can go a step further in the unfolding process, using the basic unfolding property of the least fixpoint construct μ:

```
mu_unfold ⊢ ∀f. regular f ⇒ (mu f = f (mu f))
```

Let us consider a simple example. Suppose we have the procedure call `fact`$(5, z)$ in our focus. In this case, unfolding the procedure call would yield

```
|[ var n:num | n=5.
   if (n>1) then fact(n−1,z); z:=z*n
   else z:=1 fi ]|
```

Propagating context information inside the block, we can simplify the conditional statement by removing one of its branches.

```
|[ var n:num | n=5.
   {n=5}; fact(n−1,z); z:=z*n ]|
```

Using the context information, we can rewrite the value expression of the procedure call and then propagate the context assertion through it (since the variable `n` is not affected by the procedure execution). As a result, we have:

```
|[ var n:num | n=5.
   fact(4,z); {n=5}; z:=z*n ]|
```

Rewriting the right hand side of the assignment statement using the context assertion {`n = 5`} and removing the context assertion afterwards give us the block body where the local variable `n` is not used anymore. Applying the `Block elimination` transformation rule from the menu then yields the following focus:

```
fact(4,z); z:=z*5
```

The theorem that we have proved in this derivation is

```
  fact = (mu fact. if n>1 then ...)
⊢ fact(5,z) = fact(4,z); z:=z*5
```

This corresponds exactly to one step of intuitive execution of the recursive procedure `fact`.

9.3 An Introduction of a Recursive Procedure Call

If we want to introduce a recursive procedure call in the place of some program fragment `c`, we actually have to prove that

```
⊢ c ref (call P ...)
```

where `P` is the body of a recursive procedure.

If `P` has no value parameters, then the unfolded procedure call can be simplified by eliminating the block statement that is introduced in the unfolding step. The resulting goal is then

```
⊢ c ref (mu f)
```

where (mu f) is now defined on the global program state.

This goal can be handled using the theorem mu_thm from the HOL theory of the refinement calculus:

```
val mu_thm =
  ⊢ ∀c.
      regular f ∧
      monotonic c ∧
      (∃t.∀i. {λv. t v = i}; c  ref
              f ({λu. t u < i}; c))
      ⇒
      c ref mu f
```

Since the regularity and monotonicity assumptions can be automatically discharged, mu_thm allows us to reduce an introduction of a recursive procedure call to a proof obligation of the form

```
∀i. ({λu. t u = i}; c)   ref
    f ({λu. t u < i}; c)
```

where t is a termination function (variant) supplied by the user.

If a recursive procedure has value parameters, we cannot directly use mu_thm to simplify the goal. We need some preparatory steps which transform the program fragment that we are replacing in such a way that, firstly, it becomes independent of the value expressions of a procedure call, and, secondly, it can be executed on the procedure state.

Let us explain this with a small example. Suppose we want to introduce the call fact(x + 1, y) in the place of the specification statement y := FACT (x + 1) where FACT is the HOL constant for factorial. Unfolding the procedure call, we get the following goal (proof obligation) to prove:

```
y :=  FACT(x+1)
ref
|[ var n | n=x+1.
     adapt f g (mu fact.
       if n>1 then fact(n−1,y); y:=y*n
       else y:=1 fi)
]|
```

The assignment y := FACT (x + 1) is defined on the program state (x, y), i.e., it is internally represented as assign(λ(x, y). x, FACT(x + 1)). The functions f and g are state reorderings, defined in the following way: f = (λ(n, x, y). x, n, y) and g = (λ(x, n, y). n, x, y).

To prove this goal, we start a subderivation starting with the statement $\mathtt{y} := \mathtt{FACT}(\mathtt{x}+1)$. In the first step, we introduce a block with the local variable $\mathtt{n}$ initialised in the same way as in the block of the unfolded procedure call.

```
|[ val n | n=x+1. y := FACT(x+1) ]|
```

Note that the actual representation of the assignment statement inside the block now is $\mathtt{assign}(\lambda(\mathtt{n},\mathtt{x},\mathtt{y}).\,\mathtt{n},\mathtt{x},\mathtt{FACT}(\mathtt{x}+1))$.

Propagating context information about the initial value of variable $\mathtt{n}$ inside the block and using it for refining the assignment statement gives us the following focus:

```
|[ val n | n=x+1. {n=x+1}; y := FACT n ]|
```

It is easy to notice that $\mathtt{assign}(\lambda(\mathtt{n},\mathtt{x},\mathtt{y}).\,\mathtt{n},\mathtt{x},\mathtt{FACT}\,\mathtt{n})$ (the assignment inside the block) is equal to $\mathtt{skip} \parallel \mathtt{assign}(\lambda(\mathtt{n},\mathtt{y}).\,\mathtt{n},\mathtt{FACT}\,\mathtt{n})$ on the rearranged state $(\mathtt{x},\mathtt{n},\mathtt{y})$. It can be expressed by the adaption operation using the state reordering functions $\mathtt{f}$ and $\mathtt{g}$ defined above:

```
assign(λ(n,x,y).n,x,FACT n) = adapt f g (assign(λ(n,y).n,FACT n))
```

Using this equality to rewrite the body of the block gives us the following focus:

```
|[ val n | n=x+1. adapt f g (y := FACT n) ]|
```

As a result, the following refinement is proved in the subderivation:

```
⊢ y := FACT(x+1)
  ref
  |[ val n | n=x+1. adapt f g (y := FACT n) ]|
```

Using this result and the transitivity property of the program refinement relation, we can reduce our main goal to:

```
|[ val n | n=x+1. adapt f g (y := FACT n) ]| ref
|[ val n | n=x+1. adapt f g (mu fact.
     if n>1 then fact(n−1,y); y:=y*n
     else y:=1 fi)
]|
```

Since the block statement and the adaption operator are both monotonic with respect to their statement arguments, the goal can be further reduced to:

```
y := FACT n
ref
(mu fact.
   if n>1 then fact(n−1,y); y:=y*n
   else y:=1 fi)
```

The preparation stage is over. The goal is now of the form that allows application of the theorem `mu_thm`. In order to further simplify the goal, we supply the concrete termination function `n`. This yields the following goal:

```
∀i. {n = i}; y := FACT n
     ref
     if n>1 then
        call ({n < i}; y := FACT n) (n−1,y); y:=y*n
     else y:=1 fi
```

Finally, by expanding definitions we can easily prove this goal. Note that here we used a slightly different syntax for a procedure call to stress the fact that the procedure body `fact` has been replaced with $\{\mathtt{n} < \mathtt{i}\}$; `y` := `FACT n`.

Of course, in the general case it may be not so easy to transform a program fragment into the form required for application of `mu_thm`. Some further automation is definitely needed to make an introduction of a recursive procedure easier.

9.4 Correctness of a Recursive Procedure Call

Sometimes we need to check that a recursive procedure call is correct with the respect to certain pre- and post-conditions. In order to reduce such a correctness assertion, we combine the following two theorems:

```
correct_call =
     ⊢ ∀f g c p q V.
       inverse f g ⇒
       (correct p (call c f g V) q  =
        (∀w. correct (λu. (p o SND o g) (w,u) ∧
                          ((FST o g) (w,u) = (V o SND o g) (w,u)))
                     c (λu. (q o SND o g) (w,u))

correct_mu =
     ⊢ ∀f p p′ q t.
       regular f ∧ (p implies p′) ∧
       (∀w.
         correct (λs. p′ s ∧ (t s = w))
           (f ({λs. p′ s ∧ t s < w}; nondass (λs s′. q s′))) q)
       ⇒
       correct p (mu f) q
```

The first theorem (introduced in Section 6.3.4) reduces a correction assertion of a procedure call to a correctness assertion of the procedure body. Since the body of a recursive procedure is of the form μf, the correctness assertion can be further reduced using theorem `correct_mu`, getting rid of the μ construct. The variables p′ and `t` in the theorem `correct_mu` stand for an invariant property and a termination function of the recursion encoded by μf.

Let us look into a simple example. Suppose we have defined the following recursive procedure:

```
procedure Exp (val x,y:num var z:num)
   if y=0 then skip
   else z,y := z*x,y−1; Exp(x,y,z)
   fi
```

This recursive procedure can be used to calculate the power x^y and store it into the variable `z` provided that the value of `z` before a procedure call is equal to 1. We can prove it by checking that the following correctness assertion is true:

```
correct (x=x0 ∧ y=y0 ∧ z=1)
         Exp(x,y,z)
         (z = x0^y0)
```

The specification variables `x0` and `y0` here are used to indicate the initial values of the variables `x` and `y`.

Using the theorem `correct_call` gives us (after some simplifications) the following correctness assertion on the procedure body:

```
correct (x=x0 ∧ y=y0 ∧ z=1)
         (mu X. if y=0 then skip
                else z,y := z*x,y−1; Exp(x,y,z)
                fi)
         (z = x0^y0)
```

Using the invariant `x = x0 ∧ z*x^y = x0^y0` and the termination function `y` allows us to reduce this correctness assertion (according to the theorem `correct_mu`) to the following goal:

```
regular (λX. if y=0 then ...) ∧
(∀x y z. (x=x0 ∧ y=y0 ∧ z=1) ⇒ (x=x0 ∧ z*x^y = x0^y0)) ∧
(∀w. correct
   (x=x0 ∧ y=y0 ∧ z=1 ∧ y=w)
   (if y=0 then skip
      else z,y := z*x,y−1;
           {x=x0 ∧ z*x^y = x0^y0 ∧ y<w}; z:=z′.z′=x0^y0
      fi)
   (z = x0^y0))
```

The first two conjuncts of the goal can easily be proved and discharged. Expanding the definitions of correctness and program statements, we can prove the third subgoal as well.

The final result is the theorem:

```
  Exp = mu (λExp. if y=0 then skip else ...)
⊢ correct (x=x0 ∧ y=y0 ∧ z=1)
          Exp(x,y,z)
          (z = x0^y0)
```

Note that the definition of the procedure `Exp` is presented as the assumption of this theorem.

9.5 An Introduction of a Recursive Procedure

We can introduce recursive procedures as well as ordinary procedures, entering them together with the main program in the beginning of our session or using the `Introduce New Procedure` transformation rule during our session. In both cases the parser recognizes the recursive structure of the procedure body and generates the least fixpoint operator μ around its internal representation.

In some cases, however, the procedure body is given as a nonrecursive specification which can still be implemented in a recursive way. This can be done using the recursion introduction rule `Rec_intro` included into the Refinement extension of the Refinement Calculator. This rule implements the following inference rule:

$$\frac{\{t = i\}; c \ \sqsubseteq \ f\,(\{t < i\}; c)}{c \ \sqsubseteq \ \mu f} \qquad \texttt{Rec\_intro}$$

where t is the variant function and i is a new specification variable. The rule is based on the theorem `mu_thm` shown above. This rule is also a specialisation of the lattice-theoretical property for μ introduction presented as the following inference rule in Section 4.2.7:

$$\frac{\forall i.\ C_i \ \sqsubseteq \ f(\sqcup j < i.\, C_j)}{(\sqcup i \in Nat.\, C_i) \ \sqsubseteq \ \mu f} \qquad (\mu \ \textit{introduction})$$

To obtain the rule `Rec_intro`, we should specialise C_i with $\{t = i\}; c$, the general join operator $\sqcup$ with the angelic choice $\sqcup$ on predicate transformers, and the lattice ordering $\sqsubseteq$ with the program refinement relation $\sqsubseteq$.

Let us explain how the rule `Rec_intro` works. If we want to introduce recursion in place of the current focus `c`, we are asked to supply a variant

(state function) `t` and then a subderivation starts. In the subderivation we have to transform the initial focus $\{\texttt{t} = \texttt{i}\};\texttt{c}$ to a program fragment containing $\{\texttt{t} < \texttt{i}\};\texttt{c}$. After we finish the subderivation, the transformation rule `rec_intro` extracts a function f and replaces the initial focus with $\mu\texttt{f}$.

Let us consider the factorial example. Suppose we have in the focus

```
y := FACT x
```

where `x` and `y` are variables of the type `num`. Selecting the `Rec_intro` transformation rule from the `Refinement` menu and entering `x` for the variant function yields the following focus:

```
{ x=i };
y := FACT x
```

After a number of transformations we can arrive at the following implementation:

```
if x=0 then y := 1
else
  x := x−1;
  { x<i }; y := FACT x;
  y := y*(x+1)
fi
```

Closing the window (subderivation) gives us the focus with the recursion operator μ introduced:

```
(mu X. if x=0 then y := 1
       else
         x := x−1; X; y := y*(x+1)
       fi )
```

However, we cannot directly use `Rec_intro` for transforming a nonrecursive specification into the body of a recursive procedure, since the latter should be of the form (`muX. ... call X ...`). In order to do this, we modify the transformation rule `Rec_intro` by adding one intermediate step for a procedure call introduction. The new transformation rule is called `Recpro_intro`.

Let us look once again at the factorial example. Suppose we have introduced the procedure `fact` as follows:

```
procedure fact (val x:num var y:num)
   y := FACT x
```

Focusing on the procedure body, we start recursive procedure introduction subderivation with the focus:

```
{ x=i };
y := FACT x
```

After a number of refinements we can obtain the following (slightly different) implementation:

```
if x=0 then y := 1
else
  { (x−1)<i }; y := FACT (x−1);
  y := y*x
fi
```

In order to reuse the `Rec_intro` rule to produce the format required for recursive procedures, we need to introduce a recursive procedure call into the program. In this particular case, we need to replace

$$\{(\mathtt{x}-1)<\mathtt{i}\};\ \mathtt{y} := \mathtt{FACT}(\mathtt{x}-1)$$

with a procedure call of the form

$$\mathtt{call}(\{\mathtt{x}<\mathtt{i}\};\ \mathtt{y} := \mathtt{FACT}\,\mathtt{x})\ (\_,\_)$$

The selection of concrete parameters should be extracted from the program. In this case, it is easy to see that the refinement needed to prove is

```
{ (x−1)<i }; y := FACT (x−1)
ref
call ({ x<i }; y:=FACT x) (x−1,y)
```

which is obviously true (which can be proved by unfolding the procedure call in the right hand side of refinement).

Using this theorem, the focus is transformed to:

```
if x=0 then y := 1
else
  call ({ x<i }; y:=FACT x) (x−1,y);
  y := y*x
fi
```

Closing the subderivation (applying `Rec_intro`) results in the following focus:

```
(mu X. if x=0 then y := 1
       else
         call X (x−1,y); y := y*x
       fi )
```

The modified transformation rule `Recpro_intro` differs from `Rec_intro` by the additional step introducing the `call` operator into a program. In general, this step can be made as an additional proof obligation. After this is done, the `Rec_intro` transformation rule is reused to do the rest.

9.6 Conclusions

In this chapter we showed how our approach for modelling procedures can be extended to deal with procedures containing recursive self-calls. Recursive procedures are modelled as the least fixpoints of corresponding functions on predicate transformers. We use lattice-theoretical properties of the least fixpoint operator as the basis for proving properties about recursive procedures and for writing transformation rules for working with recursive procedures. Our implementation is by no means complete; automatic matching and simplification is especially needed, for example, in the introduction of recursive procedures and their calls.

In our approach we permit only simple recursion of procedure calls. This means that no mutual or cyclic dependencies among procedures are allowed. In Chapter 13 we briefly present our theoretical study of component systems where this restriction is dropped. P.Homeier[57, 58] has done extensive work on mechanical verification of programs with mutually recursive procedures. His approach uses annotated procedure calls containing additional variant information and it automatically generates verification conditions needed to prove termination and other correctness properties of programs.

Chapter 10

Procedure Variables

10.1 Introduction

One of the advantages that higher-order logic offers us is the possibility to reason about functions of arbitrary complexity. Such higher-order objects can be used anywhere alongside with objects of "basic" types such as integers, booleans and so on. To illustrate this point, in this section we consider the use of program variables of a procedure (predicate transformer) type.

Recall that the program state that we are working on is a tuple of polymorphic type which for concrete programs can be instantiated in different ways. Program variables are then projection functions on this tuple. An application of these functions to the concrete program state gives us the value of a program variable in this state. Since the program state is of a polymorphic type, any well-defined type of higher-order logic can be used as the base type of a program variable.

We model procedures as predicate transformers working on a program state that consists of value and reference parameters. By defining a program variable of the base type $Ptrans(\Sigma \times \Gamma)$, we make it possible to store any procedure with value parameters of type Σ and reference parameters of type Γ in this variable. Variables of procedure type can be used in the same way as variables of an ordinary type – they can be assigned to, they can be used in expressions (i.e., they can be called), they can be supplied as parameters to other procedures etc.

In the first part of this chapter we discuss the most common uses of procedure variables. In the second part we consider the problem of refining the value of a variable of a procedure type.

10.2 Using Procedure Variables

In this section we consider the most typical cases of using procedure variables. Suppose that we are working with a program containing a procedure `Exp`. This procedure takes two integers `x` and `y` as the value parameters and returns the value x^y in the reference parameter `z`. The main program starts with the program variable declarations which include the declaration of a procedure variable `pvar`. The base type of this procedure variable is `Ptrans((num#num)#num)` which means that any procedure with two value parameters of integer type and one reference parameter of integer type (like `Exp`) can be assigned to this variable.

```
procedure Exp(val x,y:num; var z:num)
   <procedure Exp body>
var n,k: num;
    a1,a2,b: (num)array;
    pvar: Ptrans((num#num)#num);
...
```

Then we can assign a value to variable `pvar` in the usual way (in the assignment statement), for example,

```
pvar,n := Exp,10;
```

Once a procedure variable has been assigned some concrete value, we can use it in two ways – by calling its value (the procedure) or by passing it as a parameter to other procedures. In the first case a procedure variable should be used as the first parameter of the `call` operator. However, here we face a type clash since the `call` operator expects a predicate transformer rather than a program variable (a state projection function).

The solution is to define a special procedure call statement for this particular case. It can be done in a rather straightforward way, reusing the definition of `call`:

$\vdash_{def}$ `(vcall v f g V) q s = (call (v s) f g V) q s`

For any postcondition `q` and initial state `s` the operator `vcall` is defined as the `call` operator where the calling procedure body is taken as the value of the procedural variable `v` in the initial state `s`.

In our example we can call `pvar` in the following way:

```
pvar(n−1,2,k);
```

which is internally represented as

```
vcall pvar <reordering f> <reordering g> (λs. n−1,2)
```

Context information about the value of a procedure variable can be used to rewrite a procedure call. This is expressed in the following theorem:

```
val vcall_in_context =
⊢ {λs. v s = proc}; vcall v f g V  =
  {λs. v s = proc}; call proc f g V
```

The meaning of this theorem is quite obvious – if we know that a procedure variable contains the value of a concrete procedure `proc`, the call of such a procedure variable is equivalent to the call of the procedure `proc`.

For example, suppose the call $\texttt{pvar}(\texttt{n}-\texttt{1},\texttt{2},\texttt{k})$ occurs in the context $(\texttt{pvar} = \texttt{Exp}) \wedge (\texttt{n} = \texttt{10})$. Then it is equivalent to the ordinary procedure call $\texttt{Exp}(\texttt{9},\texttt{2},\texttt{k})$. As a result of this procedure call, the variable `k` gets the value `81`.

A procedure variable can be used as a parameter of other procedures in the usual way. For example, suppose we introduce a new procedure

```
procedure Zip_Arrays (val op: Ptrans((num#num)#num); a,b:(num)array
                      var c: (num)array)
 <procedure Zip_Arrays body>
```

This procedure takes two arrays of integers (of the same length) and "zip" them together using the supplied operation `op` ("zipping" means that we apply the operation `op` for the subsequent array elements `a`[`i`] and `b`[`i`]). The result is returned in the reference parameter `c`.

Then we can call the procedure `Zip_arrays` supplying the value of the procedure variable `pvar` as the actual operation:

```
Zip_Arrays(pvar,a1,a2,b);
```

Of course, using procedure variables in procedure calls makes sense only if these variables have earlier been assigned a value. A call to an unassigned procedure variable is, of course, allowed in our formalisation, but it is impossible to prove anything about such a procedure call.

10.3 Refinement of the Value of a Procedure Variable

The approach presented in the previous section raises some interesting questions that we cannot leave unanswered. What happens if we refine a procedure that was assigned to a procedure variable? Is the refinement of the whole program preserved in this case?

Operationally it seems all right. In all places where an old procedure (as the value of a procedure variable) was executed, execution of the new refined procedure would take place.

However, being the value of a procedure variable, the old procedure was a part of the global state. By refining the procedure, we are at the same time changing the global state. That means that the ordinary (algorithmic) refinement relation is not sufficient in this case since it requires that the relationship between programs would be proved for the same initial state:

$$S \sqsubseteq S' \equiv \forall q\, \sigma \bullet S.q.\sigma \Rightarrow S'.q.\sigma$$

where q is a postcondition and σ is an initial state.

10.3.1 Solution: Data Refinement

To prove validity of refinement of the value of a procedure variable, we need a more general refinement relation that permits a change of the program state. That suggests the use of data refinement.

As explained in Chapter 2, data refinement between statements S and S' with an abstraction relation R, denoted $S \sqsubseteq_R S'$, can be expressed via the ordinary refinement in the following way:

$$S \sqsubseteq_R S' \equiv (abst\ R);\ S;\ (repr\ R) \sqsubseteq S'$$

where *abst* and *repr* are the abstraction and representation statements allowing to simulate execution of an abstract statement S on a concrete state (see Section 2.6).

In our case the relationship between an abstract state component a of a procedure type and its concrete counterpart c can be expressed via the refinement $a \sqsubseteq c$. Note that here we have to use different names to distinguish the abstract and concrete state components. In practice, the variable names before and after data refinement can be the same.

The data refinement extension of the Refinement Calculator allows us (in most cases) to calculate the concrete program from an abstract one and the supplied abstraction relation. The calculation is done in a structural way by decomposing the initial abstract program S, and then using preproved theorems of the form $(abst\ R);\ T;\ (repr\ R) \sqsubseteq T'$ for the basic cases. Therefore, to make it possible to data-refine a program containing procedure variables, we should provide data refinement theorems for the different cases where procedure variables are used.

We consider three different cases:

1. when a procedure variable is assigned a new value;
2. when we call a procedure as the value of a procedure variable;
3. when a procedure variable is passed as the value parameter to another procedure.

10.3.2 Proving Different Cases of Data Refinement

First, let us denote our abstraction relation as `dr` = (λ(a, c, u). `a ref c`) where `u` are the state components that are not affected by data refinement. For simplicity, we assume that the procedure variable is the first component of the program state.

Assigning the value to a procedure variable Now we are ready to formulate and prove the theorem for data refinement of an assignment statement which assigns the new procedure value to a procedure variable:

```
pvar_assign_dataref =
    ⊢ ∀p p'.
       p ref p' ⇒
       (abst dr); a := p; (repr dr)  ref  c := p'
```

This theorem states that, if we know from the context that the procedure `p` is refined by the procedure `p'`, then the assignment statement `a := p` can be data refined (with the abstraction relation `dr` = (λ(a, c, u). `a ref c`)) to the assignment statement `c := p'`.

As a special case (when `p` = `p'`) we have

```
pvar_assign_dataref2 =
    ⊢ ∀p.
       (abst dr); a := p; (repr dr)  ref  c := p
```

This means that, if we do not know anything the procedure `p`, data refinement leaves the assignment unchanged.

Let us return to the example presented in Section 10.2. Suppose we have refined the procedure `Exp` to `Exp'`. This fact is added as the additional assumption of the main program. Then the data refinement extension can use the theorem `pvar_assign_dataref` and the added assumption to transform the assignment statement

```
pvar,n := Exp,10
```

to

```
pvar,n := Exp′,10
```

The abstraction relation `dr` is

$$(\lambda(\texttt{pvar}, \texttt{pvar}', (\texttt{n}, \texttt{k}, ...)).\ \texttt{pvar ref pvar}')$$

in this case.

Calling the value of a procedure variable For the remaining two cases, we assume that a procedure variable was earlier assigned some concrete value (a concrete procedure). That means that we can obtain (by using the context propagation rules) a context assertion of the form {`pvar = P`} before a procedure call. Here `pvar` is the name of a procedure variable and `P` is the name (body) of some concrete procedure.

Using the theorem `vcall_in_context`, we can then rewrite the procedure call (`vcall pvar...`) into (`call P...`). It is easy to notice that now the procedure call statement is independent of the value of the procedure variable (since the value of a procedure variable cannot be used as the body of the calling procedure and one of its parameters at the same time due to a type clash[1]). The following theorem can then be used:

```
pvar_call_dataref =
  ⊢ ∀p p′.
      p ref p′ ⇒
      (∀q x y u. call p f g V q (x,u) = call p f g V q (y,u)) ⇒
       monotonic p ⇒
       (abst dr); call p f g V; (repr dr)
       ref
       call p′ f g V
```

The assumption (`∀q x y u. call p f g V q (x,u) = ...`) expresses the fact that the procedure call is independent of the first state component (a procedure variable). It can be reduced to the corresponding assumptions on the procedure parameters and then automatically checked and discharged.

After the data refinement is done, we can once again propagate the context assertion containing information about the new refined value of a procedure variable and, applying the theorem `vcall_in_context` in a symmetrical way, restore the `vcall pvar` operator.

[1]Recall that the type of a procedure is constructed from the types of its value and reference parameters. When the value or reference parameters of a procedure are missing, the HOL type `one` is used to indicate this. Therefore, the type of a procedure always differs from the type of any of its parameters.

For the procedure call pvar(n − 1, 2, k) from the example in Section 10.2, the data refinement extension can use the calculated context assertion {pvar = Exp} to transform the call pvar(n − 1, 2, k) to Exp(n − 1, 2, k), then use the theorem pvar_call_dataref and the assumption Exp ref Exp′ to data refine it to Exp′(n − 1, 2, k). After the data refinement of the whole program is done, the new context assertion {pvar = Exp′} can be calculated before the procedure call, and used for the restoration of pvar(n − 1, 2, k).

Passing a procedure variable as a value parameter Finally, let us consider the case when a procedure variable is passed to another procedure as a value parameter. Without losing generality, we show how our method works for the case when the procedure has only one value parameter, i.e., a procedure call is of the form:

```
call proc f g (λs. pvar s)
```

where pvar is the name of a procedure variable.

In a similar way as in the previous case, we can calculate the value of a procedure variable from the preceding program context. As a result, a context assertion of the form {pvar = P} is obtained before a procedure call. Using the context information, the procedure call can be then rewritten to

```
call proc f g (λs. P)
```

where P is the value of the variable pvar before the procedure call. Once again the resulting statement is independent of the pvar value. Then the following theorem can be used:

```
pvar_vcall_dataref =
  ⊢ ∀P P′.
      (P ref P′) ∧
      (∀q x y u. call proc f g (λs. P) q (x,u) =
                 call proc f g (λs. P) q (y,u))
      ⇒
        ((call f g proc (λs. P)) ref (call f g proc (λs. P′)) ⇒
         abst dr; call proc f g (λs. P); repr dr
         ref
         call proc f g (λs. P′))
```

According to the theorem, in the case when a procedure call is independent of the procedure variable, we can reduce the proof of data refinement between the corresponding procedure calls to a proof of ordinary refinement

between them. Unfolding procedure calls and using the monotonicity properties of the block statement and the `adapt` operator, we can further reduce it to the data refinement (with the same abstraction relation `dr`) between the statements `pvar` := `p`; `proc` and `pvar` := `p`′; `proc` where `proc` is the body of the called procedure. The following syntactic derivation describes the structure of the proof.

$$
\begin{array}{ll}
 & P1 \sqsubseteq P2 \\
\vdash & \mathit{call}\ \mathit{proc}\ f\ g\ (\lambda s.\ P1) \\
= & \{\text{unfold the procedure call}\} \\
 & |[\ \mathit{var}\ \mathit{pvar} |\ \mathit{pvar} = P1.\ \mathit{adapt}\ f\ g\ \mathit{proc}\ ]| \\
= & \{\text{move initialisation inside}\} \\
 & |[\ \mathit{var}\ \mathit{pvar} |\ T.\ \mathit{adapt}\ f\ g\ (\mathit{pvar} := P1;\ \mathit{proc})\ ]| \\
\sqsubseteq & \{\text{data refinement of the block body}\} \\
 & \quad \bullet\ \mathit{adapt}\ f\ g\ (\mathit{pvar} := P1;\ \mathit{proc}) \\
 & \quad \sqsubseteq_{\{\mathit{pvar} \sqsubseteq \mathit{pvar}'\}} \\
 & \quad\quad \bullet\ \mathit{pvar} := P1;\ \mathit{proc} \\
 & \quad\quad \sqsubseteq_{\{\mathit{pvar} \sqsubseteq \mathit{pvar}'\}} \\
 & \quad\quad\quad \mathit{pvar}' := P2;\ \mathit{proc}[\mathit{pvar}/\mathit{pvar}'] \\
 & \quad\quad \mathit{adapt}\ f\ g\ (\mathit{pvar}' := P2;\ \mathit{proc}[\mathit{pvar}/\mathit{pvar}']) \\
 & |[\ \mathit{var}\ \mathit{pvar}' |\ T.\ \mathit{adapt}\ f\ g\ (\mathit{pvar}' := P2;\ \mathit{proc}[\mathit{pvar}/\mathit{pvar}'])\ ]| \\
= & \\
 & |[\ \mathit{var}\ \mathit{pvar}' |\ \mathit{pvar}' = P2.\ \mathit{adapt}\ f\ g\ (\mathit{proc}[\mathit{pvar}/\mathit{pvar}'])\ ]| \\
= & \\
 & |[\ \mathit{var}\ \mathit{pvar} |\ \mathit{pvar} = P2.\ \mathit{adapt}\ f\ g\ \mathit{proc}\ ]| \\
= & \\
 & \mathit{call}\ \mathit{proc}\ f\ g\ (\lambda s.\ P2)
\end{array}
$$

Indentations (marked by $\bullet$) indicate the subderivations needed to justify certain steps in the derivation. The main derivation preserves the ordinary refinement relation, while the subderivations preserve data refinement with the abstraction relation $(\lambda(\mathtt{pvar}, \mathtt{pvar}', \mathtt{u}).\ \mathtt{pvar} \sqsubseteq \mathtt{pvar}')$.

The complete rule for the third case can be syntactically expressed as the following inference rule:

$$\frac{P1 \sqsubseteq P2 \;\vdash\; pvar := P1;\ proc \quad \sqsubseteq_{dr} \quad pvar' := P2;\ proc[pvar/pvar']}{P1 \sqsubseteq P2 \;\vdash\; call\ proc\ f\ g\ (\lambda s.\ P1) \quad \sqsubseteq \quad call\ proc\ f\ g\ (\lambda s.\ P2)}$$

As a result, we reduced the proof of data refinement of a procedure call to the corresponding data refinement of the procedure body which recursively can be solved using the rules given for the first two cases.

Returning to the example from the previous section once again, we can see that the proof of data refinement of the procedure call `Zip_Arrays(pvar, a1, a2, b)` in the context `(pvar = Exp)` $\wedge$ `(Exp ref Exp′)` can be reduced to the proof of the following goal:

```
Exp ref Exp′
?- (call Zip_Arrays f g (λ s. Exp)) ref
   (call Zip_Arrays f g (λ s. Exp′))
```

which in turn can be reduced to the proof of the goal

```
Exp ref Exp′
?- (abst R); pvar:=Exp; Zip_Arrays; (repr R)  ref
   pvar:=Exp′; Zip_Arrays
```

where `R` is the abstraction relation $(\lambda(\texttt{pvar}, \texttt{pvar}', ...).\ \texttt{pvar ref pvar}')$.

We started this section with the question "Is refinement of the whole program preserved in the case when we refine the value of a procedure variable?". Summarising, we answer affirmatively to this question though we should admit that a complete mechanisation of the approach described can be quite involved.

10.4 Discussion

Using procedure variables is a commonplace practice in functional programming. However, in imperative programming it is rare. Actually, there are only a few imperative programming languages that allow using procedure variables. For example, in standard Pascal[103] and Oberon[93] passing procedures as value parameters to other procedures is allowed. Modula-2[104] allows all three common uses of procedure variables that we described in this chapter. However, the problems that could be solved using procedure variables are usually solved using object-oriented techniques and dynamic binding.

We have found it challenging to test how our approach handles such higher-order objects as procedure variables. Due to the nature of the program state modelled, procedure variables could be used alongside with variables of "ordinary" types without any problems. However, things got more complicated when we considered what happens if we refine the value of a procedure variable. The solution proposed (using data refinement) provides a good basis for this problem to be handled mechanically.

Chapter 11

Specifying Procedures by Hoare Triples

11.1 Introduction

In this chapter we step aside from mechanisation issues that we have been discussing so far in the Thesis and do a theoretical investigation. The reason for this is the following. Studying various theoretical treatments of procedures, we have encountered a number of syntactic correctness rules for procedure calls [47, 75, 19, 20]. The procedures in these rules were specified as correctness triples (i.e., by pre- and postconditions). Attempts to compare approaches have led to some interesting and quite general theoretical results about the relationship between specifications in the form of correctness triples and specification statements of the refinement calculus. Though we concentrate more on theoretical issues rather than mechanisation in this chapter, all theoretical results are also presented as HOL theorems proved in the HOL theory of the refinement calculus. The material of this chapter is based essentially on a published paper[66] written together with J.von Wright.

The use of pre- and postconditions to describe desired program behavior is a well-known and widespread technique. A program (or program fragment) S can be specified as follows:

$$\{p\} \;\; S \;\; \{q\}$$

where p (the precondition) and q (the postcondition) are predicates over the state space (the program variables). Such a specification is called a *Hoare triple* or a *(total) correctness assertion*.

In this chapter we investigate the relationship between specifications in the form of Hoare triples on the one hand and specification statements in the refinement calculus [4, 84] on the other. We are focusing on the issue of *sharpness*, i.e., on finding the most abstract specification satisfying a given Hoare triple.

The situation is particularly complex when *specification variables* are used. These are variables that occur free in the pre- and postcondition of the specification, without being program variables. Without specification variables it would not be possible to refer to initial values of program variables in the postcondition.

We investigate how specification variables in Hoare triples should be interpreted in general, and what their exact role is in the communication of values between pre- and postconditions. Our main result (Thm.1) shows how a Hoare triple with specification variables can be translated into the refinement calculus specification statement without losing sharpness. In this translation we eliminate specification variables from a specification, and this elimination subsumes several rules proposed by others in the past [33, 55, 91].

In the second part of the chapter we show how our results can be used for analysis and derivation of proof rules for the correctness of procedure calls. Such rules allow us to deduce correctness properties for procedure calls from a characterising correctness property of the procedure body. A procedure can be specified as follows:

$$\begin{array}{l}\mathsf{procedure}\ P(\mathsf{value}\ x;\ \mathsf{value}-\mathsf{result}\ y)\\ \quad \{p\}\ \ B\ \ \{q\}\end{array}$$

where B is a dummy standing for the body of the procedure and any implementation must work on the parameters x and y, leaving the value parameter x unchanged. A correctness rule for procedures then allows us to verify correctness formulas of the form

$$\{p'\}\ P(a,b)\ \{q'\}$$

where a and b are the actual parameters (a can be an expression but b must be a program variable).

Many rules have been proposed for procedures [47, 75, 19, 20]. A rule is most useful if it is *sharp*, i.e. if it provides a way of calculating the *weakest precondition* for a procedure call with respect to any postcondition.

Our analysis of specifications and specification variables directly yields a sharp rule for procedures in the presence of specification variables. The rule itself is not new [20], but our formulation is more general, depending

only on the specified statement satisfying a very general semantic condition (conjunctivity).

11.2 Specification Statements of the Refinement Calculus

In this section we partly recall and partly introduce the basic notions needed in our investigation: conjunctivity and monotonicity of program statements, different kinds of specification statements, and specification variables. Lemma 11.1 is to our knowledge new; more details about the rest can be found in standard introductions to predicate transformer semantics and refinement [35, 4, 84, 15].

11.2.1 A Useful Lemma

The predicate transformers that model program statements are generally assumed to satisfy certain healthiness conditions. In this investigation, the only healthiness condition that is essential is conjunctivity. Recall that a predicate transformer is called conjunctive if it satisfies the following:

$$S.(\cap i \mid i \in I \bullet q_i) \quad = \quad (\cap i \mid i \in I \bullet S.q_i)$$

for an arbitrary nonempty collection $\{q_i \mid i \in I\}$ of predicates. Furthermore, a conjunctive predicate transformer S is also monotonic: $p \subseteq q \;\Rightarrow\; S.p \subseteq S.q$.

Later in the chapter we make use of the following property which shows how we can use conjunctivity of a statement if we are interested only in some specific initial and final states (i.e., a specific context) defined correspondingly by precondition and postcondition collections $\{p_i \mid i \in I\}$ and $\{q_i \mid i \in I\}$:

Lemma 11.1 *Assume that S is a conjunctive predicate transformer, and σ a state. Furthermore, assume that $\{p_i \mid i \in I\}$ and $\{q_i \mid i \in I\}$ are collections of predicates, where i ranges over some type I. Then*

$$(\forall i \mid i \in I \bullet p_i.\sigma \Rightarrow S.q_i.\sigma) \quad \equiv \quad (\exists i \mid i \in I \bullet p_i.\sigma) \Rightarrow S.(\cap i \mid p_i.\sigma \bullet q_i).\sigma$$

Proof We first prove forward implication. Assume $(\forall i \bullet p_i.\sigma \Rightarrow S.q_i.\sigma)$ and $(\exists i \bullet p_i.\sigma)$. Then the set $\{i \mid p_i.\sigma\}$ is nonempty, so

$$S.(\cap i \mid p_i.\sigma \bullet q_i).\sigma$$

$$
\begin{array}{ll}
\equiv & \{\text{conjunctivity, nonemptiness}\} \\
& (\forall i \mid p_i.\,\sigma \bullet S.\,q_i.\,\sigma) \\
\equiv & \{\text{bounded quantification notation}\} \\
& (\forall i \bullet p_i.\,\sigma \Rightarrow S.\,q_i.\,\sigma) \\
\equiv & \{\text{assumption}\} \\
& \mathsf{T}
\end{array}
$$

For the reverse implication, we assume $(\exists i \bullet p_i.\,\sigma) \Rightarrow S.\,(\cap i \mid p_i.\,\sigma \bullet q_i).\,\sigma$. Then

$$
\begin{array}{ll}
& p_i.\,\sigma \\
\Rightarrow & \{\text{assumption}\} \\
& S.\,(\cap j \mid p_j.\,\sigma \bullet q_j).\,\sigma \\
\Rightarrow & \{\text{monotonicity; intersection is subset of } q_i\} \\
& S.\,q_i.\,\sigma
\end{array}
$$

and the proof is finished. □

11.2.2 Specification Statements and Specification Variables

In this investigation we use the *specification statement* of the form $\{p\}; [Q]$ where p is a predicate and Q is a relation. Its interpretation as a predicate transformer (see Section 2.5) is as follows:

$$(\{p\}; [Q]).\,r.\,\sigma \quad \equiv \quad p.\,\sigma \wedge (\forall \sigma' \bullet Q.\,\sigma.\,\sigma' \Rightarrow r.\,\sigma') \tag{11.1}$$

In other words, if p holds in the initial state σ and Q relates σ and some final state σ', then r must hold in σ'. In such a specification, p is the *precondition* and Q is the *next-state relation*. It is easily shown that the specification-statement predicate transformer is conjunctive.

In practice, we want to express specifications using program variables. The syntactic equivalent of a predicate over σ is then a boolean term where program variables may occur free.

A specification that allows only program variable y to be changed can be written in the form $\{p\}; [y := y' \mid Q]$ where p and Q are boolean terms over the program variables. In addition, Q may mention the (bound) variable y', which stands for the final value of y. The definition (11.1) then gets the following syntactic form:

$$(\{p\}; [y := y' \mid Q]).\,r \quad = \quad p \wedge (\forall y' \bullet Q \Rightarrow r^{y}_{y'}) \tag{11.2}$$

For example, the specification

$$\{x \geq 0\}; [y := y' \mid y'^2 \leq x < (y'+1)^2]$$

states that y is to be assigned the integer square root of x (provided x is nonnegative).

We will mostly use this format for specifications. However, we will also make comparisons with Morgan's *pre-post specification statement* [84]. It has the form $y : [p, q]$, where y is the *frame*, p is the *precondition*, and q is the *postcondition*. The example above, rewritten as a pre-post specification, is

$$y : [x \geq 0, y^2 \leq x < (y+1)^2]$$

Pre-post specification statements are simple and intuitive, but they become a bit more cumbersome if one wants to refer to initial values of changed variables in the postcondition. There are two ways of overcoming this difficulty. One is to use a convention, for example that zero-subscripted variables refer to initial values (some formalisms use priming conventions, but the idea is the same). This is syntactically simple but it introduces a new kind of entity into the logic (zero-subscripted variables) which makes the rules more complicated. A logically cleaner solution is to introduce a new binding construct. Morgan uses blocks with *logical constants*[84] of the form $[\![\mathsf{con}\ c \bullet S]\!]$. In the semantics of this construct the variable c is existentially quantified:

$$[\![\mathsf{con}\ c \bullet\ S]\!].q \quad = \quad (\exists c \bullet S.\, q) \tag{11.3}$$

Using a logical constant to hold the initial value, we can express the specification

$$\{y \geq 0\}; [y := y' \mid y'^2 \leq y < (y'+1)^2]$$

equivalently as

$$[\![\mathsf{con}\ c \bullet\ y : [y \geq 0 \wedge y = c,\ y^2 \leq c < (y+1)^2]]\!]$$

As a predicate transformer, a con-block is not generally conjunctive. However, in this case the conjunct $y = c$ in the precondition implies that there is exactly one possible initial value for c and conjunctivity is guaranteed.

11.3 Hoare Triples as Specifications

Recall that a program statement is called correct with respect to precondition p and postcondition q if any execution from an initial state in p is guaranteed to terminate in a final state in q. This traditional notion of (total) correctness is defined in the refinement calculus as follows:

$$\{p\}\ S\ \{q\} \quad \equiv \quad p \subseteq S.\, q$$

for a predicate transformer S and predicates p and q.

Correctness and refinement can be connected through specification statements. If S is conjunctive, then

$$\{p\}\ S\ \{q\} \quad \equiv \quad y : [p, q] \sqsubseteq S \tag{11.4}$$

(this is proved in [81]). Equivalently, this can be stated as follows:

$$\{p\}\ S\ \{q\} \quad \equiv \quad \{p\}; [y := y' \mid q^y_{y'}] \sqsubseteq S \tag{11.5}$$

This means that $\{p\}; [y := y' \mid q^y_{y'}]$ is the most abstract specification of a statement S that satisfies the correctness triple $\{p\}\ S\ \{q\}$; any (conjunctive) statement that satisfies this triple must be a refinement of $\{p\}; [y := y' \mid q^y_{y'}]$.

Now consider the following problem. A statement S is specified by the Hoare triple $\{p\}\ S\ \{q\}$ and we want to find an acceptable precondition of S with respect to some given postcondition r. In other words, we want a rule of the form

$$\frac{\{p\}\ S\ \{q\}}{\{?\}\ S\ \{r\}} \tag{11.6}$$

and the problem is to fill in the place of the question mark. In particular, we want to replace the question mark with a predicate that is as weak as possible. The rule is said to be sharp if it gives the weakest possible precondition.

From (11.5) we see that the weakest precondition of $\{p\}; [y := y' \mid q^y_{y'}]$ with respect to r is an acceptable precondition. We expand using (11.2) to get the precondition

$$p \wedge (\forall y' \bullet q^y_{y'} \Rightarrow r^y_{y'}) \tag{11.7}$$

In fact, replacing the question mark in (11.6) with this precondition gives us a sharp rule, since (11.5) is an equivalence rather than just an implication.

11.3.1 Handling Specification Variables

Correctness triples are restricted as a means of specification since they do not allow us to refer to initial values of variables in the postcondition. To overcome this restriction, *specification variables* (sometimes called logical variables) are used. A specification with specification variables has the form

$$\{p[m]\} \;\; S \;\; \{q[m]\}$$

where $p[m]$ is the same as p but with an explicit indication that a specification variable m may occur free. Such a specification should be interpreted as saying that the correctness condition must hold for any value of m (so m is implicitly universally quantified).

We are now faced with the following problem: *If we reformulate this specification as a specification statement, how should the specification variables be interpreted?*

The obvious solution seems to be to use the con block, treating a specification of the form $\{p[m]\}\ B\ \{q[m]\}$ as $[\![\ \mathsf{con}\ m \bullet y : [p, q]\ ]\!]$. The semantics of the con block and the pre-post specification now gives us the precondition

$$(\exists m \bullet p \ \wedge \ (\forall y' \bullet q^y_{y'} \Rightarrow r^y_{y'})) \tag{11.8}$$

for the question mark in (11.6). This corresponds to treating m as an ordinary free variable (i.e., as implicitly universally quantified) in the correctness assertion.

Does this interpretation then give us the weakest possible precondition? The following theorem shows that in general this is not the case.

Theorem 11.1 *Assume that S is conjunctive and changes only the variable(s) y. Then the correctness assertion*

$$\{p[m]\}\ S\ \{q[m]\}$$

is valid if and only if the refinement

$$\{\exists m \bullet p[m]\}; [y := y' |\ \forall m \bullet p[m] \ \Rightarrow \ q[m]^y_{y'}] \quad \sqsubseteq \quad S$$

holds.

Proof

$$\begin{array}{ll} & \{p[m]\}\ S\ \{q[m]\} \\ \equiv & \{\text{semantic interpretation}\} \\ & (\forall m \bullet \forall \sigma \bullet p[m].\,\sigma \ \Rightarrow \ S\,(q[m]).\,\sigma) \end{array}$$

$\equiv$ {swap quantifiers}
$(\forall \sigma \bullet \forall m \bullet p[m].\sigma \;\Rightarrow\; S.(q[m]).\sigma)$
$\equiv$ {Lemma 11.1, S assumed conjunctive}
$(\forall \sigma \bullet (\exists m \bullet p[m].\sigma) \;\Rightarrow S.(\cap m \mid\; p[m].\sigma \bullet q[m]).\sigma)$
$\equiv$ {S monotonic (see explanation below)}
$(\forall \sigma \bullet (\exists m \bullet p[m].\sigma) \;\Rightarrow (\forall r \bullet ((\cap m \mid\; p[m].\sigma \bullet q[m]) \subseteq r) \Rightarrow S.r.\sigma)\;)$
$\equiv$ {moving quantifier outside}
$(\forall r\; \sigma \bullet (\exists m \bullet p[m].\sigma) \;\Rightarrow (((\cap m \mid\; p[m].\sigma \bullet q[m]) \subseteq r) \Rightarrow S.r.\sigma)\;)$
$\equiv$ {shunting property $(A \Rightarrow (B \Rightarrow C) \;\equiv\; A \wedge B \;\Rightarrow\; C)$}
$(\forall r\; \sigma \bullet (\exists m \bullet p[m].\sigma) \;\wedge\; ((\cap m \mid\; p[m].\sigma \bullet q[m]) \subseteq r) \;\Rightarrow\; S.r.\sigma\;)$
$\equiv$ {definitions of set operations}
$(\forall r\; \sigma \bullet (\exists m \bullet p[m].\sigma) \;\wedge\; (\forall \gamma \bullet (\forall m \bullet p[m].\sigma \Rightarrow q[m].\gamma) \Rightarrow r.\gamma) \;\Rightarrow\; S.r.\sigma\;)$
$\equiv$ {definition of specification statement}
$(\forall r\; \sigma \bullet \{\exists m \bullet p[m]\}; [\lambda \sigma \gamma \bullet (\forall m \bullet p[m].\sigma \;\Rightarrow\; q[m].\gamma)].r.\sigma \;\Rightarrow\; S.r.\sigma)$
$\equiv$ {definition of refinement}
$\{\exists m \bullet p[m]\}; [\lambda \sigma \gamma \bullet (\forall m \bullet p[m].\sigma \;\Rightarrow\; q[m].\gamma)] \sqsubseteq S$
$\equiv$ {switch to syntactic form}
$\{\exists m \bullet p[m]\}; [y := y' |\; \forall m \bullet p[m] \;\Rightarrow\; q[m]^y_{y'}] \sqsubseteq S$

The fourth proof step uses the following property which holds for arbitrary monotonic S:

$$(\;\forall q\; \sigma \bullet\; S.q.\sigma \;\;\equiv\;\; (\forall r \bullet (q \;\subseteq\; r) \;\Rightarrow\; S.r.\sigma)\;)$$

□

11.3.2 Discussion

Theorem 11.1 has a number of consequences. First, it shows that a Hoare triple specification $\{p[m]\}\; S\; \{q[m]\}$ corresponds to a specification statement $\{\exists m \bullet p[m]\};\; [y := y' |\; \forall m \bullet p[m] \;\Rightarrow\; q[m]^y_{y'}]$. As a result, we get a specification statement where there are no specification variables anymore. Instead, the relationship between initial and final states is expressed directly, and m has become an ordinary bound variable.

Next, we use (11.5) and (11.2) to calculate the weakest precondition for S (specified by $\{p[m]\}\; S\; \{q[m]\}$) to establish postcondition r. We get

$$(\exists m \bullet p[m]) \;\wedge\; (\forall y' \bullet (\forall m \bullet p[m] \;\Rightarrow\; q[m]^y_{y'}) \;\Rightarrow\; r^y_{y'}) \qquad (11.9)$$

This precondition can be shown to be equivalent to the one in (11.8) if there is at most one value for the specification variable m that satisfies the precondition p in the initial state. However, if there is more than one such value of m, then (11.9) yields a weaker precondition than (11.8).

Finally, we note that Theorem 11.1 shows what real difference the assumption about statement conjunctivity makes in Hoare triple specifications with specification variables. In the world of all monotonic predicate transformers, a specification assertion of the form $\{p[m]\}\ S\ \{q[m]\}$ is equivalent to

$$[\![\ \mathsf{con}\ m.\ y : [p[m], q[m]]\ ]\!] \quad \sqsubseteq \quad S$$

This can be proved in a derivation similar to that in the proof of Theorem 11.1. However, if we restrict ourselves to conjunctive predicate transformers, we get the sharper result in Theorem 11.1.

11.3.3 Mechanisation of the Results

The theoretical results presented in this chapter were proved as HOL theorems in the HOL mechanisation of the refinement calculus theory. Here we present the HOL theorems of Lemma 11.1, the monotonicity property used in Theorem 11.1, and the Theorem 11.1 itself.

```
val conj_lemma =
  ⊢ ∀C p q.
       conjunctive C ⇒
       ((∀m. p m s ⇒ C (q m) s) =
        (∃m. p m s) ⇒ C (glb (λq'. ∃m. p m s ∧ (q' = q m))) s)

val mono_lemma =
  ⊢ ∀C. monotonic C ⇒
     (∀q s. C q s = (∀r. q implies r ⇒ C r s))

val Theorem1 =
  ⊢ ∀C p q.
       conjunctive C ⇒
       ((∀m. correct (p m) C (q m)) =
        {λs. ∃m. p m s}; nondass (λs s'. ∀m. p m s ⇒ q m s') ref C)
  : thm
```

Recall that `glb` is the greatest lower bound of predicates and `nondass` is nondeterministic assignment (demonic update) statement. The HOL proofs of these theorems follow very closely the formal steps presented in the derivations.

Finally, the fact that the con-block is the most abstract specification statement satisfying the assertion of the form $\{p[m]\}\ S\ \{q[m]\}$ when we restrict ourselves to monotonic predicate transformers, can be proved as the following HOL theorem:

```
val Theorem2 =
  ⊢ ∀C p′ q′.
       monotonic C ⇒
       ((∀m. correct (p m) C (q m)) =
        con (λm. {λs. p m s}; nondass (λs s′. q m s′)) ref C)
```

where the con-block is defined as follows:

```
val con_DEF =
  ⊢def ∀C q s. con C q s = (∃m. C m q s)
```

11.4 Application: Proof Rules for Procedures

In this section we show how the results of the previous section can be used for analysing proof rules for procedures. We assume that procedures are specified in the following way, using correctness triples:

$$\begin{array}{l}\mathsf{procedure}\ P(\mathsf{value}\ x;\ \mathsf{value-result}\ y)\\ \quad \{p\}\ \ B\ \ \{q\}\end{array}$$

where it is implicitly assumed that the value of the value parameter x cannot be changed during procedure execution and that the procedure body B is conjunctive.

11.4.1 A Basic Proof Rule for Procedures

We start with the proof rule for total correctness of procedure call proposed by Gries [46, 47]. It allows three kinds of procedure parameters: value, value-result and result. For simplicity we restrict ourselves to only value and value-result parameters (result parameters can be used as value-result parameters without loss of generality).

Gries's rule then states that the procedure call $P(a, b)$ is correct with respect to the following precondition and postcondition (so the rule shows how we can calculate a precondition for the procedure call with respect to some fixed postcondition r):

$$\{p^{x,y}_{a,b}\ \wedge\ (\forall u.\ q^{x,y}_{a,u}\ \Rightarrow\ r^{b}_{u})\}\quad P(a,b)\quad \{r\} \tag{11.10}$$

We can equivalently specify the procedure P as

$$\{p\}; [y := y' \mid q^y_{y'}]$$

according to (11.5). Thus the derived precondition of Gries's rule (11.10) follows directly from (11.7), with suitable substitutions to take the actual parameters into account. In fact, (11.7) shows that the rule in (11.10) is sharp.

11.4.2 Procedures with Specification Variables

Let us now consider the case when specification variables are used to allow initial values of variables to be mentioned in the postcondition of the procedure specification. A procedure specification with specification variables has the form

$$\begin{array}{l}\mathsf{procedure}\ P(\mathsf{value}\ x;\ \mathsf{value-result}\ y)\\ \quad \{p[m]\}\ \ B\ \ \{q[m]\}\end{array}$$

Gries and Levin [47] give a proof rule for such a procedure, stating that the predicate

$$(\exists m \bullet p^{x,y}_{a,b}\ \wedge\ (\forall u \bullet q^{x,y}_{a,u}\ \Rightarrow\ r^b_u)) \tag{11.11}$$

is a derived precondition of the procedure call $P(a, b)$, for postcondition r. In fact, this rule was proposed earlier (in a partial correctness framework) by Hoare [55], under the name of the *Rule of Adaption*. Note that predicate in (11.11) is exactly what we get from (11.8), i.e., when specification variables are handled as being bound in a con block.

11.4.3 The Sharpness Problem

The discussion in Section 11.3.2 now suggest that the rule in (11.11) is not sharp, i.e., it does not always give the weakest possible precondition. Actually, this has been demonstrated by Bijlsma, Matthews and Wiltink [20]. As a counterexample, they give a procedure for rounding a real number to a nearby integer. The procedure has value parameter x (ranging over reals) and value-result parameter y (ranging over integers) and is specified (using the specification variable m) in the following way:

$$\begin{array}{l}\mathsf{procedure}\ P(\mathsf{value}\ x;\ \mathsf{value-result}\ y)\\ \quad \{m \le x \le m+1\}\ \ B\ \ \{y = m\ \vee\ y = m+1\}\end{array}$$

As examples of procedure bodies satisfying this specification, obvious procedures like $Floor(x, y)$ or $Ceil(x, y)$ can be mentioned.

Now consider the call $P(a, c)$ with postcondition $c = 0$. From (11.11) we get the derived precondition *false*, although $a = 0$ is in fact a valid precondition. Bijlsma, Matthews and Wiltink only give an informal argument for this, but the following argument demonstrates that their claim is true, when the body B is required to be conjunctive. We specialise m in the correctness formula twice, once to -1 and once to 0. That gives two correctness formulas

$$\{-1 \leq x \leq 0\} \;\; B \;\; \{y = -1 \;\vee\; y = 0\}$$

and

$$\{0 \leq x \leq 1\} \;\; B \;\; \{y = 0 \;\vee\; y = 1\}$$

We then use the following rule which holds for arbitrary conjunctive S

$$\frac{\{p\}\ S\ \{q\} \qquad \{p'\}\ S\ \{q'\}}{\{p \wedge p'\}\ S\ \{q \wedge q'\}}$$

to find that B must be correct with respect to the precondition $x = 0$ and the postcondition $y = 0$.

11.4.4 A Sharp Proof Rule with Specification Variables

After demonstrating that the Gries-Levin rule is not sharp, Bijlsma, Wiltink and Matthews propose their own rule which they prove to be sharp, by a long and complicated argument involving induction over the syntax of the programming notation. According to their rule, the weakest precondition of the procedure call is

$$(\exists m \bullet p^{x,y}_{a,b}) \;\wedge\; (\forall u \bullet (\forall m \bullet p^{x,y}_{a,b} \;\Rightarrow\; q^{x,y}_{a,u}) \;\Rightarrow\; r^{b}_{u}) \tag{11.12}$$

(this precondition is weaker than the one in (11.11) if there is more than one value of the specification variable m satisfying predicate p in the initial state of the procedure call).

Now compare this with what we get if we directly use (11.9) to calculate the weakest precondition for procedure call $P(a, b)$ to establish postcondition r. In fact, what we get is exactly the precondition (11.12). Thus Theorem 11.1 directly gives us the sharp rule for procedure calls in the presence of specification variables. Note also that Theorem 11.1 only required that the specified statement (the procedure body) be conjunctive. Thus we do not need to make any assumptions about the syntax of the specified statement; the conjunctivity requirement is sufficient.

11.4.5 An Alternative Argument

Let us now look at the formula in the middle of the proof of Theorem 11.1:

$$(\forall \sigma \bullet (\exists m \bullet p[m].\,\sigma)\ \Rightarrow S.\,(\cap m \mid\ p[m].\,\sigma \bullet q[m]).\,\sigma)$$

To make the correspondence between this formula and the syntactic pre-post specification even more clear, we do one additional rewriting step using the fact that program state can be modeled as a tuple. Here we assume that the procedure state is a pair with the value and the value-result parameters as components.

Rewriting the state σ as (a, b) where a are value parameters and b are value-result parameters we get

$$(\forall a, b \bullet (\exists m \bullet p[m].(a, b))\ \Rightarrow S.\,(\cap m \mid\ p[m].(a, b) \bullet q[m]).(a, b))$$

It is easy to see that this corresponds to the following syntactic correctness formula:

$$\{\exists m \bullet p[m]^{x,y}_{a,b}\}\ \ S\ \ \{\forall m \bullet p[m]^{x,y}_{a,b}\ \Rightarrow\ q[m]^{x}_{a}\}$$

Now the specification variable m is not free anymore. Therefore, we can calculate a precondition with Gries' first rule (11.10) which was already shown to be sharp.

Substituting $(\exists m \bullet p[m]^{x,y}_{a,b})$ for p and $(\forall m \bullet p[m]^{x,y}_{a,b}\ \Rightarrow\ q[m]^{x}_{a})$ for q we get the following:

$$(\exists m \bullet p^{x,y}_{a,b})\ \wedge\ (\forall u \bullet (\forall m \bullet p^{x,y}_{a,b}\ \Rightarrow\ q^{x,y}_{a,u})\ \Rightarrow\ r^{b}_{u})$$

which is again exactly the precondition (11.12) of the general rule.

11.5 Conclusions

The contribution of the work presented in this chapter is a way of translating a specification written as a correctness assertion into a specification statement of the refinement calculus. This makes it clear what is the exact status of specification variables in a correctness assertion, when the assertion is seen as a specification. Our translation is shown to be *sharp*, i.e., it yields the most abstract specification satisfying a correctness assertion.

We have shown how specification variables in Hoare triples should be interpreted to fit together with the notions of specification and refinement in the refinement calculus. For example, our investigation shows that the statement S specified by

$$\{m \le x \le m + 1\}\ \ S\ \ \{y = m\ \vee\ y = m + 1\}$$

can be specified using the specification statement

$$\{\exists m \bullet m \leq x \leq m+1\};$$
$$[y := y' \mid \forall m \bullet\ m \leq x \leq m+1\ \Rightarrow\ y' = m\ \vee\ y' = m+1]$$

Furthermore, we find that this is not equivalent to a specification of the form

$$[\![\ \mathsf{con}\ m \bullet y : [m \leq x \leq m+1, y = m\ \vee\ y = m+1]\ ]\!]$$

However, the two specifications are closely related: the former is the least conjunctive refinement[1] of the latter.

We illustrated the results by applying them to procedure correctness. Traditionally, the refinement calculus framework has not considered correctness rules for procedures. Instead, work has concentrated on specification and refinement of procedures and procedure calls [5, 80]. Our analysis shows how correctness rules for procedures can be derived and analysed efficiently within the framework of the refinement calculus.

In particular, we exhibit a short derivation of a sharp rule for procedures in the presence of specification variables. The rule is known from earlier work [20], but there it is not explained how to arrive at this rule, and the proof of sharpness is quite long and complicated.

E.Olderog in [91] discusses the relative incompleteness of Gries's proof rule for procedures and presents an Adaptation rule which yields a weakest precondition that is very similar (but not the same) as ours. However, he considers only partial correctness, and his notion of relative completeness is not the same as our notion of sharpness.

K.Engelhardt and W.-P.de Roever[36] study simulation of specification statements in Hoare logic. However, they go in the opposite direction, reducing specification statements to Hoare triples in order to prove data refinement by means of simulation.

The results presented in this chapter provide a good basis for mechanisation of specification statements written as Hoare triples. A program fragment, for example the body of a procedure, could be specified by a correctness assertion which is then automatically translated into the internal HOL representation of the corresponding specification statement according to Theorem 11.1. We are planning to integrate this approach into the Procedure extension of the Refinement Calculator. It would allow us to describe procedures as Hoare triples in the interface provided when the actual internal representation (generated according to the HOL theorems we presented) is hidden by the parser and the pretty-printer.

[1] The least conjunctive refinement of a statement is the least (wrt. the refinement ordering) conjunctive statement that refines it.

Chapter 12

Modelling Functional Procedures

12.1 Introduction

In imperative programming two kinds of procedures are encountered. A call to an ordinary procedure is itself a program statement, while a call to a *functional procedure* is an expression. Thus, calls to functional procedures occur inside other expressions (in the right-hand side of an assignment or in the guard of a conditional or a loop). Many languages support functional procedures, but still they have been ignored in most theories of programming, such as Hoare logic or the refinement calculus. These theories typically do not treat expressions at all, assuming that the underlying logic handles them sufficiently.

In this chapter we describe how functional procedures can be handled in a weakest-precondition framework, where programs are identified with predicate transformers. We also show how our theory of functional procedures can be integrated into the HOL mechanisation of the refinement calculus. Thus we have a framework for reasoning in a mechanised logic about imperative programs that contains definitions of and calls to functional procedures. To make such reasoning possible in practice, we derive rules that reduce reasoning about the calling program to correctness reasoning about the body of the functional procedure.

We model functional procedures in their full generality; thus the body of a functional procedure can be built using standard specification syntax, including nondeterminism, sequential composition, conditionals and loops. Recursive procedures constitute a special challenge, but we show how they

can be handled, and provide a nontrivial example of reasoning about a recursive procedure for binary search.

The material of this chapter is based on a paper[67] written together with J.von Wright.

12.2 Functional Procedures

The general purpose of a procedure is to abstract a certain piece of code, giving it a name and then adapting it (through parameters) in different places of the program. Since procedures are program fragments, they can be modelled in the usual way, i.e., as predicate transformers (see Chapter 6).

The effect of calling an ordinary procedure is that the procedure body (adapted as described by the parameters) is executed in place of the procedure call. However, the effect of calling a functional procedure is that a value is returned, and calls to functional procedures appear inside expressions in assignments and guards. Thus, a call to a functional procedures cannot be replaced by the procedure body.

12.2.1 The Function Call Operator

Since a functional procedure is really a program fragment, we want to model it as a predicate transformer. To make this work, this predicate transformer has the argument (a tuple) as its initial state and the returned result as its final state. A call to such a functional procedure is then defined so that it extracts the state function from the predicate transformer (i.e., from the body of the functional procedure).

A `return` statement explicitly indicates what result should be returned by a functional procedure. It is defined as follows:

$\vdash_{def}$ $\forall$`e. return e = assign (`λ`u. e u)`

Note that by η-conversion, `return` and `assign` are exactly the same, but their intuitions differ: `assign` models an ordinary assignment to program variables, while `return` is used to produce the final (result) state; it should be executed as the very last statement before control returns to the calling program.

In order to use functional procedure calls in our program, we have to find a way of extracting the state function from the procedure body. The body of a functional procedure is defined as a predicate transformer. Operationally

we can see it as modelling backward execution – we supply a set of final states we are interested in (postcondition), and calculate the biggest possible set of initial states (weakest precondition) from which we guaranteed to reach the final states described by the postcondition. State functions, however, model forward execution – for a given initial state they calculate the final state that is the result of the state change. We need to find a translation that reverses execution modelled by the functional procedure body.

For a given initial state we consider all possible sets of reachable final states (postconditions). We then calculate the intersection of all such sets of states (the minimal set of reachable final states), and finally we select a value from this minimal set as the result of our state function.

This intuition is formalised in the following definition:

$\vdash_{def}$ `∀c. fcall c = (λu. εv. glb (λq. c q u) v)`

where `c` is the body of the functional procedure, `u` is the initial (argument) state, and `v` is the result value.

Modelling the body of a functional procedure, we usually require it to satisfy an additional healthiness condition[1] called *strictness*. The strictness property excludes any miraculous execution of a program statement. Most statements of our language are strict. The exceptions are **magic**, $[false]$, and $[\lambda ss'.\mathsf{F}]$. The HOL definition of strictness is as follows:

$\vdash_{def}$ `strict c = (c false = false)`

Note the use of the choice operator ε in the definition of `fcall`. It means that the result value of a function call is an arbitrary (but fixed) element from the set `glb (λq. c q u)`. If this set is empty, then we have no information whatsoever about the value that is returned. However, conjunctivity and strictness (the two healthiness conditions that we generally require) together guarantee that the set is nonempty.

As an example, we define a very simple functional procedure that squares a natural number as follows:

$\vdash_{def}$ `sqfun = return (λu. u*u)`

In a Pascal-like syntax this corresponds to something like the following:

$$\begin{array}{l}\mathsf{func}\ \mathit{sqfun}(x : \mathsf{num}) : \mathsf{num} = \\ \quad \mathsf{return}\ x * x\end{array}$$

A call to this functional procedure can then be as follows:

[1] So far we have used two healthiness conditions of program statements – monotonicity and conjunctivity.

```
assign λ(x,y).(x,y + fcall sqfun (x+1))
```

corresponding to an assignment of the form

$$y := y + sqfun(x + 1)$$

12.2.2 Basic Properties

We shall now discuss a number of basic properties of the function call operator that we have proved as HOL theorems. We start with a basic soundness property: if the body of the functional procedure is an assignment statement then the function call extracts the state change function from it:

```
fcall_assign =
⊢ ∀e. fcall (assign e) = e
```

The proof rests on the fact that in this case, the intersection `glb` $(\lambda$`q`. $($`assign e`$)$ `q u`$)$ is the singleton set $\{$`e u`$\}$ where `u` is the initial (argument) state of the functional procedure. Therefore, the choice operator actually has no choice but to select `e u` as the result value of the function call.

Implicitly the same property holds for any deterministic and terminating functional procedure body, since such a statement is semantically equivalent to a single `assign` statement.

Next, we have a property that allows us show that a functional procedure in fact implements a specific function.

```
fcall_thm =
  ⊢ ∀c f. conjunctive c ∧ strict c ∧
        (∀u0. correct (λu. u = u0) c (λv. v = f u0)) ⇒
        (fcall c = f)
```

Here, `c` is the body of the functional procedure and `f` is (the HOL formalisation of) the function that the procedure implements. The implementation property is reduced to a corresponding correctness property of the procedure body, which can then be proved using standard (Hoare logic) methods.

Finally, we have two theorems that show how the function call operator can be propagated past an initial assignment and distributed into a conditional:

```
fcall_seq =
  ⊢ ∀c e. fcall (assign e seq c) s = fcall c (e s)
fcall_cond =
  ⊢ ∀g c1 c2 s.
        fcall (cond g c1 c2) s = (g s → fcall c1 s | fcall c2 s)
```

These theorems will be important when proving properties of concrete implementations. They could also support a kind of partial evaluation using the actual parameters of a function call.

12.2.3 Example: Implementation Proofs

Consider the very simple task of finding the minimum of two numbers. In HOL we can formalise the minimum function in the following way:

```
⊢def ∀m n. MIN(m,n)  = (m < n → m | n)
```

In the imperative programming notation we now code a (slightly different) functional procedure `minfun`

```
⊢def minfun =
      cond (λ(x,y). x ≤ y)
           (return λ(x,y).x)
           (return λ(x,y).y)
```

The two arguments of the functional procedure `minfun` form the initial state (pair). The variables here are explicitly modelled as projection functions `FST` and `SND`. In a Pascal-style programming notation this would translate to

```
func minfun(x : num, y : num) : num =
  if x ≤ y then
    return x
  else
    return y
  endif
```

Now we can prove that `minfun` actually implements the HOL function `MIN`.

```
⊢ fcall minfun = MIN
```

The proof is straightforward: first we use the theorem `fcall_cond` to distribute `fcall` into the conditional statement, then we eliminate `fcall` using the basic property `fcall_assign`. After this follows a case split and then the proof is finished off by arithmetic reasoning. Note that this kind of implementation theorem is very strong: when reasoning about a program that contains a function call, we can replace the function call with the mathematical function that it corresponds to. Thus, we never have to refer to the definition of `minfun` after this.

12.2.4 Nondeterminism and Nontermination

It is natural to expect functional procedures to be deterministic and terminating, since they typically implement (total) functions. The theorem `fcall_assign` shows that in this case the function call extracts the implemented function.

However, our formalisation of functional procedures is more general than this, because it allows function bodies that are nondeterministic and/or nonterminating. An obvious question is now: what do we know about the value that a functional procedure returns in these cases?

Suppose that the body of a functional procedure is nonterminating. That means that there are no postconditions that can be satisfied by execution of the body. Expanding the definition of `fcall`, we can prove that the function call in this case returns an arbitrary (but fixed) value of the result type.

```
⊢ ∀c u. nonterminating c u ⇒ (fcall c u = (ε v.T))
```

Therefore, a nonterminating function body does not lead to a nonterminating function call (an expression in the HOL logic is always defined), but we get a return value about which we know absolutely nothing (apart from type information).

Let us now consider the case when the function body is (demonically) nondeterministic (but terminating). A simple case is when the procedure body consists of a single nondeterministic assignment statement `nondass R`. In this case, if for some given initial state (function parameters) `u` the set `R u` is not empty, then some selected element from the set `R u` is returned by the function call:

```
⊢ ∀R u. (∃v. R u v) ⇒ R u (fcall (nondass R) u)
```

A similar argument can also be used when the body of the nondeterministic functional procedure is more complex, since it is then equivalent to some nondeterministic assignment.

12.3 Correctness Reasoning with Functional Procedures

The usual way to prove that a program (or some program fragment) is correct with respect to a given precondition-postcondition pair is to decompose the global correctness property into correctness properties for the program components, using Hoare logic.

12.3.1 Correctness Proofs with Functional Procedures

When proving correctness of a program containing function calls, we use Hoare logic to decompose the proof in the ordinary way. When we get to the bottom level, we are faced with proving verification conditions that come from guards and assignments (e.g., conditions of the form $P \Rightarrow Q[x := E]$ that come from the assignment rule of Hoare logic). When function calls are present, such conditions express a relationship between the calling state, the argument to the function, and the result returned by the function call. The following theorem can then be used to reduce the condition to a correctness condition on the function body:

```
fcall_property =
  ⊢ ∀c R e u0.
      conjunctive c ∧ strict c ⇒
      correct (λu. u = e u0) c (λres. R u0 res) ⇒
        R u0 (fcall c (e u0))
```

Here c is the function body, and `R` expresses the relationship between the state from which the function is called (`u0`) and the function result (`e` is the function that says how the function argument is constructed from the calling state). Note also that `fcall_property` is a generalisation of `fcall_thm` (see Section 12.2.2). Since conjunctivity and strictness can be proved automatically, it gives a way of reducing a general property of a function call to a correctness property for the body of the functional procedure in question.

Let us use the squaring function to show how this is used in practice. Recall that it was defined as

```
⊢def sqfun = return (λu. u * u)
```

Suppose that we want to prove the following correctness assertion for the assignment statement with the function call:

```
⊢ correct (λ(x,y). T)
          (assign λ(x,y).(x,fcall sqfun (x + 1) − 1))
          (λ(x,y). y ≥ x)
```

Here the tupled abstraction makes the correspondence with the intended Hoare logic formula clear:

$$\{T\}\ y := \texttt{square\_fun}(x+1) - 1\ \{y \geq x\}$$

After applying the Hoare logic rule for assignment and simplifying, the goal is reduced to the following:

```
⊢ fcall sqfun (x + 1) − 1 ≥ x
```

We can now specialise the theorem `fcall_property` with `square_fun` for `c`, with $(\lambda(\mathtt{x},\mathtt{y})\,\mathtt{r}.\,\mathtt{r}-1\geq\mathtt{x})$ for `R`, with $(\lambda(\mathtt{x},\mathtt{y}).\,\mathtt{x}+1)$ for `e`, and with `x` for `u0`. This theorem reduces our goal (after the conjunctivity and strictness conditions have been automatically discharged) to

```
⊢ correct (λu. u = x+1) (return (λu. u * u)) (λr. r−1 ≥ x)
```

Now we apply the Hoare logic rule for assignments (recall that the `return` statement is an assignment) and the goal is reduced to

```
⊢ (x + 1) * (x + 1) − 1 ≥ x
```

which is a standard verification condition (and obviously true).

The functional procedure `square_fun` was very simple, but the same strategy works for more complex procedure bodies and more complex correctness conditions as well. The example shows how the verification conditions that arise from program correctness proofs lead to new correctness proofs, when function calls are present. Since the body of one function may contain calls to another function, new correctness conditions may appear, and so on. Eventually, however, all function calls have been handled and we reach the ground level where only basic verification conditions remain (unless there is recursion; see Section 12.4).

12.3.2 Contextual Correctness Reasoning

A function call can occur in a situation where a context (i.e., a restriction on the possible values of program variables) is known to hold. If this contextual knowledge can be expressed in the form of a predicate `p` that holds for the arguments at a call to `c`, then we can assume `p` as a precondition when reasoning about the body of the functional procedure `c`.

The theorem that captures this intuition is the following, in the case of an implementation proof:

```
fcall_thm_pre =
  ⊢ ∀c p f.
       conjunctive c ∧ strict c ∧
       (∀u0. correct (λu. (u = u0) ∧ p u) c (λv. v = f u0)) ⇒
       (∀u. p u ⇒ (fcall c u = f u))
```

A simple example of a situation where this property can be useful is when the function call occurs inside the guard of a conditional, e.g.,

```
cond (λ(x,y). x>0 ∧ fcall foo x) ...
```

In this case, we may instantiate `p` to $(\lambda u.u > 0)$ when using `fcall_thm_pre` to reason about the call to `foo`.

A similar argument can also be used when proving some general property of a function call (e.g., when reducing correctness conditions):

```
fcall_property_pre =
  ⊢ ∀c p R e u0.
       conjunctive c ∧ strict c ∧
       correct (λu. (u = e u0) ∧ p u) c (λv. R u0 v)) ⇒
       p (e u0) ⇒ R u0 (fcall c (e u0))
```

This is a direct generalisation of `fcall_property` (Section 12.3).

12.4 Recursive Functions

Recursion in the context of (ordinary) procedures can be defined using the least fixpoint (with respect to the refinement ordering on predicate transformers) of a functional that corresponds to the recursively defined procedure. This method cannot be used directly with functional procedures, since the `fcall` operator is not monotonic with respect to the refinement ordering (nor any other suitable ordering).

12.4.1 A Constructor of Recursive Functional Procedures

As a first step towards defining recursion we define an iterator.

```
⊢def (∀f. iter 0 f = assign (λs. ε s'. T)) ∧
     (∀n f. iter (SUC n) f = f (iter n f))
```

Since there is no bottom (or undefined) element to start the iteration from, we choose to start it from some element selected by the choice operator. This means that we have to be careful when defining the recursion operator: it is not sufficient that two consecutive iterations give the same result (the selection operator may cause this to happen "by accident"). However, if from some point on all further iterations give the same result, then the iteration has stabilised. This justifies the following definition

```
⊢def ∀f. fmu f =
       assign (λs. εa. ∃m. ∀n. n > m ⇒ (fcall (iter n f) s = a))
```

Of course, the problem of nontermination is the same as before: a potentially infinite sequence of recursive calls is modelled as terminating and returning a result about which we know nothing.

The following example shows how `fmu` is used when defining a recursive functional procedure (one where recursion occurs inside a `fcall`). We define

```
⊢def Factfun = λZ.
       cond (λx. x=0)
            (assign λx.1)
            (assign λx.x * fcall Z (x−1))
⊢def factfun = fmu Factfun
```

This corresponds to a Pascal-style function definition of the following form:

func $factfun(x : \mathsf{num}) : \mathsf{num} =$
  if $x = 0$ then
    return 1
  else
    return $x * factfun(x - 1)$
  endif

12.4.2 Basic Properties

The crucial property of the recursion operator `fmu` is that it gives the stabilisation point of `iter`, if such a point exists, for the arguments in question:

```
fmu_thm =
  ⊢ ∀f s k.
      (∀n. fcall (iter (k + n) f) s = g s) ⇒
      (fcall (fmu f) s = g s)
```

This property may not seem very informative, but it gives us the tools we need to prove properties of functional procedures defined with `fmu`. The argument `k` is crucial; it corresponds to a *termination argument* (an upper bound on the number of iterations needed to reach stability).

As an example, we briefly describe how one proves that `factfun` really implements the (built-in) `FACT` function of the HOL system.

According to `fmu_thm` it is sufficient to prove the following lemma

```
⊢ ∀x n. fcall (iter (SUC x + n) Factfun) x = FACT x
```

We have chosen `SUC x` as termination argument (which is reasonable when we are computing the factorial of `x`).

The proof of this lemma follows a fairly simple routine, involving only induction and rewriting with basic arithmetic facts. As a result, we immediately get the implementation theorem

```
⊢ fcall factfun = FACT
```

12.4.3 Example: binary search

The factorial example illustrates the `fmu` operator and it shows that it is possible to prove properties of a recursive functional procedure. However, it can be argued that `factfun` merely encodes the standard recursive definition of the factorial function into imperative form, and that the proof really only performs the corresponding decoding. In order to show that more realistic functional procedures can be handled, we now consider an example where the procedure does not correspond to an encoding of a standard recursive function definition.

Our example is a binary search, which in standard syntax is as follows:

```
func binfind(f : num → num, l : num, r : num, x : num) : bool =
  if r ≤ l then
    return F
  else
    ⟦ var m := (l + r) div 2;
      if f m < x then
        return binfind(f, m + 1, r, x)
      else if f m = x then
        return T
      else
        return binfind(f, l, m, x)
      endif endif
    ⟧
  endif
```

The aim is to show that if the first argument f is a monotonic function (i.e., sorted), then $binfind(f, l, r, x)$ returns T if exists i such that $l \leq i < r$ and $f\ i = x$, and it returns F otherwise.

We define a constant `Binfind` standing for the functional procedure of which the procedure `binfind` is the least fixpoint:

```
⊢ Binfind = λZ.
   cond (λ(f,l,r,x). r ≤ l)
    (return λ(f,l,r,x). F)
    ((assign λ(f,l,r,x). ((l + r) DIV 2,f,l,r,x)) seq
     (cond (λ(m,f,l,r,x). f m < x)
      (return λ(m,f,l,r,x). fcall Z (f,SUC m,r,x))
      (cond (λ(m,f,l,r,x). f m = x)
       (return λ(m,f,l,r,x). T)
       (return λ(m,f,l,r,x). fcall Z (f,l,m,x))
    )))
⊢ binfind = fmu Binfind
```

This corresponds exactly to the standard syntax above. The assignment

```
assign λ(f,l,r,x). ((l + r) DIV 2,f,l,r,x)
```

corresponds to the block entry, adding the new variable (m) as a first state component. No explicit block exit is needed; it is taken care of by the `return` statement.

The correctness of the binary search depends on the first argument being sorted. Thus, the theorem that we want to prove is the following:

```
⊢ ∀f x.
     (∀i j. i < j ⇒ f j ≤ f j) ⇒
     (∀l r. fcall binfind (f,l,r,x) =
              (∃i. l ≤ i ∧ i < r ∧ (f i = x)))
```

Exactly as for the simple example in Section 12.4.2, the crucial lemma shows that iteration of `Binfind` is guaranteed to terminate with a correct answer. The lemma has the following form:

```
⊢ ∀f:num→num. ∀x:num.
     (∀i j. i < j ⇒ f i ≤ f j) ⇒
     (∀d k l. fcall (iter (SUC d + k) Binfind) (f,l,l+d,x) =
                (∃i. l ≤ i ∧ i < l + d ∧ (f i = x)))
```

The critical part of the proof is an induction over d (the length of the search interval), which is also the termination argument. Since the termination argument is (approximately) halved rather than decreased by one, we must use general well-founded induction:

```
⊢ ∀P. (∀n. (∀m. m < n ⇒ P m) ⇒ P n) ⇒ (∀n. P n)
```

rather than standard induction over the natural numbers. The proof strategy is essentially the same as in the factorial example, but here we need to push `fcall` both into conditionals and past assignments (see Section 12.2.2).

The proof then reduces to basic arithmetic facts (including tedious details about integer division) and to simple properties of monotonic functions. The following two are typical examples of lemmas used in the proof:

```
⊢ (∀i j. i < j ⇒ f i ≤ f j) ∧ (f i = x) ∧ f j < x ⇒ j < i
⊢ m > 0 ⇒ m DIV 2 < m
```

This proof follows a general strategy that can be used in similar proofs. However, automating this strategy does not seem feasible, at least not when general well-founded induction is used. In this example finding the right instantiations for `d`, `k`, and `l` in the lemma required elaborate equation solving. Furthermore, assumptions about the state (such as the monotonicity assumption on `f`) may be used in nontrivial ways.

The proof also depends on pushing `fcall` into the structure of the functional procedure, and this only works going into conditionals and past assignments. Thus, the same strategy cannot be used if nondeterministic constructs are involved, or if there are assignments after a recursive call. It is not clear whether there exists a useful proof strategy in these situations.

12.5 Discussion

We have presented an approach for modelling functional procedures in a weakest precondition framework. We are explicitly interested in functional procedures as they occur in Pascal-like programs, i.e., where the procedure is described as an imperative program even though the call is used as an expression. Thus we cannot use existing theories of functional programs (e.g., [32, 97]). Instead, functional procedures are handled through the link between assignments and state transforming functions. To our knowledge, they have not treated in this way before.

We integrated this approach into the mechanised version of the refinement calculus in HOL system. The HOL formalisation of the refinement calculus contains support for correctness reasoning about programs, and we reuse it for correctness reasoning where calls to functional procedures occur.

Two ways of proving (correctness) properties of function calls were described. If the functional procedure is characterised by an implementation theorem then the function call can be replaced directly by a reference to the corresponding (mathematical) function. In other cases, the proof leads to correctness proofs for the body of the functional procedure.

Our approach for modelling functional procedures allows function bodies to be nondeterministic and/or nonterminating. Since the result of a function

call is selected using the choice operator ε, it is always well-defined. In the nondeterministic case, the result returned by the function call is an arbitrary but fixed value of the form εP (where P is some nonempty set). Thus the value returned by the function call is deterministic, but the only information we can ever get about it is that it belongs to the set P. In this sense our approach differs significantly from earlier attempts to do model expressions and expression refinement in a weakest precondition framework[74, 87, 96]. In our framework, a refinement of the body of a functional procedure does not lead to a refinement of the calling program (but proofs of properties of the calling program can generally be reused, as long as it does not make essential use of the selected element εP, and that is very rarely needed[2]).

The use of the choice operator ε in the function call definition also means that the problem of nontermination is solved in rather simplistic way. Nontermination in the body of the functional procedure cannot cause an "infinite looping" function call, since our expressions are always well-defined. Instead, a nonterminating body leads to a return value about which we know nothing.

The work presented can be extended by investigation of possible semi-automated strategies for proofs of implementation and correctness, both for simple and recursive functional procedures. The approach described in this chapter also provides a good basis for adding functional procedures to the Refinement Calculator tool.

[2]Essential use of the selected element εP means that the property proved only holds for this fixed arbitrary element belonging to the set P and does not hold for any other element from this set.

Chapter 13

Conclusions & Future Work

13.1 Conclusions

General remarks The refinement calculus is a powerful formalism for developing provably correct programs starting from their precise mathematical descriptions (specifications). However, since actual proofs tend to become too large and complicated, and therefore unmanageable, we need a mechanical assistance to guarantee their soundness.

This dissertation has presented new ideas and techniques to support mechanical development of programs on the basis of the refinement calculus theory. Most of the ideas and techniques presented in the thesis have been implemented within the Refinement Calculator – a tool for program refinement being developed at Åbo Akademi University.

Our refinement theory is mechanised in the theorem prover HOL. HOL provides us with security for expressing and proving complex refinement rules. The weakest precondition semantics of our programming language is given definitionally in higher order logic. The use of a shallow embedding within a classical logic allows us to freely mix our refinement logic with ordinary logic. All this increases our confidence that the refinement rules presented in this thesis are sound, and derivations done by the tool are logically accurate.

The work presented in the thesis relies heavily on the work of other people involved in the Refinement Calculator project. Specifically, we have extensively used general refinement and logical transformations, data refinement techniques, window inference extensions, and parsing and pretty-printing facilities that have been developed and integrated into the Refinement Calculator by other members of our group.

The main significance of our contribution lies in the treatment of procedures and procedure calls in the weakest precondition semantics (formalised in higher-order logic) and its implementation in a workable and practical fashion in the Refinement Calculator tool. That allows us to create and reason formally about modular units of programs such as procedures and functions. As a result, the standard program refinement techniques implemented in the Refinement Calculator can be applied for formal top-down development of larger programs.

The investigation of refinement and correctness properties of procedures and their calls in our model has led to a number of novel and interesting results concerning the relationship between Hoare triples and specification statements in the refinement calculus, the connection between procedure variables and data refinement, the treatment of functional procedure calls etc. All these results are proved or implemented in the rigorous environment of the HOL system.

Next we overview each of the main parts of the thesis in more detail.

Overview of main results Chapter 4 presents our mechanisation of an abstract lattice theory and its integration with the Refinement Calculator tool. This work allows us to automatically instantiate abstract lattice properties for concrete domains and use them on different abstraction levels of the program model that we are working with. Our approach is very similar to the one taken by Gunter[50] in her HOL formalisation of abstract group theory. Windley[105] has implemented a package for using abstract theories in the HOL system. The package allows us to define abstract theories algebraically, i.e., by defining abstract operations axiomatically via their basic properties. This package provides an alternative way to define an abstract lattice theory in the HOL system. However, Windley associates the underlying set of abstract elements with a HOL type. Our approach is more flexible since it allows us to use any set of elements as the underlying set of an abstract theory.

The use of abstract lattice properties for program derivations is not that apparent in the rest of the thesis. This can be explained by the fact that abstract lattice properties are mostly used on the level of program predicates and relations, and we often omitted this level of detail when presenting our program derivations.

At the high level of program statements the most notable example of the application of the abstract lattice properties is recursion introduction as a specialisation of the lattice property of the least fixpoint introduction. This

rule is very useful for reasoning about loops and recursive procedures.

In future our work can be extended with facilities for handling monoid structures which interact with the lattice ordering (e.g., so that the composition operation is monotonic in both arguments with respect to the ordering).

In Chapter 5 we described our work on the implementation of two dual approaches for introduction, calculation and the use of context information in the refinement calculus framework. While working on the mechanisation of the rules for handling context information, we focused on two main issues: the first was the proof of sharpness of the rules, and the second was the implementation of them within the Refinement Calculator tool.

There are several stubborn problems regarding context which are worth discussing. The first problem is how to manage an explosive growth of context information while propagating it through large program fragments. This can lead to difficulties in retrieving the information which is the most relevant for refinement of a certain subterm. The problem can be tackled by hiding (using a pretty-printer) a part of irrelevant accumulated context information by focusing attention/projecting on the program variables of interest. Also, we are planning to take advantage of the new HOL techniques for automatic simplification that have become available in the most recent releases of the HOL system.

The propagation of context information through loops is in general cumbersome because of the absence of sharp context propagation rules for loops. The current solution suggests that a loop invariant is propagated as the only available context information. This situation can be improved by implementing the approaches described in the thesis for strengthening a loop invariant and for propagation of context information that is not affected by loop execution.

Another problem is the propagation of context information through procedure calls. The two approaches presented suggest either a complete disclosure via unfolding of a procedure call or treating a procedure as a "black box". We believe that the optimal way could be a compromise solution proposed by Staples[99], e.g., to use an initial specification of a procedure to calculate a sufficient and manageable amount of context information.

The main part of the thesis is devoted to modelling procedures and procedure calls in the weakest precondition semantics. This allows us to prove a number of semantic properties of procedures and their calls that were just postulated in traditional syntactic approaches.

The implementation of our approach in the HOL system leads to an increase in scalability of the mechanised refinement calculus, because it allows us to deal formally with bigger programs by providing a possibility

to create and reason about modular units of programs (such as procedures and functions). We illustrated it by an example presented in Chapter 7 which demonstrated how our framework can facilitate the top-down formal development of programs.

The next step in this direction would be a generalisation of the approach by introducing modules as collections of procedures working on a local state. We present our proposal for how modules could be implemented in the next section.

We use a simple and intuitive approach to modelling procedures by means of the *let* construct. Hence we consider procedures to be independent context assumptions for the main program. Therefore, the global program state can be accessed by a procedure only via procedure parameters. This restriction means that we do not model global variables. However, our mechanisation can be easily extended by modelling global variables as "syntactic sugaring", i.e., by modifying the parser and pretty-printer in such a way that global variables declared in a procedure definition are automatically added as additional reference parameters in a procedure call.

In the thesis we have not considered data refinement of procedures and their calls. Data refinement of local variables of a procedure is easily implementable since these variables are modelled as local variables of the corresponding block statement (the procedure body). However, data refinement of the variables that occur in the list of reference parameters is directly impossible since the procedure definition is out of scope of the program that we are data refining. A possible solution could be to "wrap" the procedure call using abstraction and representation statements that are used in data refinement. Such a "wrapping" is usually called *interface refinement*[77], and a mechanisation of it is one of possible extensions of our work.

Using tuples for modelling the program state provides an extra overhead in our formalisation. When modelled as a tuple, the program state is intuitively simple. However, the order of the elements is important in tuples while program variables supposed to be independent. We often need to prove that after a certain permutation (reordering) of the state we are still working with the same state, and this can be tedious and annoying. However, this overhead is hidden from the user by the automation provided.

To reduce the complexity of our formalisation of procedures, we allow procedures to be defined only in a fixed order which determines their scope. This excludes the possibility of mutually dependent procedures. We see dropping this restriction as a possible future extension of our work. We discuss it in detail in the next section.

Work related to the mechanisation of procedures has mostly been done in

the area of program verification. Mechanisation has been provided by special programs called *Verification Condition Generators* (VCG) which reduce the task of proving program correctness to proving a set of generated lemmas of a much simpler form. We should mention VCGs developed by Igarashi, London, and Luckham[59], Boyer and Moore[22], and Gray[45] that included treatment of procedures modelled in axiomatic semantics. However, these verification condition generators were not themselves verified. This means that any proof using and relying on these VCGs tools might not be sound, even if all the verification conditions were correctly proven. The notable exception is the work of Homeier[57, 58]. He has written and verified his VCG using the HOL theorem prover. In his work Homeier models a purposely-built imperative programming language (which includes mutually-recursive procedures) in an operational semantics. His VCG then automatically generates the verification conditions needed to prove termination and other correctness properties of a program.

In his thesis, Staples[99] has presented a mechanisation of a refinement-based approach for program development in the Isabelle theorem prover. In his work program statements are modelled using a weakest precondition semantics defined in an untyped set theory. To model program states and variables, Staples uses dependently typed functions to represent program states as an underspecified map from variable names to their values. His treatment of procedures is similar to ours in the sense that he also uses the *let* construct to bind a main program and a procedure. Adaption of the procedure body in place of a procedure call is controlled by special initialisation and finalisation functions, the purpose of which is similar to state reordering functions used in our approach.

13.2 Possible Future Extensions

In this section we present our proposals for future extensions of the work presented in this thesis.

Modules Procedures can be very useful as a structuring tool for developing large programs. But talking about real software systems, we should take one step further and consider how groups of procedures can themselves be organised into larger units called *modules*. A module corresponds to data abstraction in the sense that it introduces local variables (state) that can be accessed and updated only by procedures (sometimes called *methods*) provided by a module. Therefore, the main purpose of modules is to *encapsulate*

(i.e, make hidden) their data and all aspects of their use.

Is it possible to generalize our approach from procedures to modules as well? When we are talking about modules, we usually associate two things with them – the introduction and initialisation of a local module state (variables), and the declaration of a number of module procedures that can access and update the module state. We can present an abstract module $M1$ in the following way:

$$
\begin{array}{l}
\textbf{module } M1 \ = \\
\quad \textbf{var } u \ : \ T \mid Init; \\
\quad \textbf{procedure } P_1 \ (\textbf{val } v_1; \ \textbf{var } u, r_1) \\
\quad ... \\
\quad \textbf{procedure } P_n \ (\textbf{val } v_n; \ \textbf{var } u, r_n)
\end{array}
$$

where u are the local variables of module $M1$, $Init$ is an initialisation predicate for the local state, and $P_1, ..., P_n$ are procedures of a module $M1$. The types of these procedures are of the form $Ptrans(\Sigma_i \times T \times \Gamma_i)$ where Σ_i and $T \times \Gamma_i$ are the value and reference parameter types of procedure P_i. Note that the local variables u are explicitly included into the lists of reference variables of all module procedures.

A module and a program can be then composed into one unit by a **use** operator which is defined in the following way:

$$
\begin{array}{l}
\textbf{use } M1 \textbf{ in } \ < program > = \\
\quad \textbf{let } P_1 \textbf{ in} \\
\qquad \textbf{let } P_2 \textbf{ in} \\
\qquad ... \\
\qquad\quad \textbf{let } P_n \textbf{ in} \\
\qquad\qquad |[\textbf{var } u \ : \ T \mid Init. \ < program >]|
\end{array}
$$

Here $< program >$ can contain calls to the procedures defined in the module. However, the difference between these calls and ordinary procedure calls is that the reference to the module state u should automatically be added to the list of reference parameters of a procedure call.

Any particular module could be defined in the HOL system as a pair of the form $(\texttt{Init}, (\texttt{P}_1, ...\texttt{P}_n))$. In general it would be desirable to define modules inductively as HOL objects. That would allow us to make inductive definitions of operations on modules (such as `use`), and to prove general properties of modules as HOL theorems. For example, it would then be

possible to define the notion of module refinement `mref` and to prove that the `use` operation is monotonic with respect to module refinement:

```
⊢ ∀M1 M2 C. M1 mref M2 ⇒ (use M1 C) ref (use M2 C)
```

However, module procedures can have different parameters and, therefore, be of different types, and a collection of them can only be modelled as an n-tuple (general tuple). Unfortunately, there is no way of reasoning inductively about n-tuples in the HOL system. The alternative solution could be to define modules at the meta (SML) level as a collection of HOL terms under a given name corresponding to a module name, and then to extend the parser/pretty-printer with a `use` operator so that for any particular case of `use` < `module` > < `program` > it would generate the internal HOL representation according to the definition of `use` given above.

Since the composition of a module and a program can be expanded into a collection of nested `let` operators and a block statement of a special form, we can prove refinement of a program or of the procedures of a module in the same way as we did before. A refined module, however, can be saved under a new name, and used later in other programs.

Mutually dependent procedures When modelling procedures, we have imposed the restriction that procedures cannot be mutually dependent. The order in which procedures are declared is very important since it determines the scope in which a particular procedure is visible and can be called. In this way we exclude the possibility of mutual recursion in procedure calls.

In a paper[64] written together with L.Mikhajlov and E.Sekerinski we dropped this restriction, studying a more general layout in which two components (modules) are calling methods of each other in an arbitrary way. Furthermore, no restrictions are imposed on the order in which methods of modules are declared.

Here we briefly describe the way we are modelling such a component system. A mechanisation of this approach is a part of future work.

Any component system can be seen as consisting of two components $\mathcal{A}$ and $\mathcal{B}$. Suppose that $\mathcal{A}$ has m and $\mathcal{B}$ has n methods. The components communicate by invoking each other's methods and passing parameters. For simplicity, we model method parameters by global variables that the methods of both components can access in turns. Due to encapsulation the type of the internal state of the other component is not known, we say that the body of a method of the component $\mathcal{A}$ has the type $Ptran(\Sigma \times \Delta \times \beta)$, where Σ is the type of $\mathcal{A}$'s internal state, Δ is the type of global variables

modeling method parameters, and β is a type variable to be instantiated with the type of the internal state of the other component during composition. As the internal state of the other component is not accessible, we assume that methods of $\mathcal{A}$ operate only on their internal state and the state representing method parameters and are, therefore, of the form $S \| \mathbf{skip}$. Similarly, methods of $\mathcal{B}$ have bodies that are of the form $\mathbf{skip} \| S$ and of type $Ptran(\alpha \times \Delta \times \Gamma)$, where α is a type variable.

The behaviour of a component method depends on the behavior of the methods it invokes. We can model a method of the component $\mathcal{A}$ as a function taking a tuple of method bodies and returning a method body:

$$a_i \ \widehat{=}\ \lambda Bb \bullet ab_i$$

If we introduce an abbreviation Ψ^n to stand for $\Psi \times ... \times \Psi$ with n occurrences of Ψ, we can write out the type of a_i as $Ptran^n(\Sigma \times \Delta \times \beta) \rightarrow Ptran(\Sigma \times \Delta \times \beta)$, where n is the number of methods of $\mathcal{B}$. Methods of $\mathcal{B}$ are defined in the same manner. Accordingly, we can collectively describe all methods of $\mathcal{A}$ as a function A given as follows:

$$A \ \widehat{=}\ (\lambda Bb \bullet (ab_1, ..., ab_m)) : Ptran^n(\Sigma \times \Delta \times \beta) \rightarrow Ptran^m(\Sigma \times \Delta \times \beta)$$

Therefore, the component $\mathcal{A}$ is a tuple (a_0, A), where $a_0 : \Sigma$ is an initial value of the internal state and A is the function defined above. The definition of the component $\mathcal{B}$ is similar but with the corresponding differences in typing.

Composing components $\mathcal{A}$ and $\mathcal{B}$ results in a component system that has methods of both components with all mutual calls resolved. Using fixpoints[1], we could define such a system in the following way:

$$(\mathcal{A}\,\mathbf{comp}\,\mathcal{B}) \ \widehat{=}\ ((a_0, b_0), (\mu(A \circ B),\ \mu(B \circ A)))$$

Note that during composition, the type variables α and β, representing the unknown state spaces of the components $\mathcal{B}$ and $\mathcal{A}$, get instantiated with Σ and Γ respectively, so that the composed system has methods operating on the state space $\Sigma \times \Delta \times \Gamma$.

Note that in the special case $\mathcal{B} = \mathcal{A}$ we get a module defining a set of procedures that can be calling each other in an arbitrary way. Such a module can be defined as

$$\mathcal{A} \ \widehat{=}\ (a_0, \mu(A))$$

[1] In the paper[64] we present a detailed argument explaining why we need fixpoints.

The same approach can be applied to defining mutually dependent procedures into a main program. In this case we have

$$let\ (ab_1, ab_2, ...ab_n)\ =\ \mu(A)\ \ in\ \ \ <main\ program>$$

Of course, working with procedures or modules defined in this way involves extensive dealing with fixpoints and their properties.

Other extensions There are still some topics of this thesis that would be interesting to investigate further. One possible extension is the mechanisation of interface refinement which would allow us to "wrap" procedure calls during data refinement. Another possibility is to implement a "grey box" approach which would permit us to propagate context information through procedure calls. It would also be interesting to investigate how to mechanise the new axiomatic model of the program state and variables described by Back and von Wright in [15] which allows us to handle procedures in a very simple and elegant way.

Bibliography

[1] S. Agerholm. A HOL Basis for Reasoning about Functional Programs. PhD Thesis, University of Aarhus, BRICS Department of Computer Science, 1994. BRICS Report Series RS-94-44.

[2] F. Andersen. A Theorem Prover for UNITY in Higher Order Logic. PhD Thesis, Technical University of Denmark, Lyngby.

[3] P.B. Andrews. An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof. Academic Press, 1986.

[4] R.J.R. Back. On the correctness of refinement in program development. PhD thesis Report A-1978-4, Department of Computer Science, University of Helsinki, 1978.

[5] R.J.R. Back. Procedural abstraction in the refinement calculus. Technical Report 55, Department of Computer Science, Åbo Akademi, 1987.

[6] R.J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[7] R.J.R. Back. Refinement Calculus, part II: Parallel and reactive programs. *REX Workshop for Refinement of Distributed Systems*, *Lecture Notes in Computer Science* 430, Nijmegen, the Netherlands, 1989. Springer–Verlag.

[8] R.J.R. Back. Refinement diagrams. In J.M. Morris and R.C.Show(Eds.), Proceedings of *Fourth BCS Refinement Workshop*. Springer–Verlag, 1991.

[9] R. Back, J.Grundy, and J. von Wright. Structured Calculational Proof. Technical Report No.65, Turku Centre for Computer Science (TUCS), 1996.

[10] R.J.R. Back and K. Sere. Stepwise refinement of action systems. *In Mathematics of Program Construction*, *Lecture Notes in Computer Science* 375, pg. 17–30, Groningem, the Netherlands, June 1989. Springer–Verlag.

[11] R.J.R. Back and J. von Wright. Duality in specification languages: a lattice-theoretical approach. *Acta Informatica*, 27:583–625, 1990.

[12] R.J.R. Back and J. von Wright. Refinement concepts formalised in higher-order logic. *Formal Aspects of Computing*, 2:247–272, 1990.

[13] R. Back and J. von Wright. Statement inversion and strongest postcondition. *Science of Computer Programming*, 20:223–251, 1993.

[14] R. Back and J. von Wright. Programs on Product Spaces. Technical Report No.143, Turku Centre for Computer Science (TUCS), November 1997.

[15] R. Back and J. von Wright. Refinement Calculus: A Systematic Introduction. Springer–Verlag, 1998.

[16] R. Back and J. von Wright. Products in the Refinement Calculus. Technical Report No.235, Turku Centre for Computer Science (TUCS), November 1999.

[17] R. Back and J. von Wright. Encoding, Decoding and Data Refinement. Technical Report No.236, Turku Centre for Computer Science (TUCS), November 1999.

[18] J. de Bakker Mathematical Theory of Program Correctness. Prentice–Hall International, 1980.

[19] A. Bijlsma, P.A. Matthews and J.G. Wiltink. Equivalence of the Gries and Martin Proof Rules for Procedure Calls. *Acta Informatica*, 23:357–360, 1986.

[20] A. Bijlsma, P.A. Matthews and J.G. Wiltink. A Sharp Proof Rule for Procedures in wp Semantics. *Acta Informatica*, 26:409–420, 1989.

[21] R. Boulton, A. Gordon, M. Gordon, J. Harrison, and J. Herbert. Experience with embedding hardware description languages. In Proceedings of the IFIP International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, vol. A10 of IFIP Transactions, pg. 129–156. North Holland/Elsevier, June 1992.

[22] R.S. Boyer and J.S. Moore. A verification condition generator for FORTRAN. In R.S. Boyer and J.S. Moore (Eds.), *The Correctness Problem in Computer Science*. Academic Press, London, 1981.

[23] R.M. Burstall and J. Darlington. Some transformations for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[24] M.J. Butler, J. Grundy, T. Långbacka, R. Rukšėnas, and J. von Wright. The Refinement Calculator: Proof Support for Program Refinement. Proceedings of FMP'97 – Formal Methods Pacific, Wellington, New Zealand, July 1997. Springer-Verlag, Singapore, Discrete Mathematics and Theoretical Computer Science Series, pg. 40–61, 1997.

[25] M.J. Butler, T. Långbacka. Program Derivation Using the Refinement Calculator. In J. von Wright, J. Grundy, J. Harrison (Eds.), Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics, August 1996, Turku, Finland. Springer-Verlag, Lecture Notes in Computer Science, number 1125, pg. 93–108, 1996.

[26] M.J. Butler, T. Långbacka, L. Laibinis, R. Rukšėnas, and J. von Wright. Refinement Calculator Tutorial and Manual. Åbo Akademi, Department of Computer Science.

[27] M.Büchi and W. Weck. A Plea for Grey-Box Components. Workshop on Foundations of Component-Based Systems, Zürich, September 1997.

[28] A. Camillieri. Mechanising CSP trace theory in Higher Order Logic. In *IEEE Transactions on Software Engineering*, 16(9):993–1004, 1990.

[29] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A program refinement tool. *Formal Aspects of Computing*, 10(2):97–124, 1998.

[30] 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.

[31] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[32] G. Collins. A Proof Tool for Reasoning about Functional Programs. In *Proc. 1996 International Workshop on Higher Order Logic Theorem Proving,* Lecture Notes in Computer Science 1125, pg. 109–124, Turku, Finland, August 1996. Springer–Verlag.

[33] O.-J. Dahl. Verifiable Programming. Prentice–Hall International, 1992.

[34] E.W. Dijkstra. Notes on structured programming. In *Structured Programming*, Academic Press, 1971.

[35] E.W. Dijkstra. A Discipline of Programming. Prentice–Hall International, 1976.

[36] K. Engelhardt and W.-P. de Roever. Simulation of Specification Statements in Hoare Logic. Proceedings of 21st International Symposium on Mathematical Foundations of Computer Science, *Lecture Notes in Computer Science*, 1113:324–335, Springer-Verlag, 1996.

[37] N. Francez. Program Verification. Addison-Wesley, 1992.

[38] P.H. Gardiner and C.C. Morgan. Data refinement by calculation. *Acta Informatica*, 27(6):481–503, 1990.

[39] P.H. Gardiner and C.C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87(1):143–162, 1991.

[40] S.L. Gerhart. Correctness preserving program transformations. In Proceedings of *2nd ACM Conference on Principles of Programming Languages*, pg. 54–66, 1975.

[41] M.J.C. Gordon. HOL: A proof generating system for higher-order logic. In *VLSI Specification, Verification and Synthesis*. Kluwer Academic Publishers, pg. 73–128, 1988.

[42] M.J.C. Gordon. Mechanising programming logics in higher order logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*, 387–439. Springer–Verlag, 1989.

[43] M.J.C. Gordon and T.F. Melham. Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic. Cambridge University Press, Cambridge, 1993.

[44] M.J.C. Gordon, R. Milner and C. Wadsworth. Edinburgh LCF: A mechanised logic of computation. In *Lecture Notes in Computer Science* 78. Springer–Verlag, 1979.

[45] D. Gray. A pedagogical verification condition generator. *The Computer Journal*, 30(3):239–248, June 1987.

[46] D. Gries. The Science of Programming. Springer–Verlag, 1981.

[47] D. Gries and G. Levin. Assignment and Procedure Call Proof Rules. *ACM Transactions on Programming Language Systems*, 2:564–579, 1981.

[48] J. Grundy. Window Inference in the HOL System. In *Proc. of the Int. Workshop on the HOL Theorem Proving System and Its Applications*, pg. 177-189, August 1991. IEEE Computer Society Press.

[49] J. Grundy. A Method of Program Refinement. PhD Thesis, University of Cambridge.

[50] E. Gunter. Doing algebra in higher order logic. In the HOL system documentation, Cambridge, 1990.

[51] E. Gunter. The Implementation and Use of Abstract Theories in HOL In *Proc. of the Third HOL Users Meeting*, Aarhus, Denmark, October 1990. Technical Report DAIMI PB – 340, 1990.

[52] E.C. Hehner. **do** Considered **od**: A Contribution to the Programming Calculus. *ACM Acta informatica*, 11:287–304, 1979.

[53] E.C. Hehner. A practical theory of programming. *Science of Computer Programming*, 14(2):133–158, 1990.

[54] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, Vol.12, 10:576–583, 1969.

[55] C.A.R. Hoare. Procedures and parameters: An axiomatic approach. *Lecture Notes in Mathematics*, 188:102–116, Springer-Verlag, 1971.

[56] C.A.R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1(4):271–281, 1972.

[57] P. Homeier. Trustworthy Tools for Trustworthy Programs: A Mechanically Verified Verification Condition Generator for The Total Correctness of Procedures Ph.D. Dissertation, UCLA Computer Science Department, 1995.

[58] P. Homeier and D.F. Martin. Mechanical Verification of Total Correctness through Diversion Verification Conditions In *Proc. 1998 International Conference on Theorem Proving in Higher Order Logics,* Lecture Notes in Computer Science 1479, Canberra, Australia, September 1998. Springer–Verlag.

[59] S. Igarashi, R.L. London, and D.C. Luckham. Automatic program verification: A logical basis ans its implementation. *Acta Informatica*, 4:145–182, 1975.

[60] F. Kammüller. Modular Structures as Dependent Types in Isabelle. In T. Altenkirch, W. Naraschewski, and B. Reus (Eds.), Selected Papers of International Workshop on Types for Proofs and Programs (TYPES '98), Kloster Irsee, Germany, March 27–31, 1998. Springer-Verlag, Lecture Notes in Computer Science 1657, 1998.

[61] F. Kammüller and L. C. Paulson. A Formal Proof of Sylow's First Theorem – An Experiment in Abstract Algebra with Isabelle HOL. *Journal of Automated Reasoning*, Vol. 23, November 1999.

[62] L. Laibinis. Using Lattice Theory in Higher Order Logic. In J. von Wright, J. Grundy, J. Harrison (Eds.), Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics, August 1996, Turku, Finland. Springer-Verlag, Lecture Notes in Computer Science, number 1125, pg. 315–330, 1996.

[63] L. Laibinis. Mechanising Procedures in HOL. TUCS technical report No.253, 1999.

[64] L. Laibinis, L. Mikhajlov, E.Sekerinski. Developing Components in the Presence of Re-entrance. Proceedings of World Congress on Formal Methods'99 (FM'99), September 1999, Toulouse, France. Springer-Verlag, Lecture Notes in Computer Science, number 1709, pg. 1301–1320, 1999.

[65] L. Laibinis, J. von Wright. Context handling in the Refinement Calculus framework. In U. M. Haveraaen, O. Owe (Eds.), Proceedings of the 8th Nordic Workshop on Programming Theory, pg. 139–148, December 1996, Oslo, Norway. Research Report 248, Department of Informatics, University of Oslo.

[66] L. Laibinis, J. von Wright. What's in a Specification? In J.Grundy, M.Schwenke, T.Vickers (Eds.), Proceedings of International Refinement Workshop & Formal Methods Pacific'98, September 1998, Canberra, Australia. Springer-Verlag, pg. 180–192, 1998.

[67] L. Laibinis, J. von Wright. Functional Procedures in Higher-Order Logic. TUCS technical report No.252, 1999.

[68] T. Långbacka. A HOL Formalisation of the Temporal Logic of Actions. In T. F. Melham, J. Camillieri (Eds.), Proceedings of the 7th International Workshop on Theorem Proving in Higher Order Logics, September 1994, Valletta, Malta. Springer-Verlag, Lecture Notes in Computer Science, number 859, pg. 332–345, 1994.

[69] T. Långbacka. An Interactive Environment Supporting the Development of Formally Correct Programs. PhD Thesis, Turku Centre for Computer Science, 1997.

[70] T. Långbacka, R. Rukšėnas and J. von Wright. TkWinHOL: A tool for doing window inference in HOL. In *LNCS* 971, 245–260. Springer–Verlag, 1995.

[71] T. Långbacka and J. von Wright. Refining Reactive Systems in HOL Using Action Systems. Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics, August 1997, Murray Hill, NJ, USA. Springer-Verlag, Lecture Notes in Computer Science, number 1275, pg. 183–197, 1997.

[72] Z. Luo. A Higher-order Calculus and Theory Abstraction. *Information and Computation*, 90(1), 1990.

[73] D. B. MacQueen. Using Dependent Types to Express Modular Structures. In Proceedings of 13th ACM Symposium on Principles of Programming Languages. ACM Press, 1986.

[74] B. Mahony. Expression Refinement in Higher Order Logic. In Proceedings of 1998 International Refinement Workshop and Formal Methods Pacific, pg. 230–249, Discrete Mathematics and Theoretical Computer Science, Springer–Verlag, 1998.

[75] A.J. Martin. A General Proof Rule for Procedures in Predicate Transformer Semantics. *Acta Informatica*, 20:301–313, 1983.

[76] T.F. Melham. A mechanised theory of the π-calculus in HOL. Technical Report 244, University of Cambridge Computer Laboratory, January 1992.

[77] A. Mikhajlova and E. Sekerinski. Class Refinement and Interface Refinement in Object-Oriented Programs. Proceedings of the 4th International Formal Methods Europe Symposium (FME'97), J. Fitzgerald, C.B. Jones, and P. Lucas (Eds.), Lecture Notes for Computer Science 1313, Springer–Verlag, p.82–101, 1997.

[78] R. Milner, M.J.C. Gordon, C. Wadsworth. Edinburgh LCF: A mechanised logic of computation. Springer-Verlag, Lecture Notes in Computer Science 78, 1979.

[79] A. Mohamed. The theory of the pi-calculus in HOL. PhD Dissertation, Henri Poincare University. July 1996.

[80] C.C. Morgan. Procedures, parameters and abstraction: separate concerns. *Science of Computer Programming*, 11:17–27, 1988.

[81] C.C. Morgan. The Specification Statement. *ACM Transactions on Programming Languages and Systems*, 10:403–419, 1988.

[82] C.C. Morgan. Data refinement by miracles. *Information Processing Letters*, 26:243–246, 1988.

[83] C.C. Morgan. Types and invariants in the refinement calculus. In Mathematics of Program Construction, *Lecture Notes in Computer Science* 375, pg. 363–378, The Netherlands, June 1989. Springer–Verlag.

[84] C.C. Morgan. Programming from Specifications. Prentice Hall, 1994, second edition.

[85] C.C. Morgan, A.K. McIver, and K. Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, 1996.

[86] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

[87] J.M. Morris. Non-deterministic expressions and predicate transformers. *Information Processing Letters*, 61(5):241–246, 1997.

[88] W. Narashevski and M. Wenzel. Object-oriented Verification based on Record Subtyping in Higher-Order Logic. In Proceedings of 11th International Conference on Theorem Proving in Higher Order Logics, Canberra, Australia, September 1998. Springer-Verlag, *Lecture Notes in Computer Science* 1479, pg. 349–368, 1998.

[89] D. A. Naumann. Predicate transformer semantics of Oberon-like language. In Olderog (Ed.), *Programming Concepts, Methods and Calculi*, 460–480, San Miniato, Italy, 1994. IFIP.

[90] R. Nickson and I. Hayes. Supporting contexts in program refinement. *Science of Computer Programming*, 29(3):279–302, 1997.

[91] E.R. Olderog. On the Notion of Expressiveness and the Rule of Adaptation. *Theoretical Computer Science*, 24:337–347, 1983.

[92] F. Regensburger. HOLCF: Higher Order Logic of Computable Functions. In *LNCS* 971, 293–307. Springer–Verlag, 1995.

[93] M.Reiser and N.Wirth. Programming in Oberon: Steps Beyond Pascal and Modula. Addison-Wesley, 1992.

[94] P.J. Robinson and J. Staples. Formalising the hierarchical structure of practical mathematical reasoning. *Logic and Computation*, 1:47–61, 1993.

[95] R. Rukšėnas and J. von Wright. A tool for data refinement. In *LNCS* 1479, 423–442. Springer–Verlag, 1998.

[96] M. Schwenke and K. Robinson. What If? In Second Australian Refinement Workshop, 1992.

[97] K. Slind. Function Definition in Higher-Order Logic. In *Proc. 1996 International Workshop on Higher Order Logic Theorem Proving,* Lecture Notes in Computer Science 1125, Turku, Finland, August 1996. Springer–Verlag.

[98] S. Sokolovski. Total correctness of procedures. In *Proceedings of 6th Symposium on the Mathematical Foundations of Computer Science*, *Lecture Notes in Computer Science* 53, 475–483. Springer–Verlag, 1977.

[99] M. Staples. A Mechanised Theory of Refinement. Ph.D. Thesis, Cambridge University, 1999.

[100] A. Tarski. A lattice theoretical fixed point theorem and its applications. *Pacific J. Mathemetics*, 5:285–309, 1955.

[101] M. Utting and K. Robinson. Modular reasoning in an object-oriented refinement calculus. In R.Bird, C.C.Morgan, and J.Woodcock (Eds.), *Mathematics of Program Construction*, *Lecture Notes in Computer Science* 669, 334–367. Springer–Verlag, 1993.

[102] M. Utting and K. Whitwell. Ergo user manual. Technical Report 93-19, Software Verification Research Centre, The University of Queensland, February 1994.

[103] J. Welsh and J.Elder. Introduction to Pascal. Prentice–Hall International, 1979.

[104] J. Welsh and J.Elder. Introduction to Modula-2. Prentice–Hall International, 1987.

[105] P.J. Windley. Abstract Theories in HOL. In *Proc. of the Int. Workshop on the HOL Theorem Proving System and Its Applications*, September 1992. IFIP Transactions A-20, 197–210.

[106] N. Wirth. Program development by stepwise refinement. *Communications of ACM*, 14:221–227, 1971.

[107] J. von Wright. Doing Lattice Theory in Higher Order Logic. In Technical Report 136, Reports on Computer Science and Mathematics, Series A. Åbo Akademi, Turku, 1992.

[108] J. von Wright. Program refinement by theorem prover. In *Proc. 6th Refinement Workshop*, London, January 1994. Springer–Verlag.

[109] J. von Wright. Verifying Modular Programs in HOL. Technical Report No.324, Computer Laboratory of University of Cambridge, 1994.

Turku Centre for Computer Science

TUCS Dissertations

1. **Marjo Lipponen**, On Primitive Solutions of the Post Correspondence Problem
2. **Timo Käkölä**, Dual Information Systems in Hyperknowledge Organizations
3. **Ville Leppänen**, Studies on the Realization of PRAM
4. **Cunsheng Ding**, Cryptographic Counter Generators
5. **Sami Viitanen**, Some New Global Optimization Algorithms
6. **Tapio Salakoski**, Representative Classification of Protein Structures
7. **Thomas Långbacka**, An Interactive Environment Supporting the Development of Formally Correct Programs
8. **Thomas Finne**, A Decision Support System for Improving Information Security
9. **Valeria Mihalache**, Cooperation, Communication, Control. Investigations on Grammar Systems
10. **Marina Waldén**, Formal Reasoning About Distributed Algorithms
11. **Tero Laihonen**, Estimates on the Covering Radius When the Dual Distance is Known
12. **Lucian Ilie**, Decision Problems on Orders of Words
13. **Jukkapekka Hekanaho**, An Evolutionary Approach to Concept Learning
14. **Jouni Järvinen**, Knowledge Representation and Rough Sets
15. **Tomi Pasanen**, In-Place Algorithms for Sorting Problems
16. **Mika Johnsson**, Operational and Tactical Level Optimization in Printed Circuit Board Assembly
17. **Mats Aspnäs**, Multiprocessor Architecture and Programming: The Hathi-2 System
18. **Anna Mikhajlova**, Ensuring Correctness of Object and Component Systems
19. **Vesa Torvinen**, Construction and Evaluation of the Labour Game Method
20. **Jorma Boberg**, Cluster Analysis. A Mathematical Approach with Applications to Protein Structures
21. **Leonid Mikhajlov**, Software Reuse Mechanisms and Techniques: Safety Versus Flexibility
22. **Timo Kaukoranta**, Iterative and Hierarchical Methods for Codebook Generation in Vector Quantization
23. **Gábor Magyar**, On Solution Approaches for Some Industrially Motivated Combinatorial Optimization Problems
24. **Linas Laibinis**, Mechanised Formal Reasoning About Modular Programs

Turku Centre for Computer Science
emmink isenkatu
Turku
in an

http://www.tucs.fi

Uni ersit of Turku
• Department of at ematica Sciences

o ka emi Uni ersit
• Department of Computer Science
• nstitute for ance ana ement S stems esearc

Turku Sc oo of conomics an usiness ministration
• nstitute of nformation S stems Science