Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs

Thomas Kleymann (formerly Schreiber)

Doctor of Philosophy
University of Edinburgh
1998

Abstract

Investigating soundness and completeness of verification calculi for imperative programming languages is a challenging task. Many incorrect results have been published in the past. We take advantage of the computer-aided proof tool LEGO to interactively establish soundness and completeness of both Hoare Logic and the operation decomposition rules of the Vienna Development Method (VDM) with respect to operational semantics. We deal with parameterless recursive procedures and local variables in the context of total correctness. As a case study, we use LEGO to verify the correctness of Quicksort in Hoare Logic.

As our main contribution, we illuminate the rôle of auxiliary variables in Hoare Logic. They are required to relate the value of program variables in the final state with the value of program variables in the initial state. In our formalisation, we reflect their purpose by interpreting assertions as relations on states and a domain of auxiliary variables. Furthermore, we propose a new structural rule for adjusting auxiliary variables when strengthening preconditions and weakening postconditions. This rule is stronger than all previously suggested structural rules, including rules of adaptation.

With the new treatment, we are able to show that, contrary to common belief, Hoare Logic subsumes VDM in that every derivation in VDM can be naturally embedded in Hoare Logic. Moreover, we establish completeness results uniformly as corollaries of Most General Formula theorems which remove the need to reason about arbitrary assertions.

Acknowledgements

Working towards a thesis is much more demanding than I had initially anticipated. I am grateful to my colleagues and friends who have succeeded in providing a pleasant environment to work and live in. Special thanks go to my wife Beate for her patience and support.

I would like to thank my supervisors Rod Burstall and Paul Jackson. Rod's advice has been invaluable in many respects. I have been very much enjoying discussions with him and have always left his office in good spirit. Paul has very much influenced the presentation of the thesis. He has also done a good job in pressuring me to finish.

Discussions with Healf Goguen, Martin Hofmann, Zhaohui Luo and James McKinna have been very stimulating. In fact, a remark by Healf has inspired me to investigate soundness and completeness in the first place. I would also like to thank the departmental computing support staff for providing a friendly and effective service.

I am in debt to Peter Otto for his generous support in the initial stage of this thesis. I would also like to acknowledge the financial support of the Deutsche Forschungsgemeinschaft (Sonderforschungsbereich 182, Mutliprozessor- und Netzwerkkonfigurationen), EPSRC, the British Council (ARC Project Co-Development of Object-Oriented Programs in LEGO) and the European Commission (Marie Curie Fellowship).

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text. Some of this work has been published previously (Schreiber 1997).

(*Thomas Kleymann*)

Contents

Chapter 1 Introduction **1**

1.1 Soundness and Completeness 2
1.2 Main Outcome 2
1.3 The Logical Framework 4
1.3.1 Dependent Function Types 4
1.3.2 Inductively Defined Data Types 5
1.3.3 Inductively Defined Relations 5
1.4 Notation 7
1.4.1 Implicit Quantification 7
1.5 Overview of Thesis 8

Chapter 2 Fundamental Aspects of Hoare Logic and VDM **9**

2.1 Deep and Shallow Embedding Techniques 10
2.2 Program Variables 10
2.3 The State Space 11
2.4 Interpretations 12
2.5 Expressions 13
2.6 Imperative Programs 15
2.6.1 Assignment and Arrays 15
2.7 Structural Operational Semantics 16
2.8 Low-Level Verification 17
2.9 The Vienna Development Method 20
2.9.1 Mechanisation 22
2.9.2 High-Level Verification 23
2.9.3 Soundness 25
2.9.4 Completeness 26
2.10 Hoare Logic 29
2.10.1 Verification Calculus 30
2.10.2 Soundness and Completeness 32

2.10.3 Auxiliary Variables 34
2.11 Hoare Logic subsumes VDM 42
2.12 Completeness: What does it really mean? 46
2.13 Related Work 50
2.13.1 A Rigorous Treatment of Auxiliary Variables 50
2.13.2 Machine-Checked Developments 51

Chapter 3 Recursive Procedures and Local Program Variables 57
3.1 Parameterless Recursive Procedures 58
3.1.1 Declaring and Invoking Procedures 58
3.1.2 Structural Operational Semantics 59
3.1.3 Hoare Logic with Contexts 59
3.1.4 Total Correctness 62
3.1.5 Soundness 65
3.1.6 Completeness 65
3.1.7 An Example Derivation 69
3.1.8 Rules of Adaptation 71
3.1.9 Mutually Recursive Procedures 73
3.1.10 The Vienna Development Method 75
3.2 Local Program Variables 75
3.2.1 Initialisation 77
3.2.2 States with Dynamic Types 78
3.2.3 Programs with Dynamic Sorts 80
3.2.4 Hoare Logic 81
3.2.5 The Vienna Development Method 84
3.3 Recursive Procedures and Static Binding 85
3.3.1 Static versus Dynamic Binding 87
3.3.2 States and Environments 88
3.3.3 Expressions 89
3.3.4 Assertions 90
3.3.5 The Programming Language 91
3.3.6 Structural Operational Semantics 91
3.3.7 Hoare Logic 98
3.3.8 The Vienna Development Method 110
3.4 Parameter Passing 110

Chapter 4 Case Study: Quicksort **113**
4.1 A Theory of Sorted Arrays . . . 114
4.1.1 Ordered Lists . . . 114
4.1.2 Subarrays . . . 115
4.1.3 High-Level Correctness Argument . . . 116
4.2 Specification of Quicksort . . . 117
4.2.1 Mechanisation . . . 119
4.3 A Guided Tour of the Verification Proof . . . 121
4.3.1 Verification of Quicksort Skeleton . . . 122
4.3.2 Verification of the Prepare fragment . . . 125
4.4 Conclusions . . . 128

Chapter 5 Conclusions and Future Work **130**
5.1 Syntax of Assertions . . . 130
5.2 The Rôle of Auxiliary Variables in Hoare Logic . . . 131
5.3 Future Work . . . 131

Appendix A Some Remarks on Working with the LEGO System **133**
A.1 Notation . . . 133
A.2 Recursion . . . 134
A.3 Well-Founded Relations . . . 135
A.4 Equality . . . 135
A.5 Updating Dependent Functions . . . 137

Appendix B Outline of LEGO Scripts **140**
B.1 Simple Imperative Programs . . . 140
B.2 Recursive Procedures . . . 156
B.3 Local Variables . . . 171
B.4 Recursive Procedures and Static Binding . . . 182
B.5 Quicksort . . . 215
B.6 Additional Background Theory . . . 235

Bibliography **241**

Index **249**

Chapter 1

Introduction

The correctness of software is an important issue in software engineering. In particular, in safety-critical applications e.g., controlling chemical plants, it must be ensured that the implementation satisfies the required specification. Often, it is insufficient to validate the correctness of a complex software system by relying merely on test data. Delivering the program together with a formal proof of its correctness with respect to some specification guarantees a correct program, whereas tests can only reduce errors.

In 1969, Hoare introduced a formal system, now referred to as Hoare Logic, relating imperative programs with two assertions, both first-order logical formulae. The purpose of Hoare Logic was twofold. On the one hand, it axiomatised the semantics of the programming language. Occasionally, one therefore finds *axiomatic semantics* as a synonym for Hoare Logic. On the other hand, objects of the formal system correspond to correctness formulae describing properties of programs. Thus, derivations in Hoare Logic are verification proofs. Nowadays, taking Hoare Logic rules as axiomatic definitions of programming languages is no longer seen as being adequate. Instead, one investigates soundness and completeness with respect to a simpler semantics, such as operational semantics.

Much attention was devoted to this line of research in the 70s and early 80s. However, the field appeared to run into insurmountable problems. Incompleteness results seemed to indicate that it would be impossible to design *any* verification calculus in the style of Hoare Logic as soon as one combines more complex language features. Moreover, conducting soundness and completeness proofs for more interesting programming languages turned out to be highly error-prone and laborious. Many of the proposed verification calculi were unsound or incomplete *despite* being backed up by "proofs" to the contrary.

For example, Sokołowski's (1977) presentation for recursive procedures is incomplete. Apt's (1981) extension yields a complete but, unfortunately, unsound calculus. The first sound and complete verification calculus in the style of Hoare Logic for re-

cursive procedures and total correctness was published in 1990 by America & de Boer.

1.1 Soundness and Completeness

Soundness is essential. If a system is unsound, deriving a property for a particular program within the formal system does not guarantee that the program actually fulfils the property.

In an incomplete formal system, one may only verify a strict subset of all true formulae. A naïve definition of completeness is bound to fail in the context of verification calculi. On the one hand, if the chosen underlying logical language is too weak, e.g., pure first-order logic together with the boolean constants **false** and **true**, some intermediate assertions cannot be expressed. Hence, derivations cannot be completed. On the other hand, if the logical language is too strong, e.g. Peano Arithmetic, it itself is already incomplete and the verification calculus inherits incompleteness.

To circumvent this dilemma, Cook (1978) proposed that instead one should investigate *relative completeness*. One merely considers sufficiently expressive underlying logical languages. Furthermore, the formal system is augmented by a theory[1] of first-order logic. Whenever a rule is protected by a first-order logic side-condition, the rule is already applicable when the side-condition is an element of the theory, and not only when the side-condition is itself *derivable*.

In practice, achieving relative completeness of verification calculi is highly desirable. In logic, finding valid formulae which cannot be derived is often somewhat esoteric. A different story has to be told for the notion of relative completeness in verification calculi e.g., in Sokołowski's (1977) calculus, it is very difficult to come up with any non-contrived correctness formula of a recursive procedure which can be derived!

1.2 Main Outcome

We have produced machine-checked soundness and completeness proofs for Hoare Logic and the operation decomposition rules of the Vienna Development Method (VDM). Our imperative programming language includes (parameterless) recursive procedures and local variables. We consider static binding and total correctness. Previously, machine-checked completeness results have only been available for simple imperative programs (and partial correctness).

[1] the set of all valid as opposed to derivable formulae

Our message to the designers and researchers of verification calculi is that conducting computer-aided soundness and completeness proofs is both a feasible and profitable task. We are confident that when investigating further language features, the overhead caused by mechanisation is tolerable. Today's modern proof tools are sufficiently advanced. Besides LEGO, one could for example, also resort to the systems Coq (1998), HOL (1998), Isabelle (Paulson & Nipkow 1998) or PVS (Rushby 1998).

Conducting formal proofs at a level that they can be checked by a machine requires that all aspects and all details must be thoroughly investigated. To be a feasible task, it is often necessary to first simplify the concepts themselves, before establishing the desired properties. Hence, in addition to more confidence in the mechanised proofs, one can often contribute by simplifying or illuminating some aspects of the area investigated.

Our fundamental contribution has been to highlight the rôle of auxiliary variables in Hoare Logic. Usually, assertions are interpreted as *predicates on states* where free variables denote the value of program variables in a specific state. Variables for which no counterpart appears as a program variable in the program under consideration then take on the rôle of auxiliary variables. They are required to relate the value of program variables in *different* states.

Our view of assertions emphasises the pragmatic importance of auxiliary variables. We have followed a proposal by Apt & Meertens (1980) to consider assertions as *relations on states and auxiliary variables*. Furthermore, we stipulate a new structural rule to adjust auxiliary variables when strengthening preconditions and weakening postconditions. This rule is stronger than all previously suggested structural rules, including Hoare's (1969) consequence rule and rules of adaptation. As a direct consequence of the new treatment of auxiliary variables,

- we were able to show that Sokołowski's (1977) calculus is sound and complete if one replaces Hoare's rule of consequence with ours. In particular, none of the other structural rules introduced by Apt (1981) are required.

- We have clarified the relationship between Hoare Logic and its variant VDM. We were able to show that, contrary to common belief, VDM is more restrictive than Hoare Logic in that every derivation in VDM can be naturally embedded in Hoare Logic. Intuitively, in Hoare Logic, auxiliary variables record the value of variables at an *arbitrary* reference point. In VDM, the reference point corresponds to the state of the precondition and must therefore be constantly adjusted during a derivation.

- Completeness is established by induction on the structure of programs. How-

ever, in the presence of recursive procedures, a direct proof does not go through, because the induction hypothesis is then too weak. Instead one first establishes, also by induction, that a particular correctness formula, the Most General Formula (MGF), can be derived. This obliterates the problems of having to cater for arbitrary assertions. Completeness follows as a corollary by appealing to structural rules.

This technique had previously only been applied to Hoare Logic dealing with recursive procedures. We have conducted *all* completeness proofs with the help of the MGF property, because, with our new rule of consequence, completeness follows *directly* from the MGF theorem.

The new approach has already been taught to undergraduate students as part of Hofmann's (1997) course on semantics and verification at the University of Marburg.

1.3 The Logical Framework

We have employed the computer-aided proof tool LEGO (1998) to investigate soundness and completeness for Hoare Logic and the operation decomposition rules of VDM. LEGO is a proof assistant based on type theory which provides a higher-order (intuitionistic) logic. The system supports dependent types, inductively defined data types and inductively defined relations. The underlying type theory is UTT (Luo 1994). It is not necessary to be familiar with any proof assistant or type theory in order to read this thesis.

In our formalisation, programs may contain variables with *arbitrary* sorts. This leads to a more complex model of the state space. We have found it helpful to employ dependent function types in order to model the state space.

We define the syntax of programs as an inductive data type. We capture operational semantics and the notion of derivability of correctness formulae in a verification calculus as an inductive relation. This allows us to conduct (completeness) proofs by induction on the structure of programs and (soundness) proofs by induction on the derivation of correctness formulae.

We now introduce dependent function types, inductively defined data types and inductively defined relations in more detail.

1.3.1 Dependent Function Types

In type theory, every term is annotated by its type. Only well-typed terms are well-formed e.g., $f : T \to U$ is a function which, given $x : T$ yields $f(x) : U$. We may

only apply f to elements of type T. We will however omit such annotations in our presentation when the typing discipline is evident from the context.

UTT also provides dependent types in which types themselves are functions sort : $T \to \texttt{Type}$. One writes $\prod x : T \cdot \text{sort}(x)$ rather than $T \to \text{sort}$ to denote a dependent function space. Let $f : \prod x : T \cdot \text{sort}(x)$. Given an argument $x : T$, the function f returns a value of type $\text{sort}(x)$. Dependent function types are particularly useful when one wants to specify a function in which the type of some arguments depends on the value of previous arguments.

Example 1.1 (Scalar Product on Reals) *With dependent functions, one can implement a single function*

$$f(n)(\vec{x},\vec{y}) = \sum_{1 \le i \le n} x_i \times y_i$$

to compute the scalar product of two vectors $\vec{x},\vec{y} : \mathbb{R}^n$ *of* arbitrary but identical *dimension* $n : \texttt{nat}$. *Its type is*

$$f : \prod n : \texttt{nat} \cdot (\mathbb{R}^n \times \mathbb{R}^n) \to \mathbb{R} \ .$$

1.3.2 Inductively Defined Data Types

Consider the BNF grammar

$$S ::= x := t \mid \textbf{while } b \textbf{ do } S$$

of a (very) simple imperative programming language. This is an inductive definition. We may prove properties about arbitrary programs by considering its two constructors for assignments and loops. Formally, we may appeal to the induction principle

$$\frac{\forall x \cdot \forall t \cdot \Phi(x := t) \qquad \forall b \cdot \forall S \cdot \Phi(S) \Rightarrow \Phi(\textbf{while } b \textbf{ do } S)}{\forall S \cdot \Phi(S)}$$

for an arbitrary predicate Φ on programs.

1.3.3 Inductively Defined Relations

Let $\sigma \xrightarrow{\quad S \quad} \tau$ be a new relation to describe the operational semantics of the above programming language. Intuitively, it denotes that the program S when invoked

in the state σ will terminate in the state τ. We define it by the least relation satisfying

$$\sigma \xrightarrow{x:=t} \sigma[x \mapsto \mathrm{eval}(\sigma)(t)] \tag{1.1}$$

$$\sigma \xrightarrow{\textbf{while } b \textbf{ do } S} \sigma \quad \text{provided } \mathrm{eval}(\sigma)(b) = \textbf{false}\ . \tag{1.2}$$

$$\frac{\sigma \xrightarrow{S} \eta \quad \eta \xrightarrow{\textbf{while } b \textbf{ do } S} \tau}{\sigma \xrightarrow{\textbf{while } b \textbf{ do } S} \tau} \quad \text{provided } \mathrm{eval}(\sigma)(b) = \textbf{true}\ . \tag{1.3}$$

This is an example of an inductively defined relation. Given a proof obligation of the form

$$\forall \sigma \cdot \forall S \cdot \forall \tau \cdot \sigma \xrightarrow{S} \tau \Rightarrow \Phi(\sigma, S, \tau) \tag{1.4}$$

we may pursue induction by considering all (three) possible derivations[2]. Formally, the induction principle is

$$\frac{\begin{array}{c}\forall \sigma \cdot \forall x \cdot \forall t \cdot \Phi\big(\sigma, x:=t, \sigma[x \mapsto \mathrm{eval}(\sigma)(t)]\big) \\ \forall \sigma \cdot \forall b \cdot \forall S \cdot \mathrm{eval}(\sigma)(b) = \textbf{false} \Rightarrow \Phi(\sigma, \textbf{while } b \textbf{ do } S, \sigma) \\ \forall \sigma \cdot \forall b \cdot \forall S \cdot \forall \eta, \tau \cdot \big(\mathrm{eval}(\sigma)(b) = \textbf{true} \wedge \\ \quad \sigma \xrightarrow{S} \eta \wedge \eta \xrightarrow{\textbf{while } b \textbf{ do } S} \tau \wedge \Phi(\sigma, S, \eta) \wedge \Phi(\eta, \textbf{while } b \textbf{ do } S, \tau)\big) \Rightarrow \\ \qquad\qquad \Phi(\sigma, \textbf{while } b \textbf{ do } S, \tau)\end{array}}{\forall \sigma \cdot \forall S \cdot \forall \tau \cdot \sigma \xrightarrow{S} \tau \Rightarrow \Phi(\sigma, S, \tau)} \tag{1.5}$$

Computer-aided tools are very helpful in formally dealing with such induction principles. The LEGO system generates induction principles automatically from the set of defining constructors, such as (1.1)–(1.3). It also provides an `Induction` tactic to bring the proof obligation into the shape of (1.4) before breaking it down into the cases according to the induction principle.

In practice, inversion is an important proof technique for dealing with inductively defined relations. Consider the proof of determinism. We want to show

$$\forall \sigma \cdot \forall S \cdot \forall \tau \cdot \sigma \xrightarrow{S} \tau \Rightarrow \forall \eta \cdot \sigma \xrightarrow{S} \eta \Rightarrow \tau = \eta\ .$$

The `Induction` tactic splits the proof obligation into the three cases corresponding to the induction principle (1.5). We only consider the base case where we have to establish

$$\forall \eta \cdot \sigma \xrightarrow{x:=t} \eta \Rightarrow \sigma[x \mapsto \mathrm{eval}(\sigma)(t)] = \eta\ .$$

[2] On paper, such proofs sometimes appeal to induction on the *length* of a derivation. In a rigorous formal proof, it is (technically) more elegant to inductively analyse the derivation itself.

Rather than general induction, it suffices to consider the last step of the derivation of

$$\sigma \xrightarrow{x := t} \eta \tag{1.6}$$

In other words, we want to *invert* the derivation. Intuitively, (1.6) can only be derived from the axiom (1.1). Hence, the state η must coincide with $\sigma[x \mapsto \mathrm{eval}(\sigma)(t)]$. LEGO provides state of the art support for inverting inductively defined relations (McBride 1998).

1.4 Notation

We refrain from exposing the reader to the (rather unreadable) LEGO scripts in the main body of this thesis. Instead, we adopt a more traditional informal presentation with more flexible notation. We include sufficient details to facilitate encoding our results with different proof tools. We have decorated machine-checked developments with the symbol 🗎. An outline of the LEGO scripts can be found in Appendix B. All scripts are also available on-line and can be accessed from Lego's (1998) World Wide Web page.

The reader is encouraged to compare definitions in the text with the actual LEGO scripts (Pollack 1998). Machine-checked developments are not exempt from mistakes e.g., Homeier & Martin (1996) erroneously claim

> "*The correctness of these VCG functions is established by proving the following theorems from the axioms and rules of inference of the axiomatic semantics*"

But the actual encodings in HOL reveal that correctness has instead been established by appealing to the definition of *operational* semantics.

1.4.1 Implicit Quantification

An important convention is that all free identifiers in a rule (or axiom) are implicitly universally quantified at the level of the rule. For example, the rule

$$\frac{\forall t : W \cdot \{p \wedge b \wedge u = t\}\ S\ \{p \wedge u < t\}}{\{p\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{p \wedge \neg b\}}$$

is merely a convenient short cut for

$$\forall p \cdot \forall u \cdot \forall b \cdot \forall S \cdot \frac{\forall t : W \cdot \{p \wedge b \wedge u = t\}\ S\ \{p \wedge u < t\}}{\{p\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{p \wedge \neg b\}}$$

where the order of the top-level universal quantifiers does not matter. In particular, scoping guarantees[3] that the variable t can not occur freely in p, u, b and S.

[3] The rule would be unsound otherwise.

1.5 Overview of Thesis

In **Chapter 2**, restricting our attention to simple imperative programs, we consider fundamental aspects of verification calculi such as Hoare Logic and VDM. We axiomatise the meaning of programs via a structural operational semantics. We then demonstrate that reasoning based directly on this semantic account is tedious. A more abstract approach leads naturally to the operation decomposition rules of VDM.

Turning our attention to Hoare Logic, we discuss the rôle of auxiliary variables in detail. Based on this analysis, we interpret assertions as relations on the state space and a domain of auxiliary variables. Moreover, we introduce a new structural rule to adjust auxiliary variables when strengthening preconditions and weakening postconditions.

We mechanically establish soundness and completeness for both Hoare Logic and VDM. We also show that Hoare Logic subsumes VDM and take a closer look at the definition of (relative) completeness. At the end of this chapter, we compare our approach to related work.

Chapter 2 exposes the main ideas of this thesis. In **Chapter 3**, we apply the techniques to more intricate settings. America & de Boer (1990) presented the first sound and complete formal system for recursive procedures dealing with termination in the style of Hoare Logic. We show that after replacing Hoare's (1969) rule with our new consequence rule, a third of their rules are not required; neither pragmatically, nor to achieve completeness. Furthermore, we extend our results to imperative programs supporting local variables (with static scoping) in the presence of recursive procedures.

Our meta-theoretical investigations of Hoare Logic and VDM are complemented by a case-study in **Chapter 4**. Focussing on the well-understood Quicksort algorithm, we hope to clarify how our new Hoare Logic rules may be used in practice.

In **Chapter 5**, we conclude and outline future work. All of our results have been developed using the LEGO proof tool. We comment on the feasibility of this project and on the strengths and weaknesses of the LEGO system.

Appendix A contains some further remarks on working with the LEGO system. In particular, we document how to encode a notion of well-founded relations and extensional equality. We also show how to update dependent functions and hence machine states; this requires quite sophisticated type theory. In **Appendix B**, we have collected an outline of the actual LEGO scripts.

Chapter 2

Fundamental Aspects of Hoare Logic and VDM

In this chapter, we introduce syntax and an operational semantics for a simple imperative programming language with assignments, conditionals and loops. We then present two verification calculi, VDM and Hoare Logic, and show that they are sound and complete with respect to the operational semantics. We consider total correctness i.e., besides input/output specifications, the verification calculi also deal with termination of programs.

The outline of the chapter is as follows. Having built up sufficient background theory, we consider the correctness proof of an imperative program realising the exponential function. This proof is based directly on the definition of the operational semantics. As this is rather cumbersome, we motivate a more abstract framework, VDM. We then carefully introduce the issues of soundness and completeness. We prove soundness, as usual, by induction on the derivation of the correctness formula. In our completeness proof, we are able to demonstrate that one may factor out details of assertions and adapt Gorelick's (1975) Most General Formula (MGF) theorem to the VDM setting.

Turning our attention to Hoare Logic, we discuss the rôle of auxiliary variables and propose to interpret assertions as relations on the state space and additionally a domain of auxiliary variables. Furthermore, we present a new rule of consequence. It is strictly stronger than Hoare's (1969) version and allows us to prove completeness as a trivial corollary of the MGF theorem. With the new treatment, we are able to show that derivations in VDM can be embedded in Hoare Logic.

We conclude with a summary of related work.

2.1 Deep and Shallow Embedding Techniques

Our main objective of this thesis is to mechanically establish meta-theorems such as soundness and completeness of verification calculi. It is **not** our aim to show that a proof tool such as the LEGO system is suitable to verify concrete programs relative to concrete specifications.

Traditionally, one defines syntax for expressions and relative to this setup, one characterises syntax of a programming language and syntax of an assertion language. Then, one describes the meaning of every syntactic construct. This approach is known as *deep embedding*. Alternatively one may shortcut this process and identify the syntactic representation with its denotation. This technique is known as *shallow embedding*.

The drawbacks of shallow embeddings are that

- one cannot exploit the inductive (syntactic) structure to prove properties
- the representation of concrete examples is often more difficult to comprehend

However, omitting syntactic representation cuts down the work load and is therefore often the preferred technique.

We will present a shallow embedding of expressions. Based on this definition, we will give a deep embedding of the programming language and a shallow embedding of the assertion language. A shallow embedding of the imperative programming language is inappropriate for our project, because the completeness proofs inherently rely on proofs by induction on the structure of programs.

A shallow embedding of Hoare Logic and VDM suffices to mechanically investigate soundness (Gordon 1989). However, to formalise the statement of completeness, one needs to explicitly refer to a notion of derivability. Thus, we give a deep embedding of Hoare Logic and VDM.

The main benefit of shallow embedding for our project has been that one does not have to worry about substitutions in assertions.

2.2 Program Variables

In practice, program variables are strings, sometimes restricted to a particular length. Strings are not directly supported in our logical framework. Instead of adopting an isomorphic structure e.g., lists over the finite set 0 – 255, we chose the most general structure. Any type will do as long as equality is decidable.

➥ p. 236 **Definition 2.1 (Program Variables)** *The syntactic class of program variables* **VAR** *may be an arbitrary type, provided it supports an associated decidable equality i.e., a*

boolean relation $eq : (\mathbf{VAR} \times \mathbf{VAR}) \to \texttt{bool}$ *which reflects equality*

$$\forall x, y : \mathbf{VAR} \cdot x = y \Leftrightarrow eq(x, y) = \mathbf{true} \quad .$$

We will investigate soundness and completeness relative to such a general notion of program variables. In logical frameworks supporting classical logic, the restriction to decidable equalities can be dropped. For concrete examples, we employ (finite) enumeration types $\mathbf{VAR} = (x, y, \dots)$.

Previous mechanisations only consider the case of a single data type, namely natural numbers. In a type-theoretic setting, it seems natural to investigate multiple sorts. For most data types present in imperative programming languages, we can immediately provide type-theoretic representations.

- Sum types yield finite data types e.g., booleans and characters.
- Inductive types yield data types such as natural numbers and lists.
- Employing quotients, we can construct integers from natural numbers and rationals from integers.
- Record types subsume records in programming languages.
- Using functions we may represent arrays. In particular, following the views of Dahl, Dijkstra & Hoare (1972) and Gries & Levin (1980), an assignment of an array element is regarded as functional updating.
- The type of program variables **VAR** can be used to represent pointers.

We identify the universe of data types with the universe of all types expressible in our logical framework. Conceptually, results in the sequel are therefore independent of the actual available data types. Pragmatically, we can study more interesting programming examples involving different data types. The type of variables can be declared by providing a function

$$\text{sort} : \mathbf{VAR} \to \texttt{Type} \quad .$$

2.3 The State Space

The state space records the value of every program variable. In the simple scenario of this chapter, there is exactly one entry for every program variable. We will encounter more interesting state spaces in the next chapter when dealing with local variables.

The state space is one of the major ingredients to deal with semantical aspects of imperative programs. When one encounters an assignment $x := t$, the semantics of the expression t depends on the value of all program variables occurring in t as recorded by the current state σ. Moreover, executing the assignment leads to a new state in which the entry for the program variable x in the state space has been updated. It suffices to consider *first-order substitutions* (Trakhtenbrot, Halpern & Meyer 1984) which restrict updating of an entry for $x : \mathbf{VAR}$ to values of type $\mathrm{sort}(x)$ which cannot refer to values of variables.

As we have seen, a type environment is a function from program variables to types. A state σ for a type environment sort is a function mapping program variables x to values of type $\mathrm{sort}(x)$. The state space itself is therefore a dependent function space:

➥ p. 140 **Definition 2.2 (State Space)** $\Sigma \stackrel{\text{def}}{=} \prod x : \mathbf{VAR} \cdot \mathrm{sort}(x)$

Updating (dependent) functions $\sigma[x \mapsto v]$ then corresponds to first-order substitution with $x : \mathbf{VAR}$, $v : \mathrm{sort}(x)$ and $\sigma : \Sigma$. Notice that a syntactic substitution such as

$$\sigma[x \mapsto x + 1]$$

is not well-typed. Instead, one needs to specify

$$\sigma[x \mapsto \sigma(x) + 1] \quad .$$

In LEGO, all functions are total. Thus, our model of the state space has no *undefined* values. At least for simple data types, such as integers, this is an adequate assumption, since, in the real world, one may access any entry in the memory and interpret it as a value. If one were interested in modelling undefined values for a data type T, one could store values of the sum type $T + 1$ where the right injection characterises an undefined value.

2.4 Interpretations

An interpretation determines the value of

- constants such as 0, $+$, $\wedge$
- and (free) variables

in expressions and logical formulae. In the context of program verification, logical variables are assumed to refer to program variables. Therefore, the interpretation I for variables depends on a particular state σ i.e., $[\![x]\!]\big(I(\sigma)\big) = I\big(\sigma(x)\big)$. In the following

chapter, when considering local variables, scoping dictates which program variables in the state space are to be inspected and the interpretation depends additionally on a scoping environment.

In our formalisation, we are only interested in the standard interpretation of constants. This not only simplifies the encoding, it also avoids the problematic issue of how to axiomatise the class of acceptable interpretations, see Sect. 2.12 for further details.

2.5 Expressions

Boolean expressions occur in loops and conditional statements. Other types of expressions depend on the data types expressible in the language and occur both as subexpressions of boolean expressions and in the assignment statement. One may define the syntax of expressions by a BNF grammar.

Example 2.3 (Syntax of Expression) *Homeier & Martin (1996) define two classes of expressions*

$$e ::= n \mid x \mid ++x \mid e_1 + e_2 \mid e_1 - e_2$$
$$b ::= e_1 = e_2 \mid e_1 < e_2 \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \neg b$$

We will only consider expressions without side-effects[1] and do not deal with the expression $++x$. The semantics can thus be easily fixed denotationally and is determined by an interpretation function I and a state σ.

Example 2.4 (Semantics of Expressions)

- $[\![n]\!](I(\sigma)) \stackrel{\text{def}}{=} I(n) \quad [\![x]\!](I(\sigma)) \stackrel{\text{def}}{=} I\big(\sigma(x)\big)$
- $[\![e_1\, op\, e_2]\!](I(\sigma)) \stackrel{\text{def}}{=} [\![e_1]\!](I(\sigma))\, I(op)\, [\![e_2]\!](I(\sigma)) \quad \text{for } op \in \{+, -, =, <\}$
- $[\![b_1\, op\, b_2]\!](I(\sigma)) \stackrel{\text{def}}{=} [\![b_1]\!](I(\sigma))\, I(op)\, [\![b_2]\!](I(\sigma)) \quad \text{for } op \in \{\wedge, \vee\}$
- $[\![\neg b]\!](I(\sigma)) \stackrel{\text{def}}{=} I(\neg)[\![b]\!](I(\sigma))$

Whenever we come across a boolean expression in a loop or a conditional statement, we are only interested in the value it evaluates to, **true** or **false**. Similarly, in an assignment, we treat evaluation of the expression as atomic, merely a value depending on the state space. We are not interested in syntactic properties such as whether one expression is a subterm of another expression. Ignoring the syntax of expressions paves the

[1] Such a strict distinction between expressions and commands is one of the fundamental principles underlying idealised Algol (Reynolds 1982).

way towards a reasonable level of abstraction when investigating properties of verification calculi for imperative programs without side-effects.

Restricting our attention to the standard interpretation, the semantics of an expression is determined by the state space. We only consider expressions at this semantic level:

➥ p. 140 **Definition 2.5 (Expressions)** *Given an arbitrary type T, we represent expressions by*

$$\text{expression}(T : \texttt{Type}) \stackrel{\text{def}}{=} \Sigma \to T \quad .$$

Let $e : \text{expression}(T)$ be any expression. Its evaluation depends on a concrete snapshot of the state space $\sigma : \Sigma$ We define

$$\text{eval}(\sigma)(e) \stackrel{\text{def}}{=} e(\sigma) \quad .$$

The main benefit of adopting shallow embedding is that we do not have to worry about formalising the syntax in a logical framework e.g., the expression $x * y$ where x and y are program variables with sort `nat` would be rendered as

$$\lambda\sigma \cdot \sigma(x) * \sigma(y) : \text{expression}(\texttt{nat}) \quad .$$

Moreover, substitutions are much easier to deal with at the semantic level. Consider the syntactic substitution $(x * y)[x \mapsto t]$. In the shallow embedding, this substitution can be realised by updating the state space i.e.

$$\big(\lambda\sigma \cdot \sigma(x) * \sigma(y)\big) \circ \big(\lambda\sigma \cdot \sigma[x \mapsto \text{eval}(\sigma)(t)]\big) \tag{2.1}$$

which, when applied to a concrete state σ, (β-)reduces to

$$\big(\sigma[x \mapsto \text{eval}(\sigma)(t)](x)\big) * \big(\sigma[x \mapsto \text{eval}(\sigma)(t)](y)\big) \quad . \tag{2.2}$$

This is equivalent to

$$t' * \sigma(y) \tag{2.3}$$

where t' is a suitably reduced version of $\text{eval}(\sigma)(t)$. Employing a deep embedding, one would need to additionally check that the syntactic substitution function on expressions is correct e.g., $[\![(x * y)[x \mapsto t]]\!]$ must coincide with the term (2.1).

Unfortunately, the LEGO system offers no automatic support in reducing (2.2) to (2.3). In concrete examples, such as the Quicksort verification in Cha. 4, this leads to excessively large proof obligations. Moreover, examples are more difficult to read. To increase the readability of examples, we shall present the (manually translated) syntactic counterparts of expressions.

2.6 Imperative Programs

For representing statements of the imperative programming language, we resort to a deep embedding i.e., we first embed the syntax in the logical framework and then axiomatise the semantics by an analysis of the structure of programs. In this chapter, we restrict ourselves to basic language features:

➥ p. 140 **Definition 2.6 (Abstract Syntax of Imperative Programs)**
Imperative programs $S : \texttt{prog}$ *are defined by the BNF grammar*

$$S ::= \textbf{skip} \mid x := t \mid S_1; S_2 \mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \mid \textbf{while } b \textbf{ do } S$$

where $x : \textbf{VAR}$, $t : \text{expression}(\text{sort}(x))$ *and* $b : \text{expression}(\texttt{bool})$.

As LEGO supports definitions, we may define other, possibly generic, language constructs e.g., **if** b **then** $S \stackrel{\text{def}}{=}$ **if** b **then** S **else skip**. However, any program $S : \texttt{prog}$ may be brought into normal form e.g., by expanding definitions, so that when pursuing induction on the structure of S, we only need to consider the primitive constructors given in Def. 2.6.

In examples, we assume that sequential composition is left-associative. Furthermore, we employ **begin** and **end** in our *concrete* syntax to clarify precedence.

Notice that LEGO actually needs to test equivalence of types to resolve the type dependency in the assignment constructor:

Example 2.7 *In the assignment* $k := 1$, *the principal type of* 1 *is* `nat`. *However, the typing discipline for the assignment constructor requires* 1 *to inhabit* $\text{sort}(k)$. *The typechecker thus needs to reduce* $\text{sort}(k)$ *to* `nat`.

2.6.1 Assignment and Arrays

Data types may be any type representable in type theory. This includes arrays which, following Dahl et al. (1972) and Gries & Levin (1980), one may consider as functions inhabiting $T \to U$, where U is an arbitrary type and T is any type where equality is decidable.

To represent the concrete statement $x[c] := x[d]$ for an array $x : \textbf{VAR}$ with $\text{sort}(x) = T \to U$ and $\text{sort}(c) = \text{sort}(d) = T$, the assignment constructor expects a term inhabiting $\left(\prod x : \textbf{VAR} \cdot \text{sort}(x)\right) \to T \to U$ as its second argument. Since x represents a function, the statement $x[c] := x[d]$ corresponds to updating the function x at position c i.e.,

$$x := \lambda\sigma : \Sigma \cdot (\sigma(x))[\sigma(c) \mapsto \sigma(x)(\sigma(d))] \quad .$$

2.7 Structural Operational Semantics

In this section, we axiomatise the semantics of the imperative programming language. We employ structural operational semantics (Plotkin 1981) which provides a clean way to specify the effect of each language constructor in an arbitrary state. It relates a program with its initial and final state.

➥ p. 141 **Definition 2.8 (Semantics)** *The operational semantics is defined as the least relation* $. \xrightarrow{\cdot} . \subseteq \Sigma \times \mathtt{prog} \times \Sigma$ *satisfying*

$$\sigma \xrightarrow{\textbf{skip}} \sigma \tag{2.4}$$

$$\sigma \xrightarrow{x:=t} \sigma[x \mapsto \mathrm{eval}(\sigma)(t)] \tag{2.5}$$

$$\frac{\sigma \xrightarrow{S_1} \eta \quad \eta \xrightarrow{S_2} \tau}{\sigma \xrightarrow{S_1;\, S_2} \tau} \tag{2.6}$$

$$\frac{\sigma \xrightarrow{S_1} \tau}{\sigma \xrightarrow{\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2} \tau} \quad \textit{provided } \mathrm{eval}(\sigma)(b) = \textbf{true} \ . \tag{2.7}$$

$$\frac{\sigma \xrightarrow{S_2} \tau}{\sigma \xrightarrow{\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2} \tau} \quad \textit{provided } \mathrm{eval}(\sigma)(b) = \textbf{false} \ . \tag{2.8}$$

$$\sigma \xrightarrow{\textbf{while } b \textbf{ do } S} \sigma \quad \textit{provided } \mathrm{eval}(\sigma)(b) = \textbf{false} \ . \tag{2.9}$$

$$\frac{\sigma \xrightarrow{S} \eta \quad \eta \xrightarrow{\textbf{while } b \textbf{ do } S} \tau}{\sigma \xrightarrow{\textbf{while } b \textbf{ do } S} \tau} \quad \textit{provided } \mathrm{eval}(\sigma)(b) = \textbf{true} \ . \tag{2.10}$$

Intuitively, $\sigma \xrightarrow{S} \tau$ denotes that the program S when invoked in the state σ will terminate in the state τ.

We can increase our confidence in the correctness of this axiomatisation by proving properties such as determinism.

➥ p. 142 **Lemma 2.9 (Determinism)**
Program behaviour is deterministic i.e., for all programs S and states η, σ and τ, whenever $\sigma \xrightarrow{S} \tau$ and $\sigma \xrightarrow{S} \eta$, then $\tau = \eta$.

Proof by induction on the derivation of $\sigma \xrightarrow{S} \tau$, followed by inverting $\sigma \xrightarrow{S} \eta$ in each case. □

However, being axiomatic, it is impossible to *prove* the correctness of the semantic description. We have to inspect each rule carefully.

We believe a denotational[2] semantics $M : (\texttt{prog} \times \Sigma) \rightharpoonup \Sigma$ is less suitable to *define* the behaviour of programs, because it involves more complicated mathematical constructs such as computing least fixed points e.g., Cousot (1990) argues that a denotational account should be justified with respect to an operational semantics. In our opinion, the induction principle triggered by the inductive definition of the operational semantics is sufficiently powerful to effectively investigate soundness and completeness. We do not see the need to deal with denotational semantics.

2.8 Low-Level Verification

Having integrated the semantics of a programming language in a logical framework, we can specify program behaviour and prove specifications correct. In this section, we employ the rules of the structural operational semantics to verify an imperative program implementing exponentiation. Analysing the proof, we uncover redundant steps which suggests that one might benefit from resorting to a set of higher-level proof rules.

As global program variables, we use $\mathbf{VAR} = \{x, y, r\}$ and we assume that all program variables may also take on negative values. More precisely, we declare

$$\textbf{fun}\,\mathrm{sort}(_) = \texttt{int}\ .$$

An implementation then amounts to $S_{\mathrm{exp}} \stackrel{\mathrm{def}}{=} r := 1;\ loop_{\mathrm{exp}}$ with

$loop_{\mathrm{exp}} \stackrel{\mathrm{def}}{=}$ **while** $y \neq 0$ **do begin** $r := r * x;\ y := y - 1$ **end**

A possible specification in our current setup might be

$$\forall \sigma, \tau \cdot (\sigma \xrightarrow{S_{\mathrm{exp}}} \tau) \Rightarrow \tau(r) = \sigma(x)^{\sigma(y)} \tag{2.11}$$

This is a typical input/output specification (Liskov & Berzins 1986). The state σ captures the initial state, the state τ captures the final state of the program. The proposition $\tau(r) = \sigma(x)^{\sigma(y)}$ connects the *input* values of x and y with the *output* value of r. Notice that connecting the output value of r with the *output* values of x and y e.g., $\tau(r) = \tau(x)^{\tau(y)}$, would be an unwise requirement: If S_{exp} terminates, the output value of y amounts to 0 and thus, the program S_{exp} would only satisfy the specification for an initial value of 1 for x.

Specifications of the above shape trivially hold if programs do not terminate. In such a case, there are no states σ, τ satisfying $\sigma \xrightarrow{S_{\mathrm{exp}}} \tau$. Hence, (2.11) only

[2] To avoid partiality, one may instead use a *relational* denotational semantics $M : \texttt{prog} \to 2^{\Sigma \times \Sigma}$.

specifies *partial correctness*. In this thesis, we consider *total correctness*. The proposition $\forall\sigma\cdot\exists\tau\cdot\sigma \xrightarrow{S} \tau$ characterises that S terminates on all inputs. In practice, we do not want to insist on termination for all initial states. In particular, we have designed our exponential program in such a way that it only terminates for positive values of y i.e., $\forall\sigma\cdot 0 \leq \sigma(y) \Rightarrow \exists\tau\cdot\sigma \xrightarrow{S_{\text{exp}}} \tau$. Combining the termination condition with (2.11) yields an adequate specification with respect to total correctness:

➥ p. 144

Lemma 2.10 (Correctness of Exponential Program)

$$\forall\sigma\cdot 0 \leq \sigma(y) \Rightarrow \exists\tau\cdot\sigma \xrightarrow{S_{exp}} \tau \wedge \tau(r) = \sigma(x)^{\sigma(y)}$$

Proof

Given a state σ such that

$$0 \leq \sigma(y) \quad , \tag{2.12}$$

we need to find a final state τ such that

$$\sigma \xrightarrow{S_{\text{exp}}} \tau \tag{2.13}$$

and $\tau(r) = \sigma(x)^{\sigma(y)}$ holds. Applying the rule for sequential composition (2.6), followed by the assignment axiom (2.5) simplifies the first proof obligation (2.13) to

$$\sigma[r \mapsto 1] \xrightarrow{loop_{\text{exp}}} \tau \quad . \tag{2.14}$$

We have thus reduced the original proof obligation to

$$\forall\sigma\cdot 0 \leq \sigma(y) \Rightarrow \exists\tau\cdot\sigma[r \mapsto 1] \xrightarrow{loop_{\text{exp}}} \tau \wedge \tau(r) = \sigma(x)^{\sigma(y)} \quad . \tag{2.15}$$

Since each execution of the loop body decreases the value of y until it is 0, we pursue *induction*. For the induction to go through, we have to strengthen the proof obligation (2.15) so that the state σ corresponds to an arbitrary intermediate state of the loop. In particular, it must not be restricted to the case where the value of r amounts to the initial 1. We choose

$$\forall\sigma\cdot 0 \leq \sigma(y) \Rightarrow \exists\tau\cdot\sigma \xrightarrow{loop_{\text{exp}}} \tau \wedge \tau(y) = 0 \wedge \tau(r) = \sigma(r) * \sigma(x)^{\sigma(y)} \tag{2.16}$$

and pursue induction on $\sigma(y)$. Formally, we derive

$$\forall n\cdot\forall\sigma\cdot(0 \leq \sigma(y) \wedge \sigma(y) = n) \Rightarrow \exists\tau\cdot\sigma \xrightarrow{loop_{\text{exp}}} \tau \wedge \tau(y) = 0 \wedge \tau(r) = \sigma(r) * \sigma(x)^{\sigma(y)}$$

by induction on n.

In the base case, the axiom for loops (2.9) unifies σ with τ and we have to show

$$\sigma(r) = \sigma(r) * \sigma(r)^0 \quad .$$

In the step case, for $\sigma(y) = n+1$, we need to find a state τ which satisfies

$$\sigma \xrightarrow{loop_{\text{exp}}} \tau \tag{2.17}$$

and

$$\tau(y) = 0 \wedge \tau(r) = \sigma(r) * \sigma(x)^{\sigma(y)} \quad . \tag{2.18}$$

Unwinding the loop once reduces the proof obligation (2.17) to

$$\eta \xrightarrow{loop_{\text{exp}}} \tau \quad \text{for } \eta = \sigma[r \mapsto \sigma(r) * \sigma(x)][y \mapsto \sigma(y) - 1]. \tag{2.19}$$

We may now apply the induction hypothesis with η, because $n = \eta(y)$. This yields

$$\tau(y) = 0 \wedge \tau(r) = \sigma(r) * \sigma(x) * \sigma(x)^{\sigma(y)-1}$$

which is equivalent to the remaining proof obligation (2.18) for $\sigma(y) > 0$.

□

As part of the proof, given an initial state σ satisfying a particular precondition, we needed to find a final state τ such that, according to the rule of the operational semantics, we could derive $\sigma \xrightarrow{S} \tau$. Furthermore, we had to establish that the postcondition relating σ and τ held.

However, being deterministic, the operational semantics uniquely determines such a final state. In addition, the initial correctness formula, the proof obligation (2.19) and the crucial proof obligation (2.16) are all of the form

$$\forall \sigma \cdot p(\sigma) \Rightarrow \exists \tau \cdot \sigma \xrightarrow{S} \tau \wedge q(\tau, \sigma) \tag{2.20}$$

for particular instances of $p : \Sigma \to \texttt{Prop}$, $S : \texttt{prog}$ and $q : (\Sigma \times \Sigma) \to \texttt{Prop}$. Instead of resorting to inference steps preserving this structure, the above proof has unnecessarily employed introduction and elimination rules for the logical connectives $\forall$, $\Rightarrow$ and $\exists$ to destruct and reconstruct such forms.

In VDM, one focusses on deriving correctness formulae of the form (2.20). Hoare Logic is similar, but postconditions are restricted to predicates rather than binary relations on the state space. One can thus not directly specify input/output behaviour. To compensate for this deficiency, one needs to additionally deal with auxiliary variables. We therefore first consider VDM.

2.9 The Vienna Development Method

The Vienna Development Method (VDM) is a verification calculus which relates programs with pre- and postconditions (Jones 1990, Jones & Shaw 1990). We commence by introducing VDM, in particular its operation decomposition calculus, at an informal level. Employing a shallow embedding of assertions, we then show how one might represent such a verification calculus in a logical framework. Next, we reconsider the above verification proof in VDM before investigating soundness and completeness.

In VDM, assertions are objects of a logic of partial functions (LPF). In the postcondition one may employ *hooked* program variables $\overleftarrow{x}$ to refer to the value of the program variable in the initial state. The meaning of a postcondition is determined by an interpretation $I(\tau,\sigma)$ relative to a final state τ and initial state σ. In particular, for a program variable x, we have

$$\begin{aligned}[\![x]\!]I(\tau,\sigma) &\stackrel{\text{def}}{=} I\big(\tau(x)\big)\\ [\![\overleftarrow{x}]\!]I(\tau,\sigma) &\stackrel{\text{def}}{=} I\big(\sigma(x)\big) \quad .\end{aligned}$$

Hooked notation is extended to expressions and preconditions. Hooked expressions and preconditions may only appear within postconditions. On a syntactic level, one has to hook all free occurring variables. Semantically, let t be an expression and p be a precondition. One then interprets

$$\begin{aligned}[\![\overleftarrow{t}]\!]I(\tau,\sigma) &\stackrel{\text{def}}{=} [\![t]\!]I(\sigma)\\ [\![\overleftarrow{p}]\!]I(\tau,\sigma) &\stackrel{\text{def}}{=} [\![p]\!]I(\sigma) \quad .\end{aligned}$$

VDM is a formal system for deriving correctness formulae $\{p\}\ S\ \{q\}$. Such a triple is interpreted as

$$\forall\sigma\cdot [\![p]\!]I(\sigma) \Rightarrow \exists\tau\cdot\sigma \xrightarrow{\quad S \quad} \tau \wedge [\![q]\!]I(\tau,\sigma)$$

and thus reflects the shape of correctness formulae as encountered in the above verification example. However, being a formal system which manipulates objects of the form $\{p\}\ S\ \{q\}$, VDM guides verification proofs at a more abstract level. Whereas in the above correctness proof, one was confronted with (higher-order) propositions built around a formal system relating programs with states, VDM is a formal system relating programs with *relations on states*. More precisely, VDM encompasses two orthogonal methods:

1. refining abstract objects into concrete data types available in the programming language, and

2. refining correctness formulae, which allows one to decompose programs.

We will only consider VDM's operation decomposition rules and, from now on, do no longer distinguish between them and VDM in general.

For every constructor of the imperative programming language, VDM provides a rule which allows one to decompose a program. In particular, the programs mentioned in the premisses are strict subprograms of the programs mentioned in the conclusions. Unlike the operational semantics, this also holds for loops where a well-founded induction argument has been built into the rule.

$$\{\mathbf{true}\}\ \mathbf{skip}\ \left\{ \bigwedge_{v \in \mathbf{VAR}} v = \overleftarrow{v} \right\}$$

$$\{\mathbf{true}\}\ x := t\ \left\{ x = \overleftarrow{t} \wedge \bigwedge_{v \in \mathbf{VAR} \setminus \{x\}} v = \overleftarrow{v} \right\}$$

$$\frac{\{p_1\}\ S_1\ \{p_2 \wedge r_1\} \quad \{p_2\}\ S_2\ \{r_2\}}{\{p_1\}\ S_1;\ S_2\ \{r_2 \circ r_1\}}$$

Semantically, the postcondition in the conclusion corresponds to relational composition i.e., $[\![r_2 \circ r_1]\!]I(\tau,\sigma) \stackrel{\text{def}}{=} \exists \eta \cdot [\![r_1]\!]I(\eta,\sigma) \wedge [\![r_2]\!]I(\tau,\eta)$. Syntactically, it suffices to existentially quantify intermediate values for all free occurring program variables and substitute them for all free program variables in r_1 and all hooked program variables in r_2.

Example 2.11 (VDM's Sequential Decomposition Rule) *Let*

$$r_1 \stackrel{\text{def}}{=} r = 1 \wedge 0 \leq y \wedge x = \overleftarrow{x} \wedge y = \overleftarrow{y}$$
$$r_2 \stackrel{\text{def}}{=} y = 0 \wedge \overleftarrow{r} * \overleftarrow{x}^{\overleftarrow{y}} = r * x^y \wedge y \leq \overleftarrow{y}$$

Then, $r_2 \circ r_1$ corresponds to

$$\exists x_i, y_i, r_i \cdot r_i = 1 \wedge 0 \leq y_i \wedge x_i = \overleftarrow{x} \wedge y_i = \overleftarrow{y} \wedge y = 0 \wedge r_i * x_i^{y_i} = r * x^y \wedge y \leq y_i$$

which is equivalent to $0 \leq \overleftarrow{y} \wedge y = 0 \wedge r = \overleftarrow{x}^{\overleftarrow{y}}$.

$$\frac{\{p \wedge b\}\ S_1\ \{q\} \quad \{p \wedge \neg b\}\ S_2\ \{q\}}{\{p\}\ \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \{q\}}$$

$$\frac{\{p \wedge b\}\ S\ \{p \wedge \mathit{sofar}\}}{\{p\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{p \wedge \neg b \wedge (\mathit{sofar} \vee \bigwedge_{v \in \mathbf{VAR}} v = \overleftarrow{v})\}}$$

where *sofar* is[3] transitive and well-founded.

As Jones (1990) points out, it suffices to ensure that the relation *sofar* is well-founded on the set of states *satisfying the invariant p*. It would thus suffice, if the relation $\lambda(\tau,\sigma) \cdot [\![\mathit{sofar}]\!] I(\tau,\sigma) \wedge [\![p]\!] I(\tau)$ was well-founded.

One also needs the structural rule

$$\frac{\{p_1\}\ S\ \{q_1\}}{\{p\}\ S\ \{q\}} \qquad \text{provided } p \Rightarrow p_1 \text{ and } \overleftarrow{p} \Rightarrow q_1 \Rightarrow q$$

which allows one to weaken the precondition and strengthen the postcondition in a proof obligation. This is particularly useful when one wants to apply the rule for loops as the precondition must remain invariant with respect to the body of the loop, whereas the postcondition is expected to incorporate justification for termination.

2.9.1 Mechanisation

The above presentation of VDM is due to Aczel (1982*a*, 1982*b*). There, preconditions are treated as predicates on states, whereas postconditions are captured by binary relations on states. In particular, assertions are not considered at a syntactic level and there is no reference to LPF.

We follow Aczel's approach and employ a shallow embedding to represent assertions. In a logical framework supporting set theory, one might use sets to capture relations. Using LEGO, we only consider sets of the form $\{x \mid P(x)\}$ where $P : T \to \texttt{Prop}$ is a *function* mapping elements of an arbitrary type T to LEGO's internal logic. In other words, we represent sets by their characteristic functions. Thus, in VDM, preconditions inhabit $\Sigma \to \texttt{Prop}$ and postconditions inhabit $(\Sigma \times \Sigma) \to \texttt{Prop}$.

We are interested in the meta theory of VDM. We therefore need to capture both the semantics and the notion of deriving a VDM correctness formula.

➥ p. 144 **Definition 2.12 (Semantics of VDM)** *Let*

$$\models_{\text{VDM}} \{.\}\ .\ \{.\} \subseteq (\Sigma \to \texttt{Prop}) \times \texttt{prog} \times \big((\Sigma \times \Sigma) \to \texttt{Prop}\big)$$

be a new judgement defined in terms of the operational semantics

$$\models_{\text{VDM}} \{p\}\ S\ \{q\} \stackrel{\text{def}}{=} \forall \sigma \cdot p(\sigma) \Rightarrow \exists \tau \cdot \sigma \xrightarrow{\quad S \quad} \tau \wedge q(\tau,\sigma)$$

[3] i.e., the relation $\lambda(\tau,\sigma) \cdot [\![\mathit{sofar}]\!](\tau,\sigma)$

➥ p. 145 **Definition 2.13 (VDM)** *Let*

$$\vdash_{\text{VDM}} \{.\}\ .\ \{.\} \subseteq (\Sigma \to \texttt{Prop}) \times \texttt{prog} \times \big((\Sigma \times \Sigma) \to \texttt{Prop}\big)$$

be the least relation satisfying

$$\vdash_{\text{VDM}} \{\lambda\sigma \cdot \textbf{true}\}\ \textbf{skip}\ \{\lambda(\tau,\sigma) \cdot \tau = \sigma\} \tag{2.21}$$

$$\vdash_{\text{VDM}} \{\lambda\sigma \cdot \textbf{true}\}\ x := t\ \{\lambda(\tau,\sigma) \cdot \tau = \sigma[x \mapsto \text{eval}(\sigma)(t)]\} \tag{2.22}$$

$$\frac{\vdash_{\text{VDM}} \{p_1\}\ S_1\ \{\lambda(\tau,\sigma) \cdot p_2(\tau) \wedge r_1(\tau,\sigma)\} \quad \vdash_{\text{VDM}} \{p_2\}\ S_2\ \{r_2\}}{\vdash_{\text{VDM}} \{p_1\}\ S_1;\ S_2\ \{r_2 \circ r_1\}} \tag{2.23}$$

$$\frac{\begin{array}{c}\vdash_{\text{VDM}} \{\lambda\sigma \cdot p(\sigma) \wedge \text{eval}(\sigma)(b) = \textbf{true}\}\ S_1\ \{q\} \\ \vdash_{\text{VDM}} \{\lambda\sigma \cdot p(\sigma) \wedge \text{eval}(\sigma)(b) = \textbf{false}\}\ S_2\ \{q\}\end{array}}{\vdash_{\text{VDM}} \{p\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{q\}} \tag{2.24}$$

$$\frac{\vdash_{\text{VDM}} \{\lambda\sigma \cdot p(\sigma) \wedge \text{eval}(\sigma)(b) = \textbf{true}\}\ S\ \{\lambda(\tau,\sigma) \cdot p(\tau) \wedge \mathit{sofar}(\tau,\sigma)\}}{\vdash_{\text{VDM}} \{p\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{q\}}$$

where sofar is any binary transitive relation on the state space Σ *such that*

$$\lambda(\tau,\sigma) \cdot \mathit{sofar}(\tau,\sigma) \wedge p(\tau)\ \textit{is well-founded}$$

$$\textit{and}\ q(\tau,\sigma) \stackrel{\text{def}}{=} p(\tau) \wedge \text{eval}(\tau)(b) = \textbf{false} \wedge \big(\mathit{sofar}(\tau,\sigma) \vee \tau = \sigma\big). \tag{2.25}$$

$$\frac{\vdash_{\text{VDM}} \{p_1\}\ S\ \{q_1\}}{\vdash_{\text{VDM}} \{p\}\ S\ \{q\}}$$

$$\textit{provided}\ \forall\sigma \cdot p(\sigma) \Rightarrow p_1(\sigma)\ \textit{and}\ \forall(\tau,\sigma) \cdot p(\sigma) \Rightarrow q_1(\tau,\sigma) \Rightarrow q(\tau,\sigma). \tag{2.26}$$

2.9.2 High-Level Verification

We now revisit the verification of the exponential program within the VDM framework. To increase readability, we use syntactic representations for assertions. These have been manually translated from the shallow embedding used in the actual script.

➥ p. 148 **Lemma 2.14 (VDM – Correctness of Exponential Function)**

$$\vdash_{\text{VDM}} \{0 \le y\}\ S_{exp}\ \left\{r = \overleftarrow{x}^{\overleftarrow{y}}\right\}$$

Proof According to the axiom for assignments (2.22), we may derive the correctness formula $\vdash_{\text{VDM}} \{\textbf{true}\}\ r := 1\ \{r = 1 \wedge x = \overleftarrow{x} \wedge y = \overleftarrow{y}\}$. Taking $0 \le y$ into account, by the consequence rule (2.26) we may deduce

$$\vdash_{\text{VDM}} \{0 \le y\}\ r := 1\ \{r = 1 \wedge x = \overleftarrow{x} \wedge y = \overleftarrow{y} \wedge 0 \le y\}\ . \tag{2.27}$$

The challenge in the correctness proof stems from instantiating the loop rule (2.25) with an appropriate invariance predicate and termination measure. From the postcondition of (2.27), we adopt $0 \leq y$ as an invariant. In addition, the loop leaves $r * x^y$ invariant as decreases of y are compensated by increasing r appropriately:

$$sofar \stackrel{\text{def}}{=} \overleftarrow{r} * \overleftarrow{x}^{\overleftarrow{y}} = r * x^y \wedge y < \overleftarrow{y} \quad .$$

We thus focus on deriving

$$\vdash_{\text{VDM}} \{0 \leq y\}\ loop_{\text{exp}}\ \{0 \leq y \wedge \neg(y \neq 0) \wedge sofar\} \tag{2.28}$$

which is semantically equivalent to the proof obligation (2.16). We may decompose (2.28) by the rule for loops (2.25) as *sofar* is clearly transitive and in the light of $0 \leq y$, there are only finitely many smaller $\overleftarrow{y}$. We now need to show

$$\vdash_{\text{VDM}} \{0 \leq y \wedge y \neq 0\}\ r := r * x;\ y := y - 1\ \{0 \leq y \wedge sofar\} \tag{2.29}$$

According to the axiom (2.22), we may deduce

$$\vdash_{\text{VDM}} \{\mathbf{true}\}\ r := r * x\ \{r = \overleftarrow{r} * \overleftarrow{x} \wedge x = \overleftarrow{x} \wedge y = \overleftarrow{y}\} \tag{2.30}$$

$$\text{and} \quad \vdash_{\text{VDM}} \{\mathbf{true}\}\ y := y - 1\ \{y = \overleftarrow{y} - 1 \wedge x = \overleftarrow{x} \wedge r = \overleftarrow{r}\} \tag{2.31}$$

Connecting (2.30) with the precondition of (2.29), the rule of consequence (2.26) yields

$$\vdash_{\text{VDM}} \{0 \leq y \wedge y \neq 0\}\ r := r * x\ \{r = \overleftarrow{r} * \overleftarrow{x} \wedge x = \overleftarrow{x} \wedge 0 < y \wedge y = \overleftarrow{y}\} \tag{2.32}$$

We now derive

$$\vdash_{\text{VDM}} \{0 \leq y \wedge y \neq 0\}\ r := r * x;\ y := y - 1\ \{postbody\} \tag{2.33}$$

with

$$postbody \stackrel{\text{def}}{=} \exists x_i, y_i, r_i \cdot r_i = \overleftarrow{r} * \overleftarrow{x} \wedge x_i = \overleftarrow{x} \wedge 0 < y_i \wedge y_i = \overleftarrow{y} \wedge y = y_i - 1 \wedge x = x_i \wedge r = r_i$$

by applying the sequential rule (2.23) to the correctness formulae (2.32) and (2.31). Appealing to the consequence rule (2.26), it suffices to show that the postcondition of (2.33) entails the postcondition of (2.29). The above postcondition is equivalent to

$$r = \overleftarrow{r} * \overleftarrow{x} \wedge x = \overleftarrow{x} \wedge 0 \leq y \wedge y = \overleftarrow{y} - 1 \quad . \tag{2.34}$$

It is straightforward to check that $0 \leq y \wedge sofar$ follows from (2.34). Combining the initialisation (2.27) with the loop (2.28) via the rule of sequential composition (2.23) leads to

$$\vdash_{\text{VDM}} \{0 \leq y\}\ S_{\text{exp}}\ \left\{0 \leq \overleftarrow{y} \wedge y = 0 \wedge r = \overleftarrow{x}^{\overleftarrow{y}}\right\}$$

Appealing to the consequence rule (2.26), we may weaken the postcondition and derive the desired correctness formula. □

2.9.3 Soundness

In this section, we justify the VDM rules and show that VDM is merely a more abstract calculus for verifying correctness formulae. In particular, any derivation in VDM corresponds to a proof based on the rules of the operational semantics.

➥ p. 149

Theorem 2.15 (Soundness of VDM) *For any precondition p, program S and postcondition q, whenever $\vdash_{\text{VDM}} \{p\}\ S\ \{q\}$ is derivable, $\models_{\text{VDM}} \{p\}\ S\ \{q\}$ holds.*

Proof by induction on the derivation of $\vdash_{\text{VDM}} \{p\}\ S\ \{q\}$. We consider the two cases where $\vdash_{\text{VDM}} \{p\}\ S\ \{q\}$ has been derived by the rule for conditionals or loops in more detail:

Conditional From the two induction hypotheses

$$\models_{\text{VDM}} \{\lambda\sigma \cdot p(\sigma) \wedge \text{eval}(\sigma)(b) = \textbf{true}\}\ S_1\ \{q\} \tag{2.35}$$

$$\models_{\text{VDM}} \{\lambda\sigma \cdot p(\sigma) \wedge \text{eval}(\sigma)(b) = \textbf{false}\}\ S_2\ \{q\} \tag{2.36}$$

we need to establish

$$\models_{\text{VDM}} \{p\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{q\} \quad .$$

We show this by expanding the definition of $\models_{\text{VDM}}$ and considering the cases for $\text{eval}(\sigma)(b) = \textbf{true}$ and $\text{eval}(\sigma)(b) = \textbf{false}$. The conclusion follows from (2.35) and (2.36) respectively.

Loop Given a state satisfying the predicate p, we need to find a final state τ such that

$$\sigma \xrightarrow{\textbf{while}\ b\ \textbf{do}\ S} \tau \ , \tag{2.37}$$

$$p(\tau) \ , \tag{2.38}$$

$$\text{eval}(\tau)(b) = \textbf{false} \tag{2.39}$$

and

$$\mathit{sofar}(\tau, \sigma) \vee \tau = \sigma \ . \tag{2.40}$$

We commence with a case analysis on the initial value of the guard b. If it is **false**, (2.9) is applicable. In particular, for $\tau = \sigma$, (2.38)–(2.40) trivially hold. Otherwise, we may exploit the induction hypothesis

$$\models_{\text{VDM}} \{\lambda\sigma \cdot p(\sigma) \wedge \text{eval}(\sigma)(b) = \textbf{true}\}\ S\ \{\lambda(\tau, \sigma) \cdot p(\tau) \wedge \mathit{sofar}(\tau, \sigma)\} \quad .$$

There must be a state η such that

$$\sigma \xrightarrow{S} \eta \ , \tag{2.41}$$

$p(\eta)$ and

$$sofar(\eta, \sigma) \quad . \tag{2.42}$$

Is the guard b satisfied in state η?

No: From (2.41), we may conclude $\sigma \xrightarrow{\textbf{while } b \textbf{ do } S} \eta$. For $\tau = \eta$, (2.38)–(2.40) trivially hold.

Yes: The side-condition of the rule for loops guarantees that the relation

$$\lambda(\tau, \sigma) \cdot sofar(\tau, \sigma) \wedge p(\tau)$$

is well-founded. We know that η is less than σ with respect to this ordering and may thus exploit variants of (2.38)–(2.40) where σ has been substituted by η. In particular, there has to be a state τ satisfying (2.38), (2.39),

$$\eta \xrightarrow{\textbf{while } b \textbf{ do } S} \tau \tag{2.43}$$

and

$$sofar(\tau, \eta) \vee \tau = \eta \quad . \tag{2.44}$$

In fact, τ is the desired final state. Applying the rule for loops (2.10) to (2.41) and (2.43) justifies (2.37), whereas transitivity of *sofar* leads from (2.44) and (2.42) to (2.40).

□

2.9.4 Completeness

If the VDM rules are to serve as the basis of a methodology for developing programs, completeness is an important issue i.e., given any derivable correctness formula of the form $\models_{\text{VDM}} \{p\}\ S\ \{q\}$, can we also justify it only employing the VDM rules? In other words, in moving from the level of states to the level of predicates and relations over states, have we actually achieved a more abstract perspective? Our objective is to show that whenever $\models_{\text{VDM}} \{p\}\ S\ \{q\}$ holds, the judgement $\vdash_{\text{VDM}} \{p\}\ S\ \{q\}$ is derivable.

One may prove completeness directly by induction on the structure of S e.g., Aczel (1982*a*) establishes soundness and completeness simultaneously by induction on the structure of programs. Instead, we follow an idea due to Gorelick (1975), which, previously, has only been applied to Hoare Logic dealing with recursive procedures :

1. By induction on the structure of an arbitrary program S, one establishes that a specific correctness formula $\mathrm{MGF}_{\mathrm{VDM}}(S)$, known as the *Most General Formula* (MGF), is derivable in the verification calculus.

2. Given the assumption $\models_{\mathrm{VDM}} \{p\}\, S\, \{q\}$, one may derive $\vdash_{\mathrm{VDM}} \{p\}\, S\, \{q\}$ by applying structural rules to $\vdash_{\mathrm{VDM}} \mathrm{MGF}_{\mathrm{VDM}}(S)$. All side-conditions which arise will be dealt with by the assumption.

In other words, instead of directly deriving $\models_{\mathrm{VDM}} \{p\}\, S\, \{q\} \Rightarrow \vdash_{\mathrm{VDM}} \{p\}\, S\, \{q\}$, one considers the stronger property $\vdash_{\mathrm{VDM}} \mathrm{MGF}_{\mathrm{VDM}}(S)$ for which induction goes through more easily. In particular, the direct proof cannot be applied when one considers recursive procedures, because the induction hypotheses are not strong enough.

The proposition $\vdash_{\mathrm{VDM}} \mathrm{MGF}_{\mathrm{VDM}}(S)$ asserts that, provided that one only considers input states in which the program S terminates, one may *derive* a correctness formula in which the postcondition relates all inputs with the appropriate outputs according to the underlying operational semantics of the programming language. At the semantic level, $\models_{\mathrm{VDM}} \mathrm{MGF}_{\mathrm{VDM}}(S)$ holds trivially.

➥ p. 149 **Definition 2.16 (Termination)** *For a deterministic programming language, commencing execution in a state σ, a program S terminates, if there is a final state τ according to the operational semantics,* $S{\downarrow}(\sigma) \stackrel{\mathrm{def}}{=} \exists \tau \cdot \sigma \xrightarrow{S} \tau$.

➥ p. 149 **Definition 2.17 (MGF)** $\mathrm{MGF}_{\mathrm{VDM}}(S) \stackrel{\mathrm{def}}{=} \{S{\downarrow}\}\ S\ \left\{\lambda(\tau,\sigma)\cdot\sigma \xrightarrow{S} \tau\right\}$

Having chosen a shallow embedding, we can directly express the assertions of the MGF. If one also considers syntax of the assertion language, one needs to show that it is sufficiently *expressive*. We will return to this issue in more detail in Section 2.12.

➥ p. 150 **Theorem 2.18 (MGF for VDM)** *For an arbitrary program S, $\vdash_{\mathrm{VDM}} \mathrm{MGF}_{\mathrm{VDM}}(S)$ is derivable.*

Proof The proof is by induction on the structure of S. Observe that this induction strategy affects both pre- and postcondition. In the case of a loop, from

$$\vdash_{\mathrm{VDM}} \{S{\downarrow}\}\ S\ \left\{\lambda(\tau,\sigma)\cdot\sigma \xrightarrow{S} \tau\right\} \tag{2.45}$$

we have to establish

$$\vdash_{\mathrm{VDM}} \{\textbf{while } b \textbf{ do } S{\downarrow}\}\ \textbf{while } b \textbf{ do } S\ \left\{\lambda(\tau,\sigma)\cdot\sigma \xrightarrow{\textbf{while } b \textbf{ do } S} \tau\right\} \quad . \tag{2.46}$$

We first apply the consequence rule (2.26) to the hypothesis (2.45) so that the precondition is of the form $\lambda\sigma \cdot p(\sigma) \wedge \mathrm{eval}(\sigma)(b) = \textbf{true}$ and the postcondition matches

$\lambda(\tau,\sigma)\cdot p(\tau)\wedge \mathit{sofar}(\tau,\sigma)$ for some suitable invariant p and variant *sofar*. We then apply the rule for loops and, finally, employ again the consequence rule to transform the correctness formula into the desired (2.46).

These transformations trigger some side-conditions relative to the variant and invariant to be synthesised. Applying the consequence rule for the first time yields

$$\forall\sigma\cdot\big(p(\sigma)\wedge \mathrm{eval}(\sigma)(b)=\mathbf{true}\big)\Rightarrow S{\downarrow}(\sigma) \tag{2.47}$$

$$\forall(\tau,\sigma)\cdot\big(p(\sigma)\wedge \mathrm{eval}(\sigma)(b)=\mathbf{true}\big)\Rightarrow \sigma\xrightarrow{\quad S\quad}\tau\Rightarrow \big(p(\tau)\wedge \mathit{sofar}(\tau,\sigma)\big) \tag{2.48}$$

The rule for loops is concerned with $\lambda(\tau,\sigma)\cdot \mathit{sofar}(\tau,\sigma)\wedge p(\tau)$ being well-founded and the last appeal to the rule of consequence triggers

$$\forall\sigma\cdot\mathbf{while}\ b\ \mathbf{do}\ S{\downarrow}(\sigma)\Rightarrow p(\sigma) \tag{2.49}$$

$$\forall(\tau,\sigma)\cdot\mathbf{while}\ b\ \mathbf{do}\ S{\downarrow}(\sigma)\Rightarrow \big(p(\tau)\wedge \mathrm{eval}(\tau)(b)=\mathbf{false}\wedge \mathit{sofar}^{=}(\tau,\sigma)\big)\Rightarrow \sigma\xrightarrow{\mathbf{while}\ b\ \mathbf{do}\ S}\tau \tag{2.50}$$

The termination condition for loops $p\stackrel{\mathrm{def}}{=}\mathbf{while}\ b\ \mathbf{do}\ S{\downarrow}$ is an appropriate invariant.

A single step of the loop is captured by the relation

$$\lambda(\sigma,\tau)\cdot \mathrm{eval}(\sigma)(b)=\mathbf{true}\wedge\sigma\xrightarrow{\quad S\quad}\tau\ .$$

Appealing to the operational semantics definition, it is easy to check that the transitive closure

$$\mathit{sofar}(\tau,\sigma)\stackrel{\mathrm{def}}{=}\big(\lambda(\sigma,\tau)\cdot \mathrm{eval}(\sigma)(b)=\mathbf{true}\wedge\sigma\xrightarrow{\quad S\quad}\tau\big)^{+}(\sigma,\tau)$$

approximates the loop. In particular, if τ is one of the intermediate states of the loop for which the loop guard does *not* hold, i.e., for some initial state σ, if $\mathit{sofar}^{=}(\tau,\sigma)$ and $\mathrm{eval}(\tau)(b)=\mathbf{false}$ holds, we must have $\sigma\xrightarrow{\mathbf{while}\ b\ \mathbf{do}\ S}\tau$. □

➥ p. 150 **Corollary 2.19 (Completeness of VDM)** *For any precondition p, program S and postcondition q, whenever $\models_{\mathrm{VDM}}\{p\}\ S\ \{q\}$ holds, $\vdash_{\mathrm{VDM}}\{p\}\ S\ \{q\}$ is derivable.*

It is instructive to study the completeness proof in order to appreciate the rôle of the main theorem 2.18:

Proof By Def. 2.12, the hypothesis is equivalent to

$$\forall \sigma \cdot p(\sigma) \Rightarrow \exists \tau \cdot \sigma \xrightarrow{S} \tau \wedge q(\tau, \sigma) \quad . \tag{2.51}$$

Applying the consequence rule (2.26) to the MGF i.e.,

$$\vdash_{\mathrm{VDM}} \{S\downarrow\}\ S \left\{ \lambda(\tau, \sigma) \cdot \sigma \xrightarrow{S} \tau \right\} \quad ,$$

we may derive the desired $\vdash_{\mathrm{VDM}} \{p\}\ S\ \{q\}$ provided we can satisfy the side conditions

$$\forall \sigma \cdot p(\sigma) \Rightarrow S\downarrow(\sigma) \tag{2.52}$$

and

$$\forall (\tau, \sigma) \cdot p(\sigma) \Rightarrow \sigma \xrightarrow{S} \tau \Rightarrow q(\tau, \sigma) \quad . \tag{2.53}$$

Weakening the precondition (2.52) is a direct consequence of (2.51). We may strengthen the postcondition as suggested in (2.53) due to determinism. □

2.10 Hoare Logic

Hoare Logic deviates from VDM in that the postcondition can only refer to the final and not to the initial state. We first present a set of rules for total correctness which reflects the traditional presentation. This includes a mechanised soundness and completeness result. Such meta-theoretical proofs are very similar to those in the VDM setting and we will merely highlight technical differences.

Using Hoare Logic, one cannot directly specify input/output specifications. In Sect. 2.10.3, we discuss various approaches to overcome this problem. Amongst these, we favour a proposal by Apt & Meertens (1980) to semantically consider assertions as relations between states and a domain of auxiliary variables. This idea has not previously been applied to Hoare Logic. It is useful in mechanising soundness and completeness of Hoare Logic, because it more adequately reflects the rôle of auxiliary variables employed in practice to relate output and input.

As our main contribution, we have designed a rule of consequence for adjusting auxiliary variables while strengthening preconditions and weakening postconditions. The new consequence rule is strictly stronger than Hoare's (1969) version. It plays a crucial rôle in establishing completeness as a corollary of the MGF theorem. Furthermore, it obviates the need for any further structural rules when considering recursive procedures.

In the traditional understanding of Hoare Logic (Cousot 1990), both pre- and post-conditions are first-order logic formulae where free variables denote the value of *program variables*. In particular, the validity of a formulae depends on a state σ.

For the purpose of mechanising soundness and completeness, we ignore the syntactical aspect of assertions. Employing again a shallow embedding, assertions are to inhabit the function space $\Sigma \to \texttt{Prop}$ which, intuitively, characterises predicates on states.

➥ p. 150 **Definition 2.20 (Semantics of Hoare Logic)** *Let*

$$\models_{\text{Hoare}} \{.\}\ .\ \{.\} \subseteq (\Sigma \to \texttt{Prop}) \times \texttt{prog} \times (\Sigma \to \texttt{Prop})$$

be a new judgement defined in terms of the operational semantics

$$\models_{\text{Hoare}} \{p\}\ S\ \{q\} \stackrel{\text{def}}{=} \forall \sigma \cdot p(\sigma) \Rightarrow \exists \tau \cdot \sigma \xrightarrow{\quad S \quad} \tau \wedge q(\tau)\ .$$

2.10.1 Verification Calculus

Based on work of Floyd (1967), Hoare (1969) proposed a verification calculus for partial correctness, now referred to as Hoare Logic, see Fig. 2.1. To ensure termination,

$$\{p\}\ \textbf{skip}\ \{p\}$$

$$\{p[x \mapsto t]\}\ x := t\ \{p\} \tag{2.54}$$

$$\frac{\{p\}\ S_1\ \{r\} \quad \{r\}\ S_2\ \{q\}}{\{p\}\ S_1;\ S_2\ \{q\}}$$

$$\frac{\{p \wedge b\}\ S_1\ \{q\} \qquad \{p \wedge \neg b\}\ S_2\ \{q\}}{\{p\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{q\}}$$

$$\frac{\{p \wedge b\}\ S\ \{p\}}{\{p\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{p \wedge \neg b\}} \tag{2.55}$$

$$\frac{\{p_1\}\ S\ \{q_1\}}{\{p\}\ S\ \{q\}} \qquad \text{provided } p \Rightarrow p_1 \text{ and } q_1 \Rightarrow q.$$

Figure 2.1: Hoare Logic for Partial Correctness – Syntactic Presentation

the rule for loops (2.55) needs to be modified. We introduce a termination measure u with $[\![u]\!] : \Sigma \to W$ rather than just u : expression(W). It may inspect *all* of the state space for some well-founded[4] domain $(W, <)$ which is decreased whenever the body

[4] As we have already observed in the case of VDM's rule for loops, for the special case of $W = \Sigma$, it suffices to show that the relation $\lambda(\tau, \sigma) \cdot p(\tau) \wedge \tau < \sigma$ is well-founded.

is executed:

$$\frac{\forall t : W \cdot \{p \wedge b \wedge u = t\}\ S\ \{p \wedge u < t\}}{\{p\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{p \wedge \neg b\}} \qquad \text{where } (W, <) \text{ is well-founded.}$$

A similar rule for verification calculi where postconditions may explicitly refer to the value of program variables in the initial state e.g., VDM, has been put forward by Manna & Pnueli (1974). Variants of this rule tailored for $W = \texttt{nat}$ (Harel 1980) or $W = \texttt{int}$ (Apt & Olderog 1991) have also been published previously. We prefer the well-founded version, because it simplifies the completeness proof without any impact on the soundness proof. One may instantiate $W = \Sigma$ and $u = \lambda\sigma \cdot \sigma$, and encounters almost the same proof obligations as in the case of VDM. It is well known that in practice, it is often easier to reason about termination using well-founded sets rather than being restricted to natural numbers (Dershowitz & Manna 1979).

Formalising these rules in a logical framework is as easy as in the case of VDM. In particular, representing the axiom for assignments (2.54) requires little work, because we chose a shallow embedding of assertions. Syntactically, the precondition amounts to updating a formulae at position x. We do not need to worry about formalising the notion of updating *assertions*. It can be expressed straightforwardly in terms of updating the state space i.e.,

$$[\![p\,[x \mapsto t]]\!]I(\sigma) = [\![p]\!]\big(I(\sigma[x \mapsto [\![t]\!]I(\sigma)])\big) \tag{2.56}$$

Previous experiments (Mason 1987, Homeier 1995) have made it clear that a deep embedding of assertions requires much more work. In particular, one would have to *prove* the substitution lemma (2.56) (Sieber 1981).

➥ p. 150 **Definition 2.21 (Hoare Logic)** *A verification calculus for Hoare Logic is defined as the least relation* $\vdash_{\text{Hoare}} \{.\}\ .\ \{.\} \subseteq (\Sigma \to \texttt{Prop}) \times \texttt{prog} \times (\Sigma \to \texttt{Prop})$ *satisfying*

$$\vdash_{\text{Hoare}} \{p\}\ \textbf{skip}\ \{p\}$$

$$\vdash_{\text{Hoare}} \big\{\lambda\sigma \cdot p\big(\sigma[x \mapsto \text{eval}(\sigma)(t)]\big)\big\}\ x := t\ \{p\}$$

$$\frac{\vdash_{\text{Hoare}} \{p\}\ S_1\ \{r\} \quad \vdash_{\text{Hoare}} \{r\}\ S_2\ \{q\}}{\vdash_{\text{Hoare}} \{p\}\ S_1;\ S_2\ \{q\}}$$

$$\frac{\begin{array}{c}\vdash_{\text{Hoare}} \{\lambda\sigma \cdot p(\sigma) \wedge \text{eval}(\sigma)(b) = \textbf{true}\}\ S_1\ \{q\} \\ \vdash_{\text{Hoare}} \{\lambda\sigma \cdot p(\sigma) \wedge \text{eval}(\sigma)(b) = \textbf{false}\}\ S_2\ \{q\}\end{array}}{\vdash_{\text{Hoare}} \{p\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{q\}}$$

$$\frac{\forall t : W \cdot\ \vdash_{\text{Hoare}}\ \begin{array}{c}\{\lambda\sigma \cdot p(\sigma) \wedge \text{eval}(\sigma)(b) = \textbf{true} \wedge u(\sigma) = t\} \\ S \\ \{\lambda\tau \cdot p(\tau) \wedge u(\tau) < t\}\end{array}}{\vdash_{\text{Hoare}} \{p\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{\lambda\tau \cdot p(\tau) \wedge \text{eval}(\tau)(b) = \textbf{false}\}}$$

where $(W, <)$ *is well-founded.* (2.57)

$$\frac{\vdash_{\text{Hoare}} \{p_1\}\ S\ \{q_1\}}{\vdash_{\text{Hoare}} \{p\}\ S\ \{q\}} \qquad \textit{provided } \forall\sigma \cdot p(\sigma) \Rightarrow p_1(\sigma) \textit{ and } \forall\tau \cdot q_1(\tau) \Rightarrow q(\tau). \quad (2.58)$$

2.10.2 Soundness and Completeness

➥ p. 151 **Theorem 2.22 (Soundness of Hoare Logic)** *The above verification calculus is sound.*

Proof by induction on the derivation of $\vdash_{\text{Hoare}} \{p\}\ S\ \{q\}$. □

➥ p. 151 **Definition 2.23 (Completeness of Hoare Logic)**
The above verification calculus is complete.

Proof by induction on the structure of programs. We concentrate on the case for loops. From the induction hypothesis

$$\forall p, q \cdot \models_{\text{Hoare}} \{p\}\ S\ \{q\} \Rightarrow \vdash_{\text{Hoare}} \{p\}\ S\ \{q\}$$

and the assumption

$$\models_{\text{Hoare}} \{p\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{q\} \quad (2.59)$$

we have to establish

$$\vdash_{\text{Hoare}} \{p\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{q\} \quad . \quad (2.60)$$

Our strategy is to use the rule for loops to connect the induction hypothesis with the conclusion. In particular, we have to come up with a suitable invariant and a termination measure $u : \Sigma \to W$ for some well-founded domain $(W, <)$. The proof is more challenging than establishing the MGF theorem[5], because one needs to worry about arbitrary assertions p and q. In general, the supplied precondition p cannot be expected to remain invariant. Instead, we choose

$$inv = \text{wp}(\textbf{while}\ b\ \textbf{do}\ S, q)$$

where, relative to a program and postcondition, the weakest precondition wp is defined as

$$\text{wp}(S, q)(\sigma) \stackrel{\text{def}}{=} \exists\tau \cdot \sigma \xrightarrow{\quad S \quad} \tau \wedge q(\tau) \quad .$$

It is easy to see that, for a loop, the weakest precondition[6] is bound to remain invariant.

[5] In the following section, we will strengthen the rule of consequence (2.58) so that one may establish completeness as a corollary of the MGF property.

[6] In verifying *concrete* programs, the problem of constructing suitable invariants stems from having to find syntactic representations of such predicates. It does not suffice to rely on a sufficiently expressive assertion language in which one may express weakest preconditions. Instead, one needs to find an invariant specific to the domain involved; otherwise the difficulty in verification of finding suitable invariants is merely propagated to proving more difficult side-conditions arising from applications of the consequence rule.

In finding a suitable termination measure, we borrow ideas from the corresponding proof in the context of VDM. We choose $W = \Sigma$, $u = \lambda\sigma \cdot \sigma$,

$$\tau < \sigma =$$
$$\textbf{while } b \textbf{ do } S{\downarrow}(\sigma) \wedge \left(\lambda(\sigma,\tau) \cdot \mathrm{eval}(\sigma)(b) = \textbf{true} \wedge \sigma \xrightarrow{\quad S \quad} \tau\right)^{+}(\sigma,\tau) \quad (2.61)$$

and aim to establish

$$\forall t : \Sigma \cdot \vdash_{\text{Hoare}} \{\lambda\sigma \cdot inv(\sigma) \wedge \mathrm{eval}(\sigma)(b) = \textbf{true} \wedge \sigma = t\}\ S\ \{\lambda\tau \cdot inv(\tau) \wedge \tau < t\} \quad (2.62)$$

Let t be an arbitrary state. Appealing to the induction hypothesis, it suffices to show the semantic counterpart

$$\models_{\text{Hoare}} \{\lambda\sigma \cdot inv(\sigma) \wedge \mathrm{eval}(\sigma)(b) = \textbf{true} \wedge \sigma = t\}\ S\ \{\lambda\tau \cdot inv(\tau) \wedge \tau < t\}$$

In other words, relative to an initial state σ, such that

$$\underbrace{\exists\tau \cdot \sigma \xrightarrow{\textbf{while } b \textbf{ do } S} \tau \wedge q(\tau)}_{inv(\sigma)} \qquad \text{and} \qquad \mathrm{eval}(\sigma)(b) = \textbf{true}$$

we have to find an intermediate state η such that

$$\sigma \xrightarrow{\quad S \quad} \eta \ ,$$
$$\underbrace{\exists\tau \cdot \eta \xrightarrow{\textbf{while } b \textbf{ do } S} \tau \wedge q(\tau)}_{inv(\eta)}$$

$$\text{and} \qquad \underbrace{\textbf{while } b \textbf{ do } S{\downarrow}(\sigma) \wedge \mathrm{eval}(\sigma)(b) = \textbf{true} \wedge \sigma \xrightarrow{\quad S \quad} \eta}_{\eta < \sigma}$$

These proof obligations can be resolved easily by appealing to the definition of the operational semantics. We may thus apply the rule for loops (2.57) to the correctness formula (2.62). This yields

$$\vdash_{\text{Hoare}} \qquad \left\{\lambda\sigma \cdot \exists\tau \cdot \sigma \xrightarrow{\textbf{while } b \textbf{ do } S} \tau \wedge q(\tau)\right\}$$
$$\textbf{while } b \textbf{ do } S$$
$$\left\{\lambda\eta \cdot \exists\tau \cdot \eta \xrightarrow{\textbf{while } b \textbf{ do } S} \tau \wedge q(\tau) \wedge \mathrm{eval}(\eta)(b) = \textbf{false}\right\} \quad .$$

According to the rule of consequence, we may derive the desired (2.60), provided the two side-conditions

$$\forall\sigma \cdot p(\sigma) \Rightarrow \exists\tau \cdot \sigma \xrightarrow{\textbf{while } b \textbf{ do } S} \tau \wedge q(\tau) \quad (2.63)$$

$$\forall\eta \cdot \left(\exists\tau \cdot \eta \xrightarrow{\textbf{while } b \textbf{ do } S} \tau \wedge q(\tau) \wedge \mathrm{eval}(\eta)(b) = \textbf{false}\right) \Rightarrow q(\eta) \quad (2.64)$$

hold. The first condition is identical to the assumption (2.59). In the second obligation, according to the definition of the operational semantics, the state τ must be equal to η in the case where the loop guard evaluates to **false** . □

2.10.3 Auxiliary Variables

Recall the specification of the exponential function in VDM

$$\{0 \leq y\}\ S_{\text{exp}}\ \left\{r = \overleftarrow{x}^{\overleftarrow{y}}\right\} \quad , \tag{2.65}$$

where all variables hold integer values. In Hoare Logic, we cannot directly relate the value of variables in the final with values of variables in the initial state. To overcome this deficiency, one needs to introduce auxiliary variables. These are simply program variables which only occur in assertions. In the precondition they freeze the value of other program variables so that in the postcondition one may refer to the initial value of program variables.

Let X and Y be program variables with sort `int`. In Hoare Logic, we may paraphrase the above correctness formula (2.65) as

$$\{0 \leq y \wedge x = X \wedge y = Y\}\ S_{\text{exp}}\ \left\{r = X^Y\right\}$$
$$\text{where } X \text{ and } Y \text{ must not occur free in } S_{\text{exp}}. \tag{2.66}$$

Notice that the side-condition is essential for the two correctness formulae (2.65) and (2.66) to be equivalent. However, in Hoare Logic, we may not derive correctness formulae with attached side conditions, because, formally, Hoare Logic is a calculus for deriving programs with pre- and postconditions only. Side-conditions may only appear within axioms and rules.

Strictly speaking, Hoare Logic is thus not adequate for verifying concrete programs with non-contrived specifications. Problems also arise when the program to be verified is not fully given. This is for example the case when one wants to

- develop programs hand-in-hand with their proofs of correctness (Gries 1981, Burstall & McKinna 1993)
- consider free procedures which are specified with respect to pre- and postcondition, but for which one does not consider a concrete implementation (Tarlecki 1985)
- verify programs invoking recursive procedures

Assume that S is a program about which we only know that the program variable X does not occur in it. Then, the two specifications

$$\{X = x\}\ S\ \{X = x\} \tag{2.67}$$

and

$$\{X = x+1\}\ S\ \{X = x+1\} \tag{2.68}$$

where all variables denote integer values both assert that the program S leaves x in-

variant. Whenever one can derive (2.67), one may derive (2.68) and vice versa. Nevertheless, the consequence rule (2.58) does not allow one to derive (2.68) from (2.67) or vice versa, as one would have to show $x = x + 1$.

Pragmatically, this is quite worrying. Taking software reuse seriously, imagine that (2.67) has been derived as part of the verification in a project. It is conceivable that in another project, one needs the equivalent correctness formula (2.68). Unfortunately, in such a situation, the formalism would force us to rebuild (2.68) from scratch. Of course in the new derivation one would benefit from analysing the first derivation, but such a methodology is nevertheless undesirable.

In Chapter 3, using recursive procedures, we show that precisely this problem occurs in trying to verify a program computing the factorial function. In a nutshell, similar to an inductive step, during the verification of a recursive procedure, one needs to unroll the recursive call and can exploit the correctness formula before unrolling. Unfortunately, the hypothesis only matches modulo shifts of auxiliary variables reminiscent of the difference between (2.67) and (2.68).

We need to strengthen the consequence rule and exploit the fact that some program variables are not affected by the program e.g.,

$$\frac{\{p_1\}\ S\ \{q_1\}}{\{p\}\ S\ \{q\}} \quad \text{provided } \forall X \cdot \exists t \cdot \big(\forall x \cdot p \Rightarrow (p_1\,[X \mapsto t])\big) \wedge \big(\forall x \cdot (q_1\,[X \mapsto t]) \Rightarrow q\big), \quad (2.69)$$

where x and X are two disjoint lists of all program variables occuring in any of the assertions and no variables from X appear in the program. Furthermore, t is a list of compatible expressions.

Such a consequnce rule is strictly stronger than Hoare's (1969), which one might view as a special case of 2.69 in which the choice of t is confined to X. In particular, employing the new rule of consequence, the two correctness formulae (2.67) and (2.68) are interderivable e.g., (2.68) follows from (2.67), because one may choose $t = X - 1$.

It also highlights that auxiliary variables and program variables deserve to be treated differently e.g., in the above side-condition, one only substitutes *auxiliary* variables. We would like to reflect this intuition in our encoding. We see two alternatives to capture auxiliary variables

1. explicit quantification
2. implicit quantification

Following Sieber (1985), we could extend Hoare Logic by allowing quantifiers at the *object* level i.e., correctness formulae may also take on the form of

$$\vdash_{\text{Hoare}} \forall X \cdot \{p\}\ S\ \{q\}$$

where the value of X does not depend on a state[7].

Universal quantification is one way to ensure that auxiliary variables are bound explicitly to a specific correctness formula. It binds all occurrences of auxiliary variables in the precondition, program and postcondition. We prefer an implicit quantification in which both pre- and postcondition characterise binary relations on states. The advantages over explicit quantification are that

- programs cannot mention auxiliary variables
- it reflects *current* software engineering practice and does not require a syntactic modification of correctness formulae
- it is easier to encode in a logical framework

At the syntactic level, one would need to (formally) distinguish between program variables and auxiliary variables[8]. One could for example enforce that program variables have to start with a lower-case letter, whereas auxiliary variables must start with an upper-case letter. To be well-formed, programs may only refer to program variables.

Semantically, program variables are, as before, interpreted according to the state space. However, auxiliary variables are interpreted freely. Let T be the domain of this interpretation. We revise our shallow embedding of assertions:

➥ p. 152 **Definition 2.24 (Assertions)** $\text{Assertion}(T : \texttt{Type}) \stackrel{\text{def}}{=} (T \times \Sigma) \to \texttt{Prop}$

Example 2.25 *Let $T = \{X, Y\} \to \texttt{int}$. Relative to an interpretation $Z : T$ and a state σ, we interpret*

$$[\![0 \leq y \wedge x = X \wedge y = Y]\!](Z, \sigma) = 0 \leq \sigma(y) \wedge \sigma(x) = Z(X) \wedge \sigma(y) = Z(Y) \quad .$$

The universal quantification is then integrated into the semantics:

➥ p. 152 **Definition 2.26 (Semantics of Hoare Logic)** *Parametrised by an arbitrary type T, let $\models_{\text{Hoare}} \{.\}\ .\ \{.\} \subseteq \text{Assertion}(T) \times \texttt{prog} \times \text{Assertion}(T)$ be a new judgement defined in terms of the operational semantics*

$$\models_{\text{Hoare}} \{p\}\ S\ \{q\} \stackrel{\text{def}}{=} \forall Z \cdot \forall \sigma \cdot p(Z, \sigma) \Rightarrow \exists \tau \cdot \sigma \xrightarrow{\quad S \quad} \tau \wedge q(Z, \tau) \quad .$$

[7] In Reynolds's (1982) specification logic, X may be an arbitrary *expression* and one needs to additionaly assert that X is not affected by the program S.

[8] This has also been proposed by Homeier & Martin (1996).

The previous axioms and rules from Def. 2.21 can be reused without any syntactic changes when assertions are represented as relations rather than predicates. Furthermore, we can easily formally cater for the stronger rule of consequence (2.69):

$$\frac{\vdash_{\text{Hoare}} \{p_1\}\ S\ \{q_1\}}{\vdash_{\text{Hoare}} \{p\}\ S\ \{q\}} \quad \text{provided } \forall Z \cdot \exists Z_1 \cdot \big(\forall \sigma \cdot p(Z,\sigma) \Rightarrow p_1(Z_1,\sigma)\big) \wedge \big(\forall \tau \cdot q_1(Z_1,\tau) \Rightarrow q(Z,\tau)\big)$$

where $p_1, q_1 : \text{Assertion}(T)$ and $p, q : \text{Assertion}(U)$ are assertions for not necessarily identical domains T and U. In deriving completeness as a corollary from the MGF property, besides being able to switch the domain of auxiliary variables, we also need a weaker side-condition in which the choice of Z_1 may additionally depend on the initial value of variables:

$$\frac{\vdash_{\text{Hoare}} \{p_1\}\ S\ \{q_1\}}{\vdash_{\text{Hoare}} \{p\}\ S\ \{q\}}$$
where $p_1, q_1 : \text{Assertion}(U)$ and $p, q : \text{Assertion}(T)$ for arbitrary types T and U
$$\text{and } \forall Z \cdot \forall \sigma \cdot p(Z,\sigma) \Rightarrow \Big(\exists Z_1 \cdot p_1(Z_1,\sigma) \wedge \big(\forall \tau \cdot q_1(Z_1,\tau) \Rightarrow q(Z,\tau)\big)\Big). \quad (2.70)$$

In classical logic, $\forall Z \cdot \forall \sigma \cdot \exists Z_1 \cdot \big(p(Z,\sigma) \Rightarrow p_1(Z_1,\sigma)\big) \wedge \big(\forall \tau \cdot q_1(Z_1,\tau) \Rightarrow q(Z,\tau)\big)$ is an equivalent side-condition. Constructively, it is stronger and not sufficient. We will present a further variant on page 71 in connection with the rule of adaptation. However, the above consequence rule (2.70) suffices for all examples and completeness proofs in this thesis. In moving from predicates to binary relations, we retain all other axioms and rules:

➥ p. 152 **Definition 2.27 (Hoare Logic)** *A verification calculus for Hoare Logic which takes auxiliary variables seriously is defined as the least relation*

$$\vdash_{\text{Hoare}} \{.\}\ .\ \{.\} \subseteq \text{Assertion}(T) \times \texttt{prog} \times \text{Assertion}(T)$$

indexed by an arbitrary type T such that

$$\vdash_{\text{Hoare}} \{p\}\ \textbf{skip}\ \{p\} \quad (2.71)$$

$$\vdash_{\text{Hoare}} \big\{\lambda(Z,\sigma) \cdot p\big(Z, \sigma[x \mapsto \text{eval}(\sigma)(t)]\big)\big\}\ x := t\ \{p\} \quad (2.72)$$

$$\frac{\vdash_{\text{Hoare}} \{p\}\ S_1\ \{r\} \quad \vdash_{\text{Hoare}} \{r\}\ S_2\ \{q\}}{\vdash_{\text{Hoare}} \{p\}\ S_1;\ S_2\ \{q\}}$$

$$\frac{\vdash_{\text{Hoare}} \{\lambda(Z,\sigma) \cdot p(Z,\sigma) \wedge \text{eval}(\sigma)(b) = \textbf{true}\}\ S_1\ \{q\} \quad \vdash_{\text{Hoare}} \{\lambda(Z,\sigma) \cdot p(Z,\sigma) \wedge \text{eval}(\sigma)(b) = \textbf{false}\}\ S_2\ \{q\}}{\vdash_{\text{Hoare}} \{p\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{q\}}$$

$$\frac{\forall t : W \cdot \vdash_{\text{Hoare}} \{\hat{p}\}\ S\ \{\lambda(Z,\tau) \cdot p(Z,\tau) \wedge u(\tau) < t\}}{\vdash_{\text{Hoare}} \{p\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{\lambda(Z,\tau) \cdot p(Z,\tau) \wedge \text{eval}(\tau)(b) = \textbf{false}\}}$$
$$\textit{where } \hat{p}(Z,\sigma) \stackrel{\text{def}}{=} p(Z,\sigma) \wedge \text{eval}(\sigma)(b) = \textbf{true} \wedge u(\sigma) = t$$
$$\textit{and } (W,<) \textit{ is well-founded.} \quad (2.73)$$

$$\frac{\vdash_{\text{Hoare}} \{p_1\}\ S\ \{q_1\}}{\vdash_{\text{Hoare}} \{p\}\ S\ \{q\}}$$
$$\textit{where } p_1, q_1 : \text{Assertion}(U) \textit{ and } p, q : \text{Assertion}(T) \textit{ for arbitrary types } T \textit{ and } U$$
$$\textit{and } \forall Z \cdot \forall \sigma \cdot p(Z,\sigma) \Rightarrow \left(\exists Z_1 \cdot p_1(Z_1,\sigma) \wedge \left(\forall \tau \cdot q_1(Z_1,\tau) \Rightarrow q(Z,\tau) \right) \right). \quad (2.70)$$

2.10.3.1 Universal Quantification versus Auxiliary Variables

Notice that the universally quantified variable in the rule for loops (2.73) plays the same rôle as an auxiliary variable. It must not occur[9] in the program S. In particular, the rule

$$\frac{\{\hat{p}\}\ S\ \{\lambda((Z,t),\tau) \cdot p(Z,\tau) \wedge u(\tau) < t\}}{\{p\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{\lambda(Z,\tau) \cdot p(Z,\tau) \wedge \text{eval}(\tau)(b) = \textbf{false}\}}$$
$$\text{where } \hat{p}((Z,t),\sigma) \stackrel{\text{def}}{=} p(Z,\sigma) \wedge \text{eval}(\sigma)(b) = \textbf{true} \wedge u(\sigma) = t$$
$$\text{and } (W,<) \text{ is well-founded.}$$

is (semantically) equivalent.

We have chosen the version (2.73) with universal quantification to illuminate the similarities with the rule for procedure calls in the following chapter in which the universal quantification ranges over *two* correctness formulae and cannot be internalised as an auxiliary variable. Furthermore, the implicitly quantified version needs to extend the domain of auxiliary variables in the premiss to guarantee that t is fresh. Hence, one could not restrict the domain of auxiliary variables to the state space.

In practice, rules with universal quantifications in the *premiss* pose no problems. In a refinement proof, one continues deriving the pure Hoare triple in which the quantified variable is replaced by a constant.

2.10.3.2 Soundness

We refer the reader to Fig. 2.2 on the next page for a syntactic presentation of this calculus. There, the side condition of the rule of consequence is tailored to classical logic and x denotes the list of all program variables, whereas Z denotes the list of all auxiliary variables.

[9] in our formalisation in LEGO, S is universally quantified outside the scope of t.

$$\{p\}\ \textbf{skip}\ \{p\}$$

$$\{p[x \mapsto t]\}\ x := t\ \{p\}$$

$$\frac{\{p\}\ S_1\ \{r\} \qquad \{r\}\ S_2\ \{q\}}{\{p\}\ S_1;\ S_2\ \{q\}}$$

$$\frac{\{p \wedge b\}\ S_1\ \{q\} \qquad \{p \wedge \neg b\}\ S_2\ \{q\}}{\{p\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{q\}}$$

$$\frac{\forall t : W \cdot \{p \wedge b \wedge u = t\}\ S\ \{p \wedge u < t\}}{\{p\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{p \wedge \neg b\}} \qquad \text{where } (W, <) \text{ is well-founded.}$$

$$\frac{\{p_1\}\ S\ \{q_1\}}{\{p\}\ S\ \{q\}}$$
provided $\forall Z \cdot \forall x \cdot \exists Z_1 \cdot \big(p \Rightarrow (p_1[Z \mapsto Z_1])\big) \wedge \big(\forall x \cdot (q_1[Z \mapsto Z_1]) \Rightarrow q\big)$.

Figure 2.2: Hoare Logic for Total Correctness – Syntactic Presentation

➥ p. 154 **Theorem 2.28 (Soundness of Hoare Logic)** *The above verification calculus is sound.*

Proof by induction on the derivation of $\vdash_{\text{Hoare}} \{p\}\ S\ \{q\}$. If the rule of consequence (2.70) has been applied last, from the induction hypothesis $\models_{\text{Hoare}} \{p_1\}\ S\ \{q_1\}$ i.e.,

$$\forall Z_1 \cdot \forall \sigma \cdot p_1(Z_1, \sigma) \Rightarrow \exists \tau \cdot \sigma \xrightarrow{\quad S \quad} \tau \wedge q_1(Z_1, \tau) \tag{2.74}$$

and the side-condition

$$\forall Z \cdot \forall \sigma \cdot p(Z, \sigma) \Rightarrow \Big(\exists Z_1 \cdot p_1(Z_1, \sigma) \wedge \big(\forall \tau \cdot q_1(Z_1, \tau) \Rightarrow q(Z, \tau)\big) \Big) \tag{2.75}$$

we need to show that $\models_{\text{Hoare}} \{p\}\ S\ \{q\}$ holds.

Given an auxiliary variable Z and an initial state σ satisfying $p(Z, \sigma)$, we need to find a final state τ such that $\sigma \xrightarrow{\quad S \quad} \tau$ and $q(Z, \tau)$. Combining $p(Z, \sigma)$ and (2.75), we can extract a witness Z_1 such that $p_1(Z_1, \sigma)$ holds, and for any state τ, we may reduce the task of showing $q(Z, \tau)$ to $q_1(Z_1, \tau)$. Appealing to (2.74), from $p_1(Z_1, \sigma)$ we may infer that there is a state τ satisfying both $\sigma \xrightarrow{\quad S \quad} \tau$ and $q_1(Z_1, \tau)$. □

2.10.3.3 Completeness

Having represented assertions as binary relations over an arbitrary domain and the state space, we can immediately formulate the MGF property for Hoare Logic. Picking the

state space as the domain of auxiliary variables, we could simulate

$$\vdash_{\mathrm{VDM}} \{S\downarrow\}\ S\ \left\{\lambda(\tau,\sigma)\cdot\sigma \xrightarrow{\quad S\quad} \tau\right\}$$

by

$$\vdash_{\mathrm{Hoare}} \{\lambda(Z,\sigma)\cdot\sigma = Z \wedge S\downarrow(\sigma)\}\ S\ \left\{\lambda(Z,\tau)\cdot Z \xrightarrow{\quad S\quad} \tau\right\}\ .$$

However, if one exploits the auxiliary variable to capture the final rather than the initial state, one has the more elegant

➥ p. 154

Definition 2.29 (MGF)

$$\mathrm{MGF}_{\mathrm{Hoare}}(S) \stackrel{\mathrm{def}}{=} \left\{\lambda(Z,\sigma)\cdot\sigma \xrightarrow{\quad S\quad} Z\right\}\ S\ \{\lambda(Z,\tau)\cdot Z = \tau\}$$

Notice that the precondition coincides with the weakest precondition relative to the program S and postcondition $\lambda(Z,\tau)\cdot Z = \tau$.

➥ p. 154

Theorem 2.30 (MGF for Hoare Logic) *For an arbitrary program S, one may derive* $\vdash_{\mathrm{Hoare}} \mathrm{MGF}_{\mathrm{Hoare}}(S)$.

Proof by induction on the structure of S. In the case of a loop, from the induction hypothesis

$$\vdash_{\mathrm{Hoare}} \left\{\lambda(Z,\sigma)\cdot\sigma \xrightarrow{\quad S\quad} Z\right\}\ S\ \{\lambda(Z,\tau)\cdot Z = \tau\} \tag{2.76}$$

we have to establish

$$\vdash_{\mathrm{Hoare}} \left\{\lambda(Z,\sigma)\cdot\sigma \xrightarrow{\textbf{while } b \textbf{ do } S} Z\right\}\ \textbf{while } b \textbf{ do } S\ \{\lambda(Z,\tau)\cdot Z = \tau\}\ . \tag{2.77}$$

The proof follows the structure of the same property for VDM on page 27. To exploit the rule for loops, we want to transform the hypothesis into the form

$$\vdash_{\mathrm{Hoare}} \{\hat{p}\}\ S\ \{\lambda(Z,\tau)\cdot p(Z,\tau) \wedge u(\tau) < t\}$$

where $t : \Sigma$ is a constant and the termination measure $u = \lambda\sigma\cdot\sigma$ is the identity function. For the invariant, we choose

$$p(Z,\sigma) = \sigma \xrightarrow{\textbf{while } b \textbf{ do } S} Z$$

and for the variant $<$, we reuse the well-founded relation (2.61) employed in the direct proof of completeness on page 33.

This transformation is sanctioned by the rule of consequence (2.70). As a side-condition, for arbitrary states σ and Z, given

$$\sigma \xrightarrow{\textbf{while } b \textbf{ do } S} Z \tag{2.78}$$

$$\mathrm{eval}(\sigma)(b) = \textbf{true} \tag{2.79}$$

we need to find a state Z_1 such that $\sigma \xrightarrow{S} Z_1$ and

$$\forall \tau \cdot (Z_1 = \tau) \Rightarrow \left(\tau \xrightarrow{\textbf{while } b \textbf{ do } S} Z_1 \wedge \sigma < t\right) \ .$$

Inverting the derivation of (2.78) in the presence of (2.79), these obligations can be resolved easily.

To conclude the proof, we need to again apply the rule of consequence to mediate between the conclusion of the loop rule

$$\vdash_{\text{Hoare}} \{p\}\ \textbf{while } b \textbf{ do } S\ \{\lambda(Z,\tau) \cdot p(Z,\tau) \wedge \mathrm{eval}(\tau)(b) = \textbf{false}\}$$

and the desired correctness formula (2.77). In this case, for arbitrary states σ and Z such that $\sigma \xrightarrow{\textbf{while } b \textbf{ do } S} Z$, we have find a state Z_1 such that

$$\sigma \xrightarrow{\textbf{while } b \textbf{ do } S} Z_1 \tag{2.80}$$

$$\forall \tau \cdot \left(\tau \xrightarrow{\textbf{while } b \textbf{ do } S} Z_1 \wedge \mathrm{eval}(\tau)(b) = \textbf{false}\right) \Rightarrow Z_1 = \tau \tag{2.81}$$

We resolve (2.80) by selecting $Z_1 = Z$, whereas (2.81) is a direct consequence of the operational semantics. □

Having treated auxiliary variables in a rigorous manner, it is interesting to observe that the main Theorem 2.30 singles out *the set of all states* as the domain for auxiliary variables i.e., $T = \Sigma$. Intuitively, such a restriction ties the auxiliary variables together with the program variables. Let $\{x_1, \ldots, x_n\}$ be the set of all program variables with corresponding types $\{T_1, \ldots, T_n\}$ occurring in some program S. An assertion of the form

$$\lambda\left(Z : T_1 \times \cdots \times T_n, \sigma : \Sigma\right) \cdot P(Z.1, \ldots, Z.n, \sigma)$$

is equivalent to

$$\lambda(Z, \sigma : \Sigma) \cdot P(Z(x_1), \ldots, Z(x_n), \sigma) \ .$$

➥ p. 154 **Corollary 2.31 (Completeness of Hoare Logic)** *For any precondition p, program S and postcondition q, whenever $\models_{\text{Hoare}} \{p\}\ S\ \{q\}$ holds, then $\vdash_{\text{Hoare}} \{p\}\ S\ \{q\}$ is derivable.*

Proof Let S be a program and $p, q : \text{Assertion}(T)$ be assertions for an *arbitrary* domain T. Given $\models_{\text{Hoare}} \{p\}\ S\ \{q\}$ i.e.,

$$\forall Z \cdot \forall \sigma \cdot p(Z, \sigma) \Rightarrow \exists \tau \cdot \sigma \xrightarrow{\quad S \quad} \tau \wedge q(Z, \tau) \tag{2.82}$$

we need to establish $\vdash_{\text{Hoare}} \{p\}\ S\ \{q\}$.

By the consequence rule (2.70), we may directly infer $\vdash_{\text{Hoare}} \{p\}\ S\ \{q\}$ from the MGF, triggering the side condition

$$\forall Z \cdot \forall \sigma \cdot p(Z, \sigma) \Rightarrow \exists Z_1 \cdot \sigma \xrightarrow{\quad S \quad} Z_1 \wedge \forall \tau \cdot \tau = Z_1 \Rightarrow q(Z, \tau) \ . \tag{2.83}$$

Clearly, (2.82) implies (2.83). □

We learn from the proof that in order to derive the correctness of a specification using an arbitrary domain of free variables T, applying the new consequence rule will confine the domain of auxiliary variables to the state space Σ. In particular, notice that the consequence rule has mediated between the state space and the domain T as representing auxiliary variables. Implementors of computer-aided verification systems can employ this transformation and safely restrict the domain of auxiliary variables to the state space for all other rules. Alternatively, one might implement a system in which the user can only formulate specifications where auxiliary variables live in the state space. Again, the above proof suggests that this is a sensible approach. In fact, in the system VDM, such a restriction has essentially been made.

Furthermore, notice that in the completeness proof, we only needed to apply the consequence rule (2.70). Thus, completeness always follows as a corollary of the MGF theorem, irrespective of any programming language extensions.

2.11 Hoare Logic subsumes VDM

In the traditional understanding of Hoare Logic, assertions characterise *predicates* on states. VDM is more flexible in that postconditions may, in addition to the final state, also refer to the initial state. In that respect, one may argue that VDM subsumes Hoare Logic.

However, we question such a view, because, pragmatically, in Hoare Logic, one uses auxiliary variables to relate output with input. Thus, one may choose an *arbitrary* reference point. In particular, during a derivation of a correctness formula $\{p\}\ S\ \{q\}$ in Hoare Logic, both pre- and postcondition may refer to the value of program variables before executing S. VDM is more restrictive. In a derivation, the rule of sequential composition enforces that, in order to derive $\{p\}\ S_1; S_2\ \{q\}$ one must adapt the point of reference so that the postcondition for S_2 is expressed relative to some intermediate

state. We should emphasise that this argument reflects the *intended* use of both systems. Of course, one could mirror Hoare Logic derivations in VDM by also employing auxiliary variables. But notice that one would then have to also improve VDM's rule of consequence to simulate our version for Hoare Logic.

As a novelty, we show that Hoare Logic subsumes VDM in the sense that an arbitrary derivation in VDM can be embedded in Hoare Logic. In particular, every correctness formulae in the VDM derivation is uniformly translated and every decomposition rule in VDM is simulated by the corresponding rule in Hoare Logic where both premiss and conclusion may have to be adjusted by our new rule of consequence.

This result hinges on a rigorous treatment of auxiliary variables in Hoare Logic and the new rule of consequence. In particular, VDM's consequence rule can be simulated by our rule of consequence, but not Hoare's (1969).

➥ p. 155 **Theorem 2.32 (Embedding VDM Correctness Formulae in Hoare Logic)**
Let $p \subseteq \Sigma$ and $q \subseteq \Sigma \times \Sigma$ be assertions and S an arbitrary program. Whenever

$$\vdash_{\text{VDM}} \{p\}\ S\ \{q\}$$

is derivable, we may also establish

$$\vdash_{\text{Hoare}} \{\lambda(Z,\sigma)\cdot\sigma = Z \wedge p(\sigma)\}\ S\ \{\lambda(Z,\tau)\cdot q(\tau,Z)\} \quad .$$

By appealing to soundness and completeness, it is evident that for every derivation in VDM, one can reconstruct a proof of the corresponding correctness formula in Hoare Logic:

Proof Assume $\vdash_{\text{VDM}} \{p\}\ S\ \{q\}$. By soundness of VDM, we have

$$\forall\sigma\cdot p(\sigma) \Rightarrow \exists\tau\cdot\sigma \xrightarrow{\quad S \quad} \tau \wedge q(\tau,\sigma)$$

which is equivalent to

$$\forall Z\cdot\forall\sigma\cdot\big(\sigma = Z \wedge p(\sigma)\big) \Rightarrow \exists\tau\cdot\sigma \xrightarrow{\quad S \quad} \tau \wedge q(\tau,Z) \quad .$$

Hence, by completeness of Hoare Logic, we may derive

$$\vdash_{\text{Hoare}} \{\lambda(Z,\sigma)\cdot\sigma = Z \wedge p(\sigma)\}\ S\ \{\lambda(Z,\tau)\cdot q(\tau,Z)\} \quad .$$

□

However, we are primarily interested in the *structure* of the translation. The following constructive proof shows explicitly how to transform VDM into Hoare Logic derivations. Notice that the proof transformation is compositional!

Proof by induction on the derivation of $\vdash_{\text{VDM}} \{p\}\ S\ \{q\}$. We consider the interesting cases where the last rule applied has been the rule for sequential composition, loops, or the rule of consequence.

Sequential Composition From the induction hypotheses

$$\vdash_{\text{Hoare}} \{\lambda(Z,\sigma)\cdot\sigma = Z \wedge p_1(\sigma)\}\ S_1\ \{\lambda(Z,\tau)\cdot p_2(\tau)\wedge r_1(\tau,Z)\} \tag{2.84}$$

and

$$\vdash_{\text{Hoare}} \{\lambda(Z,\sigma)\cdot\sigma = Z \wedge p_2(\sigma)\}\ S_2\ \{\lambda(Z,\tau)\cdot r_2(\tau,Z)\} \tag{2.85}$$

we have to show

$$\vdash_{\text{Hoare}} \{\lambda(Z,\sigma)\cdot\sigma = Z \wedge p_1(\sigma)\}\ S_1;\ S_2\ \{\lambda(Z,\tau)\cdot r_2\circ r_1(\tau,Z)\}\quad .$$

We employ the strengthened rule of consequence to shift the reference point of the second assumption (2.85) to the initial state of the *compound* statement, yielding

$$\vdash_{\text{Hoare}} \{\lambda(Z,\sigma)\cdot p_2(\sigma)\wedge r_1(\sigma,Z)\}\ S_2\ \{\lambda(Z,\tau)\cdot r_2\circ r_1(\tau,Z)\} \tag{2.86}$$

This transformation triggers the proof obligation

$$\forall Z,\sigma\cdot\big(p_2(\sigma)\wedge r_1(\sigma,Z)\big)\Rightarrow \exists Z_1\cdot\big(\sigma = Z_1\wedge p_2(\sigma)\big)\wedge\big(\forall\tau\cdot r_2(\tau,Z_1)\Rightarrow(r_2\circ r_1)(\tau,Z)\big)$$

which can be resolved easily when choosing $Z_1 = \sigma$. Notice that Z_1 has to capture the value of all program variables in the initial state. In particular, neither Hoare's (1969) nor the draft rule of consequence (2.69) would have backed up the required transformation.

Finally, applying the rule of sequential composition to the correctness formulae (2.84) and (2.86) completes the case.

Loops From the induction hypothesis

$$\vdash_{\text{Hoare}}\quad \begin{array}{c}\{\lambda(Z,\sigma)\cdot\sigma = Z\wedge p(\sigma)\wedge \text{eval}(\sigma)(b) = \textbf{true}\}\\ S\\ \{\lambda(Z,\tau)\cdot p(\tau)\wedge \mathit{sofar}(\tau,Z)\}\end{array} \tag{2.87}$$

we have to show

$$\vdash_{\text{Hoare}} \quad \begin{array}{c} \{\lambda(Z,\sigma)\cdot \sigma = Z \wedge p(\sigma)\} \\ \textbf{while } b \textbf{ do } S \\ \{\lambda(Z,\tau)\cdot p(\tau) \wedge \text{eval}(\tau)(b) = \textbf{false} \wedge \mathit{sofar}^{=}(\tau,Z)\} \end{array} \tag{2.88}$$

where *sofar* is transitive and $\lambda(\tau,\sigma)\cdot \mathit{sofar}(\tau,\sigma) \wedge p(\tau)$ is well-founded.

VDM's rule for loops is stronger than the well-founded version (2.73), because in the postcondition of the conclusion, the rule for VDM additionally records information regarding the variant *sofar*. To compensate, instead of p, we use the weaker invariant

$$\mathit{inv}(Z,\sigma) = \mathit{sofar}^{=}(\sigma,Z) \wedge p(\sigma) \quad .$$

Another difficulty arises because we have to express termination relative to a new constant t. Again, it is up to the strengthened rule of consequence to reconcile the differences between the hypothesis (2.87) and

$$\vdash_{\text{Hoare}} \quad \begin{array}{c} \{\lambda(Z,\sigma)\cdot \mathit{inv}(Z,\sigma) \wedge \text{eval}(\sigma)(b) = \textbf{true} \wedge \sigma = t\} \\ S \\ \{\lambda(Z,\tau)\cdot \mathit{inv}(Z,\tau) \wedge \mathit{sofar}(\tau,t)\} \end{array} \quad . \tag{2.89}$$

We first deal with the proof obligation triggered by the side-condition of the rule of consequence. For arbitrary states Z, σ such that

$$\mathit{inv}(Z,\sigma) \quad , \tag{2.90}$$

$\text{eval}(\sigma)(b) = \textbf{true}$ and $\sigma = t$, we have to find a state Z_1 such that $\sigma = Z_1$, $p(\sigma)$, $\text{eval}(\sigma)(b) = \textbf{true}$ and

$$\forall\tau\cdot \big(p(\tau) \wedge \mathit{sofar}(\tau,Z_1)\big) \Rightarrow \big(\mathit{inv}(Z,\tau) \wedge \mathit{sofar}(\tau,t) \wedge p(\tau)\big) \tag{2.91}$$

holds. Obviously, we have to chose $Z_1 = \sigma$. We consider (2.91) in more detail. Let τ be an arbitrary state such that $p(\tau)$ and

$$\mathit{sofar}(\tau,\sigma). \tag{2.92}$$

Besides $p(\tau)$ and $\mathit{sofar}(\tau,\sigma)$, we have to establish

$$\mathit{sofar}^{=}(\tau,Z) \quad . \tag{2.93}$$

We resolve (2.93) by appealing to the transitivity of *sofar* in the presence of (2.92) and (2.90).

Applying the rule for loops to (2.89) produces

$$\vdash_{\text{Hoare}} \quad \begin{array}{c} \{inv\} \\ \textbf{while } b \textbf{ do } S \\ \{\lambda(Z,\tau)\cdot inv(Z,\tau) \wedge \text{eval}(\tau)(b) = \textbf{false}\} \end{array} \quad .$$

Employing the rule of consequence to weaken the precondition and strengthen the postcondition leads directly to the desired (2.88).

Consequence VDM's consequence rule can be seen as an instance of our strengthened consequence rule for Hoare Logic. From the induction hypothesis

$$\vdash_{\text{Hoare}} \{\lambda(Z,\sigma)\cdot \sigma = Z \wedge p_1(\sigma)\}\ S\ \{\lambda(Z,\tau)\cdot q_1(\tau,Z)\}$$

and the side-conditions $p \subseteq p_1$ and $\forall\sigma,\tau\cdot p(\sigma) \Rightarrow q_1(\sigma,\tau) \Rightarrow q(\sigma,\tau)$, we have to derive $\vdash_{\text{Hoare}} \{\lambda(Z,\sigma)\cdot \sigma = Z \wedge p(\sigma)\}\ S\ \{\lambda(Z,\tau)\cdot q(\tau,Z)\}$. From the side-conditions we may conclude

$$\forall Z,\sigma\cdot \sigma = Z \wedge p(\sigma) \Rightarrow \exists Z_1\cdot \sigma = Z_1 \wedge p_1(\sigma) \wedge \forall\tau\cdot q_1(\tau,Z_1) \Rightarrow q(\tau,Z) \quad .$$

Thus, our rule of consequence leads also to the desired correctness formula.

□

There is a one-to-one correspondence between the rules of Hoare-style $\vdash_{\text{Hoare}}$ and VDM's decomposition rules $\vdash_{\text{VDM}}$. According to Theorem 2.32, specifications in VDM correspond to a particular class of specification in Hoare Logic, in which the auxiliary variables are devoted to freezing the values of all program variables prior to execution.

The additional flexibility available in Hoare Logic is to blame for the more elaborate consequence rule, whereas the hard-wired perspective between pre- and postconditions in VDM is responsible for a more complicated sequential rule. Ultimately, it is a matter of taste which framework to use.

2.12 Completeness: What does it really mean?

In this section, we take a closer look at the meaning of completeness for verification calculi. We introduce Cook's (1978) notion of *relative completeness* and critically review some of the incompleteness results in the literature. We restrict our discussion to Hoare Logic. The case for VDM is similar.

It is unrealistic to expect than one can design a verification calculus in which one can *derive* all *valid* correctness formulae. As a prerequisite for being complete, the

underlying language in which one formulates assertions must be sufficiently expressive so than one may formulate all intermediate assertions required in the derivation, such as invariants. In particular, for an arbitrary postcondition and program, the assertion language must be able to express the weakest precondition[10]

Definition 2.33 (Weakest Precondition) *Let S be an arbitrary program and $q \subseteq \Sigma$ be a set of states.*

$$\mathrm{wp}(S,q) \stackrel{\text{def}}{=} \left\{\sigma \;\middle|\; \exists\tau \cdot \left(\sigma \xrightarrow{\quad S \quad} \tau\right) \wedge (\tau \in q)\right\}$$

As soon as one considers languages with more interesting features such as loops, closure under weakest preconditions leads to logics which, by Gödels's incompleteness theorem, cannot themselves be complete.

To avoid this problem, Cook has proposed that one investigates *relative completeness* in an attempt to separate the reasoning about programs from the reasoning about the underlying logical language. One only considers expressive first-order logics. Furthermore, rules of the verification calculus may be applied in a derivation if the logical side-condition is valid rather than derivable. In particular, completeness no-longer compares a proof-theoretic with a model-theoretic account.

Expressiveness and relative completeness are defined relative to an interpretation I which itself depends on a state σ. Variables are interpreted according to the value of program variables recorded in σ whereas the interpretation of constants e.g., 0, $+$, $\wedge$, may vary from interpretation to interpretation. Following the presentation in (Cousot 1990), we characterise expressive interpretations by the following definition:

Definition 2.34 (Expressiveness) *The interpretation I is said to be expressive for the programming language* `prog` *and the assertion language L if and only if*

$$\forall S \cdot \forall q \cdot \exists p \cdot \forall\sigma \cdot [\![p]\!]I(\sigma) \Leftrightarrow \sigma \in \mathrm{wp}\big(S, \lambda\tau \cdot [\![q]\!]I(\tau)\big)$$

For the purpose of this section, we redefine validity and derivability of correctness formulae $\{p\}\ S\ \{q\} \subseteq L \times \texttt{prog} \times L$ relative to an arbitrary interpretation I.

Definition 2.35 (Semantics of Hoare Logic)

$$\models^{I}_{\text{Hoare}} \{p\}\ S\ \{q\} \stackrel{\text{def}}{=} \forall\sigma \cdot [\![p]\!](I(\sigma)) \Rightarrow \exists\tau \cdot \sigma \xrightarrow{\quad S \quad} \tau \wedge [\![q]\!](I(\tau))$$

Definition 2.36 (Hoare Logic) *Let $\vdash^{I}_{\text{Hoare}} \{.\}\ .\ \{.\}$ be the least relation satisfying*

$$\vdash^{I}_{\text{Hoare}} \{p\}\ \textbf{skip}\ \{p\}$$

[10] see e.g., the completeness proof on page 32

$$\vdash^I_{\text{Hoare}} \{p[x \mapsto t]\}\ x := t\ \{p\}$$

$$\frac{\vdash^I_{\text{Hoare}} \{p\}\ S_1\ \{r\} \qquad \vdash^I_{\text{Hoare}} \{r\}\ S_2\ \{q\}}{\vdash^I_{\text{Hoare}} \{p\}\ S_1;\ S_2\ \{q\}}$$

$$\frac{\vdash^I_{\text{Hoare}} \{p \wedge b\}\ S_1\ \{q\} \qquad \vdash^I_{\text{Hoare}} \{p \wedge \neg b\}\ S_2\ \{q\}}{\vdash^I_{\text{Hoare}} \{p\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{q\}}$$

$$\frac{\forall t : W \cdot \vdash^I_{\text{Hoare}} \{p \wedge b \wedge u = t\}\ S\ \{p \wedge u < t\}}{\vdash^I_{\text{Hoare}} \{p\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{p \wedge \neg b\}}$$

where $(I(W), I(<))$ *is well-founded.*

$$\frac{\vdash^I_{\text{Hoare}} \{p_1\}\ S\ \{q_1\}}{\vdash^I_{\text{Hoare}} \{p\}\ S\ \{q\}}$$

provided for all states σ*, both* $[\![p \Rightarrow p_1]\!] I(\sigma)$ *and* $[\![q_1 \Rightarrow q]\!] I(\sigma)$ *hold.*

Definition 2.37 (Relative Completeness) *Let I be an* arbitrary expressive *interpretation for the programming language* `prog` *and the assertion language L. Then, for any assertion* $p, q : L$ *and program* S : `prog`*, the verification calculus is deemed to be relative complete, if one can show that* $\models^I_{\text{Hoare}} \{p\}\ S\ \{q\}$ *implies* $\vdash^I_{\text{Hoare}} \{p\}\ S\ \{q\}$.

Our machine-checked proof from page 32 is less general. We have merely shown that Hoare Logic is complete when one considers the standard interpretation for LEGO's internal logic. In particular, we have assumed that the interpretation supports our particular choice of the well-founded domain $(\Sigma, <)$. To establish relative completeness one can make no further assumption about the interpretation besides being expressive in the sense of Def. 2.34.

It is well-known that in many circumstances, this requirement is neither sufficient nor necessary i.e. there may be other interpretations for which one may establish a complete verification calculus or there may be calculi which cannot be shown to be complete for certain interpretations considered expressive according to Def. 2.34:

- Clarke Jr. (1979) has shown that one cannot establish relative completeness for imperative programming languages which support procedures as parameters, recursion, static scoping, global variables and internal procedures. The completeness proof breaks down, because one is forced to consider interpretations with *finite* domains.

- Clarke, German & Halpern (1983) show that expressiveness is not sufficient when one considers recursive procedures. For the completeness proof to go through they restrict interpretations to being both expressible and Herbrand-definable[11].

[11] This refined condition is sufficient but not necessary (Grabowski 1985).

- Bergstra & Tucker (1982) show that, for simple imperative programs and Peano Arithmetic as underlying logical language, *all* interpretations give rise to a complete verification calculus

Incompleteness results in Hoare Logic are sometimes misunderstood. They emphasise that previously investigated notions of expressiveness do not capture the precise requirements of the underlying logical language. Furthermore, such incompleteness results should not be considered as criticism of Hoare Logic as a methodology. Regardless of what language features one is interested in, one ought to always be able to cook up a complete verification calculus in the style of Hoare logic with a *particular* assertion language and a *particular* (standard) interpretation. In particular, merely considering standard interpretations, none of the above (relative) incompleteness results go through. Aczel (1982*b*) writes

> " *In the literature the question of the completeness of the proof rules has been entwined with issues concerning the expressiveness of the formal languages to be used in formulating the pre- and post-conditions of the program specifications. In my view the expressiveness of such formal languages should be of no concern when evaluating the completeness of proof rules for program verification.* "

In setting up the notion of validity and derivability of a correctness formula in a machine-checked development, our choice of a shallow embedding of assertions reflects Aczel's view of completeness. We do not consider syntax of an assertion language. Instead assertions are directly captured by predicates on the state space, which, for a machine-checked development, are typically formulated in the native logic of the logical framework. This is a fundamentally different approach to Hoare Logic. Traditionally, assertions are considered to be simply formulae of first-order logic, which are interpreted in the usual way, except that the value of variables is determined by a state.

Following Aczel's approach, working out a suitable syntax for assertions can be regarded as an orthogonal problem. In particular, our new approach to auxiliary variables has led to interpreting assertions as relations on the state space *and* a domain of auxiliary variables. This ought to be reflected by also distinguishing program variables from auxiliary variables at the syntactic level.

Furthermore, when new programming features lead to a new notion of the state space e.g., local variables, the syntax of assertions will have to be adapted to access all information in the state space. We refer to Section 3.3.4 on page 90 for further details. As this thesis establishes, using a shallow embedding of assertions to investigate completeness works smoothly for imperative programs in the presence of recursive procedures and blocks. Modern computer-aided proof systems support such shallow

embeddings because of their powerful internal logics. We see no reason why it should not also work when adding further language features.

2.13 Related Work

For a survey of Hoare Logic, we recommend (Cousot 1990). Proving soundness and completeness for Hoare Logic in the context of simple imperative programs is standard. We commence by relating our approach of employing binary relations in assertions with previous work. We then compare our representation with previous machine-checked developments. We discuss related work concerning recursive procedures and local program variables throughout Cha. 3.

2.13.1 A Rigorous Treatment of Auxiliary Variables

The crucial rôle of auxiliary variables is well known in software engineering practice (Clint 1981, Vickers 1991), but previous presentations of Hoare Logic have swept this aspect under the carpet. The idea of representing assertions as binary relations to provide a *semantic* counterpart for auxiliary variables goes back to Apt & Meertens (1980). They show that with the new representation, in the general case of arbitrary context-free schemes a program is (partially) correct if and only if a suitable *finite* system of intermediate assertions can be found.

As far as we are aware, the only other variant of Hoare Logic in which both assertions are considered as binary relations to cater for auxiliary variables is due to Tarlecki (1985). In his system, the consequence rule has been built into every axiom and the rule for loops. Essentially, it only differs from Hoare Logic in that invariants *inv must not* mention auxiliary variables. The assumption of the rule is in VDM form i.e., the auxiliary variables in the precondition freeze the initial value of all program variables. The conclusion caters for arbitrary assertions p and q, and thus the rule of consequence has to be built in through further side-conditions. Another deviation from VDM's rule of loops is that Tarlecki's (1985) variant *sofar* is not required to be well-founded. Termination is guaranteed by a separate relation $<$ relative to a termination measure $t : \Sigma \to W$.

$$\frac{\begin{array}{c}\{\lambda(Z,\sigma)\cdot Z=\sigma\wedge inv(\sigma)\wedge \mathrm{eval}(\sigma)(b)=\mathbf{true}\}\\ S\\ \{\lambda(Z,\tau)\cdot sofar(Z,\tau)\wedge inv(\tau)\wedge t(\tau)<t(Z)\}\end{array}}{\{p\}\ \mathbf{while}\ b\ \mathbf{do}\ S\ \{q\}}$$

provided

$$(W, <) \text{ is well-founded}$$

$$\forall Z, \sigma \cdot p(Z, \sigma) \Rightarrow inv(\sigma) \qquad \text{(Consequence)}$$

$$\forall \sigma \cdot p(\sigma) \Rightarrow sofar(\sigma, \sigma) \qquad \text{(Reflexivity)}$$

$$\forall Z, \sigma \cdot \exists \eta \cdot sofar(Z, \eta) \wedge inv(\eta) \wedge sofar(\eta, \sigma) \wedge inv(\sigma) \Rightarrow sofar(Z, \sigma) \qquad \text{(Transitivity)}$$

$$\forall Z, \tau \cdot \exists \sigma \cdot \Big(p(Z, \sigma) \wedge sofar(\sigma, \tau) \wedge inv(\tau) \wedge \text{eval}(\tau)(b) = \textbf{false} \wedge t(\tau) \leq t(\sigma)\Big) \Rightarrow q(\tau, Z) \qquad \text{(Consequence)}$$

The difficulty in program verification is largely due to discovering appropriate invariants (Gries 1981, Gries 1982, Ireland & Stark 1997). Not being able to mention auxiliary variables in the invariant appears to be a troublesome requirement.

2.13.2 Machine-Checked Developments

We are not aware of any previous machine-checked proofs of soundness and completeness for VDM. Unless otherwise indicated, the following formalisations deal with Hoare Logic in the setting of *partial* correctness. We first review work done in the Edinburgh Logical Framework where Hoare Logic is axiomatised and neither soundness nor completeness have been formally pursued. We then discuss formalisations in the system HOL in which soundness is established by deriving the axioms and rules of Hoare Logic from the underlying operational semantics. Finally, we present Nipkow's (1998) machine-checked development in Isabelle which caters for both soundness and completeness by separating between the semantics and a notion of derivability in Hoare Logic.

2.13.2.1 An Axiomatic Approach to Hoare Logic

Employing the Edinburgh Logical Framework, in Mason's (1987) work, terms are either boolean expressions inhabiting the type b, or expressions of type i. All program variables denote values of type i. In particular, one can inspect their value with the help of a global lookup function $! : \textbf{VAR} \to i$. This simplifies the treatment of assignments which can thus be represented without dependent types i.e.,

$$:= : (\textbf{VAR} \times i) \to \texttt{prog} \ .$$

Assertions are objects of type o which are constructed syntactically from expressions e.g.,

$$=_o : (i \times i) \to o$$
$$\Rightarrow_o : (o \times o) \to o$$

To exploit the provisions for λ-binding in the framework, instead of a purely syntactic account of quantification e.g.,

$$\exists_o : (\mathbf{VAR} \times o) \to o$$

Mason chose

$$\exists_o : (i \to o) \to o \quad .$$

Rather than providing an interpretation for terms and assertions, Mason suggests axiomatising a predicate $\vdash_o$ to capture derivability (Harper, Honsell & Plotkin 1993).

Derivability in Hoare Logic is then also axiomatised by a set of rules where correctness formulae are triples of the form $o \times \mathtt{prog} \times o$. The only difficulty is caused by the assignment axiom

$$\{p[x \mapsto t]\}\ x := t\ \{p\} \quad , \tag{2.54}$$

because one would have to formalise the notion of updating the value of variables in an assertion. This is the major problem in choosing a deep embedding for assertions. Unlike the shallow embedding scenario, moving to function application to relay the work to the internals of the logical framework does not succeed. In particular, the axiom

$$\vdash_{\text{Hoare}} \{p(t)\}\ x := t\ \{p(x!)\} \tag{2.94}$$

for $p : i \to o$ is unsound. Intuitively, the abstraction in p may not necessarily bind all occurrences of references to the program variable x. More precisely, for

$$p = \lambda u \cdot x! \neq_{\mathtt{int}} u \quad , \tag{2.95}$$

we would be able to erroneously derive

$$\vdash_{\text{Hoare}} \{x! \neq_{\mathtt{int}} 1\}\ x := 1\ \{x! \neq_{\mathtt{int}} x!\} \quad .$$

2.13.2.1.1 Non-Interference Therefore, Mason suggests that one augments the assignment axiom (2.94) with a predicate $\# \subseteq \mathbf{VAR} \times o$ on x and $\forall p$ which guarantees that x does not interfere with p. The author characterises non-interference with six further rule schema e.g.,

$$x \# y! \qquad \text{provided } x \neq y$$
$$\frac{x \# t_1 \quad x \# t_2}{x \# (t_1 =_i t_2)}$$

Similar ideas can be found in specification logic.

2.13.2.1.2 Explicit Quantification Another solution to ensure that $x \# \forall p$ has been proposed by Caplan (1995). He extends the notion of a correctness formula CF to include quantification of program variables

$$\{.\}\ .\ \{.\} : (o \times \mathtt{prog} \times o) \to CF$$
$$\forall_{CF} : (\mathbf{VAR} \to CF) \to CF$$

His axiom

$$\vdash_{\text{Hoare}} \forall_{CF} \left(\lambda x \cdot \{p(t(x))\}\ x := t(x)\ \{p(x!)\}\right) \tag{2.96}$$

for $p : i \to o$ and $t : \mathbf{VAR} \to i$ ensures that the active program variable of an assignment is bound. His system also includes a rule for permuting the order of quantification and a proof transformation rule to add quantifiers. If one were to apply the rule of assignment to an assertion in which an active program variable occurs freely such as (2.95), α-conversion of the λ-calculus guarantees that the active identifier in (2.96) is suitably renamed. In particular, one may derive

$$\vdash_{\text{Hoare}} \forall_{CF}(\lambda y \cdot \{p(1)\}\ y := 1\ \{p(y!)\})\ ,$$

whereas

$$\vdash_{\text{Hoare}} \forall_{CF}(\lambda x \cdot \{p(1)\}\ x := 1\ \{p(x!)\})$$

is *not* an instance of the axiom (2.96). Furthermore, it is not derivable by any other means due to the exclusion of any elimination rules for quantifiers at the level of correctness formulae.

Soundness is not formally investigated. Instead, Mason (1987) and Caplan (1995) verify on paper that their encoding adequately reflects Hoare's (1969) axiomatic presentation.

2.13.2.2 Hoare Logic as a Set of Derived Rules

Machine-checked soundness proofs for verification calculi have been pioneered by Gordon (1989). Instead of axiomatising Hoare Logic, one axiomatises a more basic semantical account[12] for the imperative programming language. Assertions are represented by predicates on the state space. Thus, updating assertions can be expressed in terms of updating states.

Similar to Def. 2.20 on page 30, the *semantics* of Hoare Logic[13] $\models_{\text{Hoare}} \{.\} \cdot \{.\}$ can then be described in terms of the underlying semantics. Deriving the Hoare Logic rules of Figure 2.1 on page 30 e.g., showing

$$\frac{\begin{array}{l}\models_{\text{Hoare}} \{\lambda\sigma \cdot p(\sigma) \wedge \text{eval}(\sigma)(b) = \textbf{true}\}\ S_1\ \{q\} \\ \models_{\text{Hoare}} \{\lambda\sigma \cdot p(\sigma) \wedge \text{eval}(\sigma)(b) = \textbf{false}\}\ S_2\ \{q\}\end{array}}{\models_{\text{Hoare}} \{p\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{q\}}$$

then corresponds to a proof of soundness.

2.13.2.2.1 Side Effects In (Homeier 1995) and[14] (Homeier & Martin 1996), expressions are also restricted to two kinds, booleans and natural numbers. Moreover, program variables may only store natural numbers.

As a novelty, expressions in programming language may have side effects. For a program variable x, the expression $++x$, when evaluated, first increments the value of x and then yields its new value. To cater for expressions with side-effects, it is advisable to provide a *deep* embedding of expressions e.g., the meaning of $++x$ is given operationally by

$$\frac{\sigma \xrightarrow{x \Downarrow n} \tau}{\sigma \xrightarrow{++x \Downarrow n+1} \tau[x \mapsto n+1]}\ .$$

With side-effects, the order of evaluation matters e.g.,

$$\frac{\sigma \xrightarrow{e_1 \Downarrow n_1} \eta \quad \eta \xrightarrow{e_2 \Downarrow n_2} \tau}{\sigma \xrightarrow{e_1 + e_2 \Downarrow n_1 + n_2} \tau}\ . \tag{2.97}$$

Additionally, the authors give a denotational account for the side-effect free fragment of the class of expressions e.g., one has also $[\![e_1 + e_2]\!]I(\sigma) \stackrel{\text{def}}{=} [\![e_1]\!]I(\sigma) + [\![e_2]\!]I(\sigma)$.

[12] Gordon chooses relational denotational semantics.

[13] Gordon (1989) also shows how to extend this technique to total correctness, VDM, Dijkstra's weakest preconditions and dynamic logic.

[14] Beware that the presentation of Hoare Logic in (Homeier & Martin 1996) is (trivially) incomplete because the authors have omitted a consequence rule. Thus, one can e.g. not derive $\{x = 1\}$ **skip** $\{\textbf{true}\}$. Their main contribution is to derive a Verification Condition Generator which is proven sound with respect to an *operational* semantics.

The expression $++x$ must however not occur in assertions and, thus, one has a choice between deep and shallow embedding of assertions. Homeier & Martin chose a deep embedding for assertions. Instead of axiomatising derivability or shallowly embedding assertions as predicates on the state space, assertions are evaluated relative to the standard[15] interpretation e.g.,

$$[\![t_1 =_{\mathtt{nat}} t_2]\!]I(\sigma) \stackrel{\text{def}}{=} [\![t_1]\!]I(\sigma) = [\![t_2]\!]I(\sigma)$$
$$[\![\exists_o x \cdot p]\!]I(\sigma) \stackrel{\text{def}}{=} \exists n \cdot [\![p]\!]I(\sigma[x \mapsto n])$$

To deal with the assignment axiom $\{p[x \mapsto t]\}\ x := t\ \{p\}$, one needs to implement a substitution function on assertions which has to translate subexpressions of t of the form $++x$ to $x+n$ for some suitable n e.g., $(x =_{\mathtt{nat}} X)[x \mapsto (++x)+(++x)]$ yields $x+4 =_{\mathtt{nat}} X$. One ought to also verify the correctness of substitution by showing the substitution lemma

$$[\![p[x \mapsto t]]\!]I(\sigma) = [\![p]\!]I(\tau[x \mapsto [\![n]\!]I(\tau)]) \qquad \text{where } \sigma \xrightarrow{t \Downarrow n} \tau,$$

but this does not seem to have been formally established by the authors.

Norrish (1996,1997) is working on formalising the operational semantics of the programming language C. In particular, he introduces non-determinism for evaluating expressions, because, in accordance with the C standard (Schildt et al. 1990), side effects do not have to be applied as they are generated. Furthermore, expressions involving binary operators are not necessarily evaluated by evaluating one argument and then the next. Thus, the rule (2.97) is not sound for C.

Based on the detailed operational semantics, Norrish sketches Hoare Logic rules for a well-behaved class of C programs e.g.,

$$\frac{\models_{\text{Hoare}} \{p\}\ S_1\ \{r\} \qquad \models_{\text{Hoare}} \{r\}\ S_2\ \{q\}}{\models_{\text{Hoare}} \{p\}\ S_1;\ S_2\ \{q\}}$$

provided S_1 contains no interrupt statements.

2.13.2.3 Semantics and Derivability of Correctness Formulae in Hoare Logic

Nipkow (1998) has been the first to conduct a machine-checked *completeness* proof for Hoare Logic dealing with simple imperative programs. To formally deal with completeness, one needs to distinguish between semantics $\models_{\text{Hoare}} \{.\}\ .\ \{.\}$ and derivability $\vdash_{\text{Hoare}} \{.\}\ .\ \{.\}$ of correctness formulae. For simple imperative programs, instead of appealing to the MGF, completeness is established as a corollary of

$$\vdash_{\text{Hoare}} \{\text{wp}(S,q)\}\ S\ \{q\}$$

[15] All logical connectives are directly embedded in the native logic of the framework HOL.

for arbitrary programs S and assertions[16] q. Following (Gordon 1989), assertions are encoded as predicates on the state space. Thus, one does not have to worry about how to overcome expressivity conditions such as having to syntactically formulate an assertions which corresponds to $\mathrm{wp}(S,q)$.

The development follows the first 100 pages of Winskel's (1993) textbook on semantics. It includes various operational and denotational accounts together with equivalence proofs. Furthermore, Nipkow produces machine-checked soundness and completeness proofs for Hoare Logic and a Verification Condition Generator. This work uncovered a mistake in Winskel's (1993) completeness proof for Hoare Logic.

[16] Induction on S does not go through if one investigates recursive procedures. It then becomes essential to employ the MGF property.

Chapter 3

Recursive Procedures and Local Program Variables

In this chapter, we extend our soundness and completeness results to more interesting program language features. We consider recursive procedures and local program variables with static binding. We produce machine-checked soundness and completeness proofs for parameterless recursive procedures and local program variables. We sketch extensions to mutually recursive procedures and call-by-value parameters.

To adequately represent recursive procedures, we present the verification calculi as a Gentzen-style formal system with contexts. As a new contribution, we improve Sokołowski's (1977) rule for recursive procedures by allowing a well-founded termination argument. In our rules for local program variables, we exploit the fact that, due to our choice of a shallow embedding, assertions may refer to the values of all program variables, and not just those in scope.

In view of the history of unsound and incomplete verification calculi for imperative programs dealing with recursive procedures, the surprising result of this chapter is that, due to our new approach to auxiliary variables, none of the soundness and completeness results are particularly difficult to achieve. Moreover, for Hoare Logic in the setting of total correctness, our results are an improvement over previous work. We require only two thirds of the total number of America & de Boer's (1990) nine rules.

The outline of this chapter is as follows. We first consider recursive procedures. We then focus on a verification calculus for simple imperative programs and local program variables. In Sect. 3.3, we combine recursive procedures with local program variables. Finally, in Sect. 3.4, we sketch extensions to call-by-value parameter passing. We compare our approach to previous work throughout this chapter.

3.1 Parameterless Recursive Procedures

In this section, we extend both Hoare Logic and VDM with parameterless recursive procedures and establish that the new verification calculi are sound and complete. In the previous chapter, we had urged that auxiliary variables should be taken seriously for a more adequate formalisation of Hoare Logic. This section substantiates this claim.

Apt (1981) and America & de Boer (1990) present a verification calculus in the style of Hoare Logic in which, besides a decomposition rule for procedure invocations, they need to add three further structural rules in addition to Hoare's (1969) consequence rule to guarantee completeness. We are able to show that those four structural rules are subsumed by our new consequence rule.

As a further contribution, we show that Hoare's (1971) rule of adaptation, a specially tailored rule structural rule essential in the presence of recursion, is just a trivial instance of our new consequence rule, which we had motivated in the previous chapter for simple imperative programs.

Again, proving these meta-theoretical results for VDM is analogous and we merely highlight technical differences for VDM.

3.1.1 Declaring and Invoking Procedures

Restricting our attention to a single procedure declaration, invoking this procedure can be achieved unambiguously with a constructor **call** : `prog`.

➥ p. 156 **Definition 3.1 (Syntax)** *The class of imperative programs* S : `prog` *which may appeal to a recursive procedure is defined by the BNF grammar*

$$S ::= \textbf{skip} \mid x := t \mid S_1;\ S_2 \mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \mid \textbf{while } b \textbf{ do } S \mid \textbf{call}$$

where x : **VAR**, t : expression$\big(\mathrm{sort}(x)\big)$ *and* b : expression(`bool`).

We assume that the procedure declaration is given statically e.g., at compile-time. In the sequel, let S_0 : `prog` denote the body of the procedure.

➥ p. 157 **Example 3.2 (Factorial)** *The body of a recursive procedure computing the factorial of* x *is given by*

$$\begin{aligned} S_0 \stackrel{\text{def}}{=}\ & \textbf{if } x = 0 \ \ \textbf{then } y := 1 \\ & \textbf{else begin } x := x - 1;\ \textbf{call};\ x := x + 1;\ y := y * x \ \textbf{end} \end{aligned}$$

Notice that the argument x *and the result* y *are not parameters, but global variables. Clearly,* x *plays the rôle of a call-by-value parameter. The procedure ensures that its value remains invariant. Similarly, the variable* y *is a designated call-by-name parameter.*

3.1.2 Structural Operational Semantics

Whenever a procedure invocation is encountered, computation proceeds by executing the body of the procedure S_0.

➥ p. 157 **Definition 3.3 (Semantics)** *We extend the operational semantics from Def. 2.8 by*

$$\frac{\sigma \xrightarrow{S_0} \tau}{\sigma \xrightarrow{\textbf{call}} \tau} \quad . \tag{3.1}$$

Again, notice that a denotational characterisation would require more sophisticated mathematical notions such as least fixpoints.

3.1.3 Hoare Logic with Contexts

Without recursion, deriving a correctness formula is easy. Whenever one encounters a procedure invocation, one has to proceed by instead analysing the procedure body (Apt 1981) i.e.,

$$\frac{\vdash_{\text{Hoare}} \{p\}\ S_0\ \{q\}}{\vdash_{\text{Hoare}} \{p\}\ \textbf{call}\ \{q\}} \quad .$$

This rule would however lead to infinite derivations when S_0 calls itself. Induction comes to the rescue. Let us first omit the issue of termination. We may simply assume $\vdash_{\text{Hoare}} \{p\}\ \textbf{call}\ \{q\}$ to conclude $\vdash_{\text{Hoare}} \{p\}\ S_0\ \{q\}$ (Hoare 1971) i.e.,

$$\frac{\{p\}\ \textbf{call}\ \{q\} \vdash_{\text{Hoare}} \{p\}\ S_0\ \{q\}}{\vdash_{\text{Hoare}} \{p\}\ \textbf{call}\ \{q\}} \quad . \tag{3.2}$$

This rule introduces a fundamental change in deriving correctness formulae. Derivations are now to be considered with respect to a context. Instead of a Hilbert-style calculus, Hoare Logic now amounts to a Gentzen-style sequent calculus. Actually, the rule above is to be understood with respect to an arbitrary context

$$\frac{\{p\}\ \textbf{call}\ \{q\}, \Gamma \vdash_{\text{Hoare}} \{p\}\ S_0\ \{q\}}{\Gamma \vdash_{\text{Hoare}} \{p\}\ \textbf{call}\ \{q\}} \quad . \tag{3.3}$$

The previous presentation needs to be revised to support contexts e.g.,

$$\frac{\Gamma \vdash_{\text{Hoare}} \{p\}\ S_1\ \{r\} \quad \Gamma \vdash_{\text{Hoare}} \{r\}\ S_2\ \{q\}}{\Gamma \vdash_{\text{Hoare}} \{p\}\ S_1; S_2\ \{q\}}$$

$$\Gamma \vdash_{\text{Hoare}} \{p\}\ \textbf{skip}\ \{p\}$$

Furthermore, such sequent calculi are usually equipped with a collection of structural rules, see e.g. (Takeuti 1975):

$$\frac{\Gamma \vdash P}{D, \Gamma \vdash P} \quad \text{(Weakening)} \qquad \frac{\Gamma \vdash P \quad P, \Delta \vdash Q}{\Gamma, \Delta \vdash Q} \quad \text{(Cut)}$$

$$\frac{D, D, \Gamma \vdash P}{D, \Gamma \vdash P} \quad \text{(Contraction)} \qquad P \vdash P \quad \text{(Initial Sequent)}$$

$$\frac{\Gamma, C, D, \Delta \vdash P}{\Gamma, D, C, \Delta \vdash P} \quad \text{(Exchange)}$$

As usual, we abbreviate $\emptyset \vdash P$ by $\vdash P$.

Olderog (1981) has observed that this presentation is redundant for Hoare Logic in the sense that if $\Gamma \vdash P$ is derivable than there is also a *standard* proof which only involves a *single* application of the procedure rule (3.3). Since we are only interested in deriving correctness formulae in the empty context, it suffices to consider the original version (3.2) after all. Thus, the antecedent contains at most one correctness formula and, apart from the Initial Sequent, none of the structural rules are actually required.

Redundancy in a formal system may be a healthy feature if the additional flexibility gives rise to easier derivations. This is however not the case here when restricting antecedents to at most one correctness formula. Hoare Logic (and VDM) is a syntax-directed methodology (Harel 1980). A rule, read backwards in refinement style, reduces the complexity by focussing the attention on strict subcomponents of the program to be verified. Intuitively, when one has to verify the correctness of $\{p\}$ **call** $\{q\}$ one may reduce this task to showing $\{p\}\ S_0\ \{q\}$, where we may discharge *any* further intermediate obligations of the form $\{p'\}$ **call** $\{q'\}$ with the assistance of the consequence rule. Just like in the case of a loop, it is crucial to find an appropriately weak precondition p so that all appeals to $\{p\}$ **call** $\{q\}$ can be resolved.

Furthermore, unlike traditional sequent calculi, the antecedent may only mention correctness formulae dealing with procedure invocation. In particular, due to our restriction to a single procedure, it suffices to consider contexts with at most one correctness formulae about procedure invocations.

➥ p. 158 **Definition 3.4 (Contexts)** *A context* Γ : `Context` *can be represented by the BNF grammar* $\Gamma ::= \emptyset \mid \{p\}$ **call** $\{q\}$.

Since we are working in a logical framework which supports contexts at the meta level, the question arises as to whether one can avoid explicitly introducing contexts at the object level. Having already introduced a Hilbert-style notion of derivability, one may be tempted to represent the sequent $P \vdash Q$ by $\vdash P \Rightarrow \vdash Q$. However, in general, $P \vdash Q$ is a *stronger* property.

Consider a formal system which is not complete i.e., there exists a formula P such that one can neither derive P, nor its complement i.e., $\nvdash P$ and $P \nvdash \mathbf{false}$. As $\vdash P$ does not hold, we may infer any property from it, in particular, $\vdash P \Rightarrow \vdash \mathbf{false}$; yet, by assumption, $P \vdash \mathbf{false}$ does not hold. Moreover, in the setting of Hoare Logic, de Bakker (1980) shows that $\vdash P \Rightarrow \vdash Q$ does not imply $P \vdash Q$.

An orthogonal complication arises because we define derivability of correctness formulae by considering the smallest fixpoint of a set of predicates on a relation $\vdash_{\text{Hoare}} \{.\}\ .\ \{.\}$. A rule of the form

$$(\vdash P \Rightarrow \vdash Q) \Rightarrow \vdash R \tag{3.4}$$

is not an *adequate* representation of

$$(P \vdash Q) \Rightarrow \vdash R \quad . \tag{3.5}$$

Assume that we have to derive $\vdash R$. By (3.4), it suffices to show that from $\vdash P$, we can infer $\vdash Q$. We may now refine $\vdash Q$ and discharge any intermediate proof obligation $\vdash P$. However, due to having closed off the set of rules by the smallest fixpoint, we may additionally assume that $\vdash P$ has been derived by one of the constructors. In particular, we may analyse the *derivation* of $\vdash P$. This is clearly not the intention of (3.5).

Constructors of the form (3.4) are not permitted in inductive definitions due to the negative occurrence of $\vdash$ in the premiss. Such a syntactic check guarantees that there is a smallest fixpoint. Due to the structure of the *other* rules of Hoare Logic, it is however legitimate to add the rule

$$\frac{\vdash_{\text{Hoare}} \{p\}\ \mathbf{call}\ \{q\} \Rightarrow \vdash_{\text{Hoare}} \{p\}\ S_0\ \{q\}}{\vdash_{\text{Hoare}} \{p\}\ \mathbf{call}\ \{q\}} \tag{3.6}$$

and *derive* that the set of rules has a smallest fixpoint. One can show that it leads to a consistent definition in that only valid correctness formulae may be derived. Moreover, this system is elementary equivalent to the sequent style presentation i.e., it admits precisely the same correctness formulae. However, adding further (sound) rules may jeopardise the soundness of the whole system (Trakhtenbrot et al. 1984).

We have decided against working with a set of rules involving (3.6), because

- our logical framework does not offer any help in closing a set of rules unless it is an inductive definition.
- In our opinion, this presentation is not adequate in that it does not reflect the intuition of the informal rule (3.2).
- The calculus is not sound in the usual sense. In particular, adding further (sound) rules may yield an unsound formal system.

We thus prefer to formalise Hoare Logic with contexts as a Gentzen-style system.

3.1.4 Total Correctness

To guarantee termination, one needs to introduce a termination measure on states. In the case of loops, we compared states before and after an execution of the loop body. For recursive procedures, we can guarantee progress if recursive procedure invocations are only permitted in strictly smaller states with respect to the termination measure. Sokołowski (1977) suggests

$$\frac{\forall n : \mathtt{nat} \cdot \{p(n)\}\ \mathbf{call}\ \{q\} \vdash_{\text{Hoare}} \{p(n+1)\}\ S_0\ \{q\}}{\vdash_{\text{Hoare}} \{\exists n : \mathtt{nat} \cdot p(n)\}\ \mathbf{call}\ \{q\}} \qquad \text{provided } \neg p(0). \tag{3.7}$$

This rule is both sound and sufficient to obtain a complete set of rules for recursive procedures (America & de Boer 1990, Schreiber 1997). However, it is too cumbersome to apply in most examples, because of the requirement that the termination measure decreases by exactly one. In particular, in the Quicksort algorithm, which we verify in Chap. 4, the sorting procedure is invoked recursively where the array size is strictly smaller, but not necessarily by one element. We have therefore extended Sokołowski's rule to an arbitrary well-founded measure

$$\frac{\forall t : W \cdot \{\exists u : W \cdot p(u) \wedge u < t\}\ \mathbf{call}\ \{q\} \vdash_{\text{Hoare}} \{p(t)\}\ S_0\ \{q\}}{\vdash_{\text{Hoare}} \{\exists t : W \cdot p(t)\}\ \mathbf{call}\ \{q\}} \qquad \text{where } (W, <) \text{ is well-founded.} \tag{3.8}$$

As Apt (1981) points out, merely adding such a rule for procedure invocations in a setting with Hoare's (1969) consequence rule does not lead to a complete system. We will show that a cure is to employ the new rule of consequence (2.70) which is more flexible in dealing with auxiliary variables[1]. We have collected the set of rules dealing with recursive programs which we have formalised in Figure 3.1 on the next page. Other solutions have been more elaborate. More precisely, Apt (1981) proposes to add three more rules to be able to modify auxiliary variables in correctness formulae dealing with recursive procedures, see Figure 3.2 on page 64.

However, America & de Boer (1990) have shown that this extension yields an *unsound* system. Their remedy consists of distinguishing different kinds of (auxiliary) variables. In particular, the universally quantified counter variable of the procedure invocation rule must not occur in the list of variables $\overline{Y}$ and $\overline{Z}$ in the Substitution and Elimination rules of Figure 3.2 on page 64.

Having introduced Hoare Logic as a sequent calculus, we extend the semantics to incorporate contexts by

[1] We have demonstrated in (Schreiber 1997) that soundness and completeness can also be established when replacing the well-founded measure by the original version from (3.7).

$$\{p\}\ \mathbf{call}\ \{q\} \vdash_{\text{Hoare}} \{p\}\ \mathbf{call}\ \{q\} \tag{3.9}$$

$$\Gamma \vdash_{\text{Hoare}} \{p\}\ \mathbf{skip}\ \{p\}$$

$$\Gamma \vdash_{\text{Hoare}} \left\{\lambda(Z,\sigma)\cdot p\big(Z,\sigma[x \mapsto \text{eval}(\sigma)(t)]\big)\right\}\ x{:=}t\ \{p\}$$

$$\frac{\Gamma \vdash_{\text{Hoare}} \{p\}\ S_1\ \{r\} \quad \Gamma \vdash_{\text{Hoare}} \{r\}\ S_2\ \{q\}}{\Gamma \vdash_{\text{Hoare}} \{p\}\ S_1;\ S_2\ \{q\}}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{Hoare}} \{\lambda(Z,\sigma)\cdot p(Z,\sigma) \wedge \text{eval}(\sigma)(b) = \mathbf{true}\}\ S_1\ \{q\} \\ \Gamma \vdash_{\text{Hoare}} \{\lambda(Z,\sigma)\cdot p(Z,\sigma) \wedge \text{eval}(\sigma)(b) = \mathbf{false}\}\ S_2\ \{q\}\end{array}}{\Gamma \vdash_{\text{Hoare}} \{p\}\ \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \{q\}}$$

$$\frac{\forall t : W \cdot \Gamma \vdash_{\text{Hoare}} \{\hat{p}\}\ S\ \{\lambda(Z,\tau)\cdot p(Z,\tau) \wedge u(\tau) < t\}}{\Gamma \vdash_{\text{Hoare}} \{p\}\ \mathbf{while}\ b\ \mathbf{do}\ S\ \{\lambda(Z,\tau)\cdot p(Z,\tau) \wedge \text{eval}(\tau)(b) = \mathbf{false}\}}$$

where $\hat{p}(Z,\sigma) \stackrel{\text{def}}{=} p(Z,\sigma) \wedge \text{eval}(\sigma)(b) = \mathbf{true} \wedge u(\sigma) = t$
and $(W,<)$ is well-founded.

$$\frac{\forall t : W \cdot \{\lambda(Z,\sigma)\cdot \exists u : W \cdot p(u)(Z,\sigma) \wedge u < t\}\ \mathbf{call}\ \{q\} \vdash_{\text{Hoare}} \{p(t)\}\ S_0\ \{q\}}{\vdash_{\text{Hoare}} \{\lambda(Z,\sigma)\cdot \exists t : W \cdot p(t)(Z,\sigma)\}\ \mathbf{call}\ \{q\}}$$

where $(W,<)$ is well-founded. (3.10)

$$\frac{\Gamma \vdash_{\text{Hoare}} \{p_1\}\ S\ \{q_1\}}{\Gamma \vdash_{\text{Hoare}} \{p\}\ S\ \{q\}}$$

where $p_1, q_1 : \text{Assertion}(U)$ and $p, q : \text{Assertion}(T)$ for arbitrary types T and U
and $\forall Z \cdot \forall \sigma \cdot p(Z,\sigma) \Rightarrow \Big(\exists Z_1 \cdot p_1(Z_1,\sigma) \wedge \big(\forall \tau \cdot q_1(Z_1,\tau) \Rightarrow q(Z,\tau)\big)\Big)$.

Figure 3.1: Hoare Logic for Recursive Procedures – Total Correctness

$$\frac{\{p\}\ \mathbf{call}\ \{q\}}{\{p\,[\overline{Z}\mapsto\overline{Y}]\}\ \mathbf{call}\ \{q\,[\overline{Z}\mapsto\overline{Y}]\}}$$
where $\overline{Z}\cap free(S_0)=\emptyset$ and $\overline{Y}\cap free(S_0)=\emptyset$. (Substitution)

$$\frac{\{p\}\ \mathbf{call}\ \{q\}}{\{p\wedge r\}\ \mathbf{call}\ \{q\wedge r\}}$$ where $free(r)\cap free(S_0)=\emptyset$. (Invariance)

$$\frac{\{p\}\ \mathbf{call}\ \{q\}}{\{\exists\overline{Z}\cdot p\}\ \mathbf{call}\ \{q\}}$$ where $\overline{Z}\cap free(S_0,q)=\emptyset$. (Elimination)

Figure 3.2: Hoare Logic – Additional Structural Rules (Apt 1981)

➥ p. 160

Definition 3.5 (Semantics of Hoare Logic)
We define $\Gamma\models_{\text{Hoare}}\{p\}\ S\ \{q\}$ *by case analysis on the structure of the context* Γ*:*

$$\Gamma\models_{\text{Hoare}}\{p\}\ S\ \{q\}\stackrel{\text{def}}{=}\begin{cases}\models_{\text{Hoare}}\{p\}\ S\ \{q\} & \text{for } \Gamma=\emptyset,\\ (\models_{\text{Hoare}}\{p'\}\ \mathbf{call}\ \{q'\})\Rightarrow(\models_{\text{Hoare}}\{p\}\ S\ \{q\}) & \text{for } \Gamma=\{p'\}\ \mathbf{call}\ \{q'\}.\end{cases}$$

Notice that, semantically, existential quantification in the precondition corresponds to universal quantification at the level of a correctness formula:

➥ p. 160

Lemma 3.6 (Existential Quantification in Preconditions)
For all programs S*, indexed preconditions* $p:U\to\text{Assertion}(T)$ *and postconditions* $q:\text{Assertion}(T)$ *for arbitrary types* T *and* U*, the propositions*

$$\models_{\text{Hoare}}\{\lambda(Z,\sigma)\cdot\exists t\cdot p(t)(Z,\sigma)\}\ S\ \{q\}\qquad\textit{and}\qquad\forall t\cdot\models_{\text{Hoare}}\{p(t)\}\ S\ \{q\}$$

are equivalent.

Proof Existential quantification on the left side of an implication is equivalent to universal quantification. □

However, for derivability in Hoare Logic, it is preferable to work with the two existential quantifiers in rule (3.8):

- Replacing the existential quantifier in the premiss would lead to a more complicated notion of contexts.
- Introducing universal quantifiers at the level of correctness formulae in the conclusion alters the notion of derivability.

3.1.5 Soundness

➥ p. 162 **Theorem 3.7 (Soundness)**

Let p and q be arbitrary assertions and S be any program. For an arbitrary context Γ, whenever $\Gamma \vdash_{\text{Hoare}} \{p\}\ S\ \{q\}$ is derivable, $\Gamma \models_{\text{Hoare}} \{p\}\ S\ \{q\}$ holds.

Proof by induction on the derivation of

$$\Gamma \vdash_{\text{Hoare}} \{p\}\ S\ \{q\} \tag{3.11}$$

Consider the new case that (3.11) has been derived by applying the rule for procedure invocations (3.10). From the induction hypothesis

$$\forall t : W \cdot \{\lambda(Z,\sigma) \cdot \exists u : W \cdot p(u)(Z,\sigma) \wedge u < t\}\ \mathbf{call}\ \{q\} \models_{\text{Hoare}} \{p(t)\}\ S_0\ \{q\} \tag{3.12}$$

for some well-founded structure $(W, <)$, we need to show

$$\models_{\text{Hoare}} \{\lambda(Z,\sigma) \cdot \exists t : W \cdot p(t)(Z,\sigma)\}\ \mathbf{call}\ \{q\} \quad .$$

Appealing to Lemma 3.6, we may externalise the quantification in the proof obligation, leading to

$$\forall t : W \cdot \models_{\text{Hoare}} \{p(t)\}\ \mathbf{call}\ \{q\} \quad .$$

We pursue well-founded induction. We thus have to establish $\models_{\text{Hoare}} \{p(t)\}\ \mathbf{call}\ \{q\}$ for a fixed $t : W$ and may additionally appeal to the hypothesis

$$\forall u : W \cdot u < t \Rightarrow \models_{\text{Hoare}} \{p(u)\}\ \mathbf{call}\ \{q\}$$

which is equivalent to the required instance of the context in (3.12). □

3.1.6 Completeness

Completeness is again a straight-forward corollary of the MGF property.

➥ p. 165 **Theorem 3.8 (MGF)** $\vdash_{\text{Hoare}} \left\{\lambda(Z,\sigma) \cdot \sigma \xrightarrow{\quad S \quad} Z\right\}\ S\ \{\lambda(Z,\tau) \cdot Z = \tau\}$

In its proof, in the case of a procedure invocation, we have to plant a termination measure in the precondition so that $\exists t : W \cdot \psi(t)(\tau,\sigma)$ is equivalent to $\sigma \xrightarrow{\quad \mathbf{call} \quad} \tau$ for a suitable ψ. We exploit ideas from a completeness proof by America & de Boer (1990) in which $\psi(t)(\tau,\sigma)$ asserts that the execution of $\sigma \xrightarrow{\quad \mathbf{call} \quad} \tau$ can be accomplished with a recursive depth of at most $t : \mathtt{nat}$. In a denotational setting with $[\![.]\!] : \Sigma \rightharpoonup \Sigma$, they define

$$\begin{aligned} \psi(t)(\tau,\sigma) &\stackrel{\text{def}}{=} [\![S^{[t]}]\!](\sigma) = \tau \wedge \tau \neq \bot \\ \text{where} \quad S^{[t]} &\stackrel{\text{def}}{=} S[S_0^{(t)}/\mathbf{call}] \ , \\ S_0^{(0)} &\stackrel{\text{def}}{=} \Omega \\ \text{and} \quad S_0^{(t+1)} &\stackrel{\text{def}}{=} S_0[S_0^{(t)}/\mathbf{call}] \ . \end{aligned}$$

This seems cumbersome. Denotational semantics gives a too abstract view for expressing such fine-grained attributes and one has to simulate a low-level account by selectively substituting the non-terminating program Ω and deduce properties by checking if test runs terminate. In particular, Ω has nothing to do with the notion of recursive depth. We prefer the following definition which is simply an annotated version of the operational semantics where every rule apart from procedure invocation preserves recursive depth.

➥ p. 163

Definition 3.9 (Recursive Depth)
Let $\sigma \xrightarrow{S}_n \tau \subseteq \Sigma \times \mathtt{prog} \times \mathtt{nat} \times \Sigma$ *be a new judgement which is defined as the least relation satisfying*

$$\sigma \xrightarrow{\textbf{skip}}_n \sigma$$

$$\sigma \xrightarrow{x:=t}_n \sigma[x \mapsto \mathrm{eval}(\sigma)(t)]$$

$$\frac{\sigma \xrightarrow{S_1}_n \eta \quad \eta \xrightarrow{S_2}_n \tau}{\sigma \xrightarrow{S_1;\, S_2}_n \tau}$$

$$\frac{\sigma \xrightarrow{S_1}_n \tau}{\sigma \xrightarrow{\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2}_n \tau} \quad \textit{provided } \mathrm{eval}(\sigma)(b) = \textbf{true} \ .$$

$$\frac{\sigma \xrightarrow{S_2}_n \tau}{\sigma \xrightarrow{\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2}_n \tau} \quad \textit{provided } \mathrm{eval}(\sigma)(b) = \textbf{false} \ .$$

$$\sigma \xrightarrow{\textbf{while } b \textbf{ do } S}_n \sigma \quad \textit{provided } \mathrm{eval}(\sigma)(b) = \textbf{false} \ .$$

$$\frac{\sigma \xrightarrow{S}_n \eta \quad \eta \xrightarrow{\textbf{while } b \textbf{ do } S}_n \tau}{\sigma \xrightarrow{\textbf{while } b \textbf{ do } S}_n \tau} \quad \textit{provided } \mathrm{eval}(\sigma)(b) = \textbf{true} \ .$$

$$\frac{\sigma \xrightarrow{S_0}_n \tau}{\sigma \xrightarrow{\textbf{call}}_{n+1} \tau}$$

➥ p. 164

Lemma 3.10 (Non-strict Recursive Depth) *Bounded Recursive Depth is not a strict bound i.e., if* $\sigma \xrightarrow{S}_n \tau$ *then for all* $m \geq n$*, we may infer* $\sigma \xrightarrow{S}_m \tau$.

Proof by induction on the derivation of $\sigma \xrightarrow{S}_n \tau$. □

➥ p. 164

Lemma 3.11 (Bounded Recursive Depth)

1. *If a program terminates, the recursive depth must be finite i.e., if* $\sigma \xrightarrow{S} \tau$ *then there exists a bound* n *such that* $\sigma \xrightarrow{S}_n \tau$.
2. *Bounded recursive depth is a mere annotation of operational semantics i.e., for an arbitrary depth* n, $\sigma \xrightarrow{S}_n \tau$ *implies* $\sigma \xrightarrow{S} \tau$.

Proof

1. by induction on the derivation of $\sigma \xrightarrow{S} \tau$. We need to specify a concrete recursive depth bound. If $\sigma \xrightarrow{S} \tau$ has been derived by one of the axioms (2.4), (2.5) or (2.9), we choose 0. The induction hypotheses for conditional and procedure invocation predetermine the overall recursive depth.

   Consider the case of sequential composition. From the induction hypotheses

   $$\exists n \cdot \sigma \xrightarrow{S_1}_n \tau \qquad \text{and} \qquad \exists n \cdot \tau \xrightarrow{S_2}_n \eta$$

   we want to conclude $\exists n \cdot \sigma \xrightarrow{S_1;\, S_2}_n \eta$.

   Let n_1 and n_2 be recursive depth bounds such that

   $$\sigma \xrightarrow{S_1}_{n_1} \tau \qquad \text{and} \qquad \tau \xrightarrow{S_2}_{n_2} \eta \ .$$

   Resorting to Lemma 3.10, we weaken these to

   $$\sigma \xrightarrow{S_1}_{\max(n_1,n_2)} \tau \qquad \text{and} \qquad \tau \xrightarrow{S_2}_{\max(n_1,n_2)} \eta \ .$$

   Finally, the rule for sequential composition unites them to

   $$\sigma \xrightarrow{S_1;\, S_2}_{\max(n_1,n_2)} \eta \ .$$

2. by induction on the derivation of $\sigma \xrightarrow{S}_n \tau$

□

The MGF proof is conducted by induction on the structure of programs. We abbreviate

$$\begin{aligned} \psi(n)(Z,\sigma) &\stackrel{\text{def}}{=} \sigma \xrightarrow{\textbf{call}}_{n+1} Z \\ \hat{\psi}(n)(Z,\sigma) &\stackrel{\text{def}}{=} \exists u : \mathtt{nat} \cdot \psi(u)(Z,\sigma) \wedge u < n \\ q(Z,\tau) &\stackrel{\text{def}}{=} Z = \tau \ . \end{aligned}$$

In the case of a procedure invocation, we cannot appeal to an induction hypothesis. Instead we derive the following pseudo-induction hypothesis which allows us to apply the rule for procedure calls (3.10).

➥ p. 165 **Lemma 3.12 (Completeness Step)** *For an arbitrary recursive depth n, one may derive*

$$\{\hat{\psi}(n)\}\ \mathbf{call}\ \{q\} \vdash_{\text{Hoare}} \{\psi(n)\}\ S_0\ \{q\} \quad .$$

Proof Appealing to the consequence rule and the definition of bounded recursive depth, we equivalently transform the proof obligation into

$$\{\hat{\psi}(n)\}\ \mathbf{call}\ \{q\} \vdash_{\text{Hoare}} \left\{\lambda(Z,\sigma)\cdot\sigma \xrightarrow{\quad S_0 \quad}_n Z\right\}\ S_0\ \{q\} \quad .$$

This presentation makes induction more amenable to the structure of the procedure body S_0.

The cases where S_0 is not a procedure invocation are isomorphic to the corresponding proofs of the MGF property 2.10.3.3 on page 40. One only needs to ensure that recursive depth is preserved. In particular, the correctness formula in the context is ignored.

In the case of $S_0 = \mathbf{call}$, from being able to access the context

$$\{\hat{\psi}(n)\}\ \mathbf{call}\ \{q\} \vdash_{\text{Hoare}} \{\hat{\psi}(n)\}\ \mathbf{call}\ \{q\} \quad ,$$

we need to show

$$\{\hat{\psi}(n)\}\ \mathbf{call}\ \{q\} \vdash_{\text{Hoare}} \left\{\lambda(Z,\sigma)\cdot\sigma \xrightarrow{\quad \mathbf{call} \quad}_n Z\right\}\ \mathbf{call}\ \{q\} \quad . \tag{3.13}$$

To arrive at (3.13), we appeal to the rule of consequence. As a side-condition, given arbitrary states τ and σ such that $\sigma \xrightarrow{\mathbf{call}}_n \tau$, we have to find a suitable recursive depth $u < n$ such that $\sigma \xrightarrow{\mathbf{call}}_{u+1} \tau$. We choose $u = n-1$. Notice that n cannot have been 0 due to the definition of recursive depth.

□

It is now easy to prove the MGF Theorem:

Proof of Theorem 3.8 by induction on the structure of S. In the case of a procedure invocation, we have to establish

$$\vdash_{\text{Hoare}} \left\{\lambda(Z,\sigma)\cdot\sigma \xrightarrow{\quad \mathbf{call} \quad} Z\right\}\ \mathbf{call}\ \{q\} \quad . \tag{3.14}$$

Applying the procedure invocation rule to the pseudo-hypothesis of Lemma 3.12, we may derive

$$\vdash_{\text{Hoare}} \{\lambda(Z,\tau)\cdot\exists n : \mathtt{nat}\cdot\psi(n)(Z,\sigma)\}\ \mathbf{call}\ \{q\} \quad .$$

The consequence rule mediates the differences in the preconditions and, given arbitrary states τ and σ such that $\sigma \xrightarrow{\mathbf{call}} \tau$, one has to find a depth u such that $\sigma \xrightarrow{\mathbf{call}}_{u+1} \tau$. Applying Lemma 3.11.1 to the assumption, we may extract a depth n satisfying $\sigma \xrightarrow{\mathbf{call}}_n \tau$. Again, we choose $u = n-1$ as n cannot have been 0 due to the definition of recursive depth. □

For completeness, we are only interested in top-level correctness formulae in the empty context. Specifically, the presentation of Figure 3.1 on page 63 is not complete for arbitrary contexts. The rule for procedure calls has been tailored to only yield atomic correctness formulae.

➥ p. 165 **Corollary 3.13 (Completeness)** *Whenever* $\vdash_{\text{Hoare}} \{p\}\ S\ \{q\}$ *is derivable from the axioms and rules of Figure 3.1 on page 63,* $\models_{\text{Hoare}} \{p\}\ S\ \{q\}$ *holds.*

Proof Completeness follows immediately from the MGF Theorem. See the proof of Corollary 2.31 on page 41 for further details. □

3.1.7 An Example Derivation

In this section, we employ the new set of rules to prove that an algorithm to compute $y = x!$ leaves the input x invariant. In particular, we give an example of how to invoke the rule for recursive procedures for a suitable termination measure. The algorithm can be shown correct as easily with Sokołowski's (1977) rule for procedures. For a more complex example, we refer to the case study of Chapter 4.

➥ p. 165 **Lemma 3.14 (Invariance Property of the Factorial Algorithm)** *For the declaration of the procedure in Example 3.2 on page 58, we may derive*

$$\vdash_{\text{Hoare}} \{x = X\}\ \textbf{call}\ \{x = X\} \quad .$$

An interesting aspect of this proof is that our stronger rule of consequence is essential. In particular, Apt (1981) has proven that if one only uses Hoare's (1969) consequence rule, Lemma 3.14 does not hold and thus one would have a seriously incomplete verification calculus. The proof also motivates that in practice it is convenient to be able to use a rule of adaptation to compute preconditions. It will turn out that such a rule is merely a trivial instance of our new rule of consequence.

Proof Termination is guaranteed because the recursive depth is bounded by the initial value of the input parameter x. Employing the consequence rule, we record the convergence measure

$$\vdash_{\text{Hoare}} \{\exists n \cdot p(n)\}\ \textbf{call}\ \{x = X\} \qquad \text{with } p(n) \stackrel{\text{def}}{=} x = X \wedge n = x.$$

Thus the way has been smoothed to invoke the procedure invocation rule[2] (3.10) leading to the proof obligation

$$\Gamma_n \vdash_{\text{Hoare}} \{p(n)\}\ S_0\ \{x = X\}$$
$$\text{where } \Gamma_n = \{\exists u \cdot p(u) \wedge u < t\}\ \textbf{call}\ \{x = X\} \text{ for an arbitrary bound } n. \quad (3.15)$$

We may retrieve the information in the context by axiom (3.9)

$$\Gamma_n \vdash_{\text{Hoare}} \{\exists u \cdot p(u) \wedge u < t\}\ \textbf{call}\ \{x = X\} \quad . \quad (3.16)$$

The strategy is now to successively apply the Hoare Logic rules corresponding to the outermost constructor of the desired proof obligation (3.15). A proof obligation referring to a procedure invocation can be tackled by the assumption (3.16).

We commence with the rule for conditionals. The **then** branch is easily resolved. Otherwise, we have to show

$$\Gamma_n \vdash_{\text{Hoare}} \begin{array}{c} \{\exists u \cdot p(u) \wedge u < t \wedge x \neq 0\} \\ x := x - 1;\ \textbf{call};\ x := x + 1;\ y := y * x \\ \{x = X\} \end{array} \quad .$$

Since the assignment axiom is left-constructive (Dahl 1992) in that it accepts an arbitrary postcondition, a sound strategy[3] is to unroll the program from behind. Thus, when the rule for sequential composition leads to branching in the proof obligation, one ought to concentrate on the second. Eventually, this approach leads to the two goals

$$\Gamma_n \vdash_{\text{Hoare}} \{\exists u \cdot p(u) \wedge u < t \wedge x \neq 0\}\ x := x - 1\ \{?i\}$$
$$\Gamma_n \vdash_{\text{Hoare}} \{?i\}\ \textbf{call}\ \{x + 1 = X\} \quad (3.17)$$

for some intermediate assertion $?i$. Given a concrete instance, we can employ the rule of consequence to derive (3.17) from (3.16). Since $?i$ is a postcondition in a further proof obligation, it would be best if $?i$ would be as weak as the consequence rule admits. In other words, it would be pragmatically useful to also have a left-constructive version of the rule of consequence. We discuss such rules, also called rules of adaptation, in more detail in the following section. A suitable assertion would be $?i \stackrel{\text{def}}{=} x + 1 = X$. In the next section, we present a rule which automatically computes an equivalent version. □

[2] To successfully apply Sokołowski's (1977) version, one has to ensure that $p(0)$ does not hold. A suitable choice would be $p(n) \stackrel{\text{def}}{=} x = X \wedge n = x + 1$, because all variables are declared as natural numbers.

[3] in particular, for Verification Condition Generators

Notice that regardless of the choice for $?i$ in the above proof, Hoare's (1969) rule of consequence would fail at this point, because the postconditions of (3.16) and (3.17) are incompatible. We had already discussed this deficiency of the standard presentation of Hoare Logic in Sect. 2.10.3. It is an intrinsic problem for which an inadequate treatment of auxiliary variables is to blame.

Without procedures, completeness is not threatened because a mismatch of auxiliary variables such as (3.28) can always be reconciled by rebuilding the complete derivation. One may compensate by readjusting the auxiliary variables accordingly in the leaves of the proof tree. Whenever axiom (2.71) or (2.72) is encountered one instead chooses an equivalent version where auxiliary variables have been shifted. Such a patch cannot be applied in the presence of rule (3.8). It also introduces an axiom i.e., $\{\exists u \cdot p(u) \wedge u < t\}$ **call** $\{q\}$ for some $t : (W, <)$, but, unlike the axiom for the empty program or an assignment, the pre- and postcondition is bound by the *specific* assertions of a correctness formula in the derivation.

In particular, completeness could be achieved in the standard presentation by integrating our new consequence rule in the procedure invocation rule. We will discuss a similar rule in the following section. However, we feel that pragmatically, it is better to provide an improved consequence rule at all times.

Previously, a common approach has been to retain Hoare's consequence rule and adopt a flurry of further rules to achieve completeness e.g.,

> " *It will turn out that we need five additional [rules], before a completeness theorem becomes possible.* " (de Bakker 1980)

Perhaps even more worrying is that fact that *four* of these are actually employed in the verification of the invariance property of the factorial function. In such a setting, using Hoare Logic to derive properties of even simple programs appears to be unnecessarily complicated.

3.1.8 Rules of Adaptation

Among the additional rules to add in order to retain completeness for imperative programs dealing with recursive procedures, a rule of adaption has been considered by various authors e.g., (Hoare 1971, Morris n.d., London, Guttag, Horning, Lampson, Mitchell & Popek 1978, Gries & Levin 1980).

In general, rules of adaptation are of the form

$$\frac{\Gamma \vdash_{\text{Hoare}} \{p_1\}\, S\, \{q_1\}}{\Gamma \vdash_{\text{Hoare}} \{p\}\, S\, \{q\}}$$

for arbitrary assertions p_1, q_1, q, and particular proposals for the adapted precondition p. Ideally, the rule should be left-maximal (Dahl 1992) i.e., the precondition p

should be the weakest possible so that

$$\forall S \cdot \models_{\text{Hoare}} \{p_1\}\ S\ \{q_1\} \Rightarrow \models_{\text{Hoare}} \{p\}\ S\ \{q\} \tag{3.18}$$

holds.

Not distinguishing between program and auxiliary variables, Hoare (1971) has proposed

$$p(\sigma) \stackrel{\text{def}}{=} \exists Z_1 \cdot p_1(\sigma) \wedge \forall \tau \cdot q_1(\tau) \Rightarrow q(\tau) \tag{3.19}$$

where Z_1 is a list of all variables[4] free in p_1, q_1, but not in q.

However, while adding Hoare's rule of adaptation leads to a complete verification calculus, Morris (n.d.) and Olderog (1983) have shown that, in general, (3.19) is not the weakest possible. Morris (n.d.) points out two[5] instructive counter examples:

Example 3.15 *Let*

$$p_1(\sigma) \stackrel{\text{def}}{=} q_1(\sigma) \stackrel{\text{def}}{=} \sigma(Z) > 0 \wedge \sigma(x) > 0$$
$$q(\sigma) \stackrel{\text{def}}{=} \sigma(Z) \geq 0 \wedge \sigma(x) > 0$$

where Z does not occur in the program under consideration. The weakest precondition is then given by $\lambda\sigma \cdot \sigma(Z) \geq 0 \wedge \sigma(x) > 0$*. However, (3.19) requires the stronger* $\lambda\sigma \cdot \sigma(Z) > 0 \wedge \sigma(x) > 0$ *(modulo equivalence transformations), because the auxiliary variable Z occurs in both premiss and conclusion.*

This problem can be solved by formally treating assertions as relations of auxiliary variables and states i.e.,

$$p(Z, \sigma) \stackrel{\text{def}}{=} \exists Z_1 \cdot p_1(Z_1, \sigma) \wedge \forall \tau \cdot q_1(Z_1, \tau) \Rightarrow q(Z, \tau) \quad . \tag{3.20}$$

A corresponding rule of adaptation (3.21) is obviously a trivial instance of our rule

$$\frac{\vdash_{\text{Hoare}} \{p_1\}\ S\ \{q_1\}}{\vdash_{\text{Hoare}} \{\lambda(Z, \sigma) \cdot \exists Z_1 \cdot p_1(Z_1, \sigma) \wedge \forall \tau \cdot q_1(Z_1, \tau) \Rightarrow q(Z, \tau)\}\ S\ \{q\}} \tag{3.21}$$

Figure 3.3: Rule of Adaptation

of consequence (2.70). Conversely, if one replaces the new consequence rule by the

[4] In practice, this would be a list of *auxiliary* program variables.

[5] A third one only applies to partial correctness

rule of adaptation *and* Hoare's (1969) rule of consequence, one retains a sound and complete verification calculus.

Morris' second example shows that the choice for the auxiliary variable Z_1 may have to depend on the value of variables in the final state:

Example 3.16 *Let*

$$\begin{aligned} p_1(Z,\sigma) &\stackrel{\text{def}}{=} Z = 0 \vee Z = 1 \\ q_1(Z,\tau) &\stackrel{\text{def}}{=} \tau(x) \neq Z \\ q(Z,\tau) &\stackrel{\text{def}}{=} \sigma(x) \neq 0 \wedge \sigma(x) \neq 1 \end{aligned}$$

The weakest precondition is then given by **true**. *However, both (3.19) and the weaker (3.20) are equivalent to* **false**.

Relaxing the precondition p so that the witness Z_1 can benefit from inspecting the final value of program variables according to τ leads to

$$p(Z,\sigma) \stackrel{\text{def}}{=} \forall \tau \cdot \exists Z_1 \cdot p_1(Z_1)(\sigma) \wedge (q_1(Z_1)(\tau) \Rightarrow q(Z)(\tau)) \quad . \tag{3.22}$$

We do not know if this is the weakest possible precondition for a rule of adaptation in the setting of total correctness. Thus, we can also not answer the question if its corresponding rule of consequence is optimal. We have used the weaker version (3.21) throughout this thesis. The problematic issue raised in the last example did not surface neither in any of the soundness and completeness proofs, nor in any of the examples.

It seems to be easier to design and verify an optimal rule of adaptation for partial correctness. In particular, Morris (n.d.) has been able to show that

$$p(Z,\sigma) \stackrel{\text{def}}{=} \forall \tau \cdot \big(\forall Z_1 \cdot p_1(Z_1,\sigma) \Rightarrow q_1(Z_1,\tau)\big) \Rightarrow q(Z,\tau)$$

is the weakest possible precondition[6] which satisfies (3.18). This version is however only sound when one considers *partial* correctness.

3.1.9 Mutually Recursive Procedures

Sokołowski (1977) sketched an extension to mutual recursion of which properties have to be derived simultaneously for *all* procedures with the *same* recursion-depth counter. Let *Proc* denote a (finite) set of procedure identifiers such that for each $P : Proc$, the program S_P stands for the body of the procedure P. Following Sokołowski (1977), one might then extend our well-founded variant (3.8) to

$$\frac{\forall P : Proc \cdot \forall t : W \cdot \Gamma_t \vdash \{p(t)\}\ S_P\ \{q\}}{\forall P : Proc \cdot \{\exists t : W \cdot p(t)\}\ \textbf{call}\ P\ \{q\}} \qquad \begin{array}{l}\text{where } (W,<) \text{ is well-founded} \\ \text{and, for all } P \in Proc,\ \big(\{\exists u : W \cdot p(u) \wedge u < t\}\ \textbf{call}\ P\ \{q\}\big) \in \Gamma_t.\end{array}$$

[6] A more elaborate version has been published by Olderog (1983).

Pandya & Joseph (1986) have pointed out that this rule is difficult to apply, because it forces one to find termination measures across multiple procedure invocations. In their version, one may appeal to some correctness formulae in the context Γ_t without having to worry about decreasing the termination measure. Specifically, one may choose a subset of *header* procedures $H \subseteq \texttt{Prop}$ such that every cycle in the procedure call graph must contain at least one header procedure. Then, a decrease of the termination measure needs only be established for header procedures i.e.,

$$\Gamma_t = \left\{ \{\exists u : W \cdot p(u) \wedge u < t\} \ \mathbf{call}\ P\ \{q\} \;\middle|\; P \in H \right\} \bigcup \left\{ \{p(t)\} \ \mathbf{call}\ P\ \{q\} \;\middle|\; P \in Proc \setminus H \right\}$$

A further improvement would be to allow different termination measures for different procedures, where assumptions about the correctness of procedure calls can be gradually accumulated. As a *new* rule, we propose

$$\frac{\forall t : W \cdot \Gamma, \{\exists u : W \cdot p(u) \wedge u < t\} \ \mathbf{call}\ P\ \{q\} \vdash \{p(t)\}\ S_P\ \{q\}}{\Gamma \vdash \{\exists t : W \cdot p(t)\} \ \mathbf{call}\ P\ \{q\}} \quad \text{where } (W, <) \text{ is well-founded.} \quad (3.23)$$

Assume that P and Q are two mutually recursive procedures and we want to derive a property of **call** P (in the empty context). Applying the above rule (3.23), we consider deriving S_P relative to a context containing a correctness formula about **call** P. Unrolling the procedure body of S_P, a call to itself can be directly dealt with by appealing to the context. If S_P invokes the procedure Q, we again employ the rule for procedure invocations (3.23). Hence, with a context containing correctness formulae about **call** P and **call** Q, we may resolve the remaining proof obligation concerning the procedure body of Q.

A much more radical approach for dealing with recursive procedures has been advocated by Homeier (1995). Building on ideas of Pandya & Joseph (1986) to analyse the graph structure of recursive calls, he extends the notion of correctness formulae to e.g., also include *Entrance Specifications* $\{p\}\ S \to P\ \{q\}$ and *Path Entrance Specifications* $\{p\}\ P_1 - ps - P_n\ \{q\}$. Intuitively

$\{p\}\ S \to P\ \{q\}$ holds if the program S is executed in a state satisfying p, then if at any point within S procedure P is called, then at the entry of P, q is satisfied.

$\{p\}\ P_1 - ps - P_n\ \{q\}$ holds if execution begins at the entry of the procedure P_1 in a state satisfying p, and if in the execution of the body S_1, procedure calls are made successively deeper to the procedures listed in ps, and finally a call is made to the procedure P_n, then at that entry of P_n, q is satisfied.

Such an extension leads to a considerable blow up in the number of verification rules. Further investigations are needed to compare the suitability of a verification calculus with a rule such as (3.23) with Homeier's (1995) system.

3.1.10 The Vienna Development Method

Extending VDM with recursive procedures follows exactly the same paradigm as for Hoare Logic. We introduce contexts along with an axiom for looking up correctness formulae in contexts and supply a rule for recursive procedure invocations. The new rules are identical to those for Hoare Logic, except that preconditions are unary predicates on the state space. Thus, the proofs of soundness and completeness can be carried out with the same techniques discussed for Hoare Logic.

➥ p. 170 **Theorem 3.17 (Soundness)** *The rules of Figure 3.4 on the next page are sound.*

Proof by induction on the structure of programs. □

➥ p. 171 **Theorem 3.18 (Completeness)** *The rules of Figure 3.4 on the following page are complete.*

Proof This is a straightforward consequence of the MGF property which can be established along the very same lines as in the setting of Hoare Logic. □

3.2 Local Program Variables

To simplify matters, we consider local program variables *without* procedures in this section. Section 3.3 will combine these two features.

To cope with run-time storage allocation in which, due to shadowing, program variables are no longer unique identifiers, one usually extends the notion of states and considers them either as stacks or introduces addresses. Exploiting an implicit stack mechanism (Sieber 1981, Olderog 1981), in our formalisation, we are able to work with essentially the same representation of states as in the case of simple imperative programs.

We start with an informal view of local program variables. We then go on to discuss the new requirements of the state space so that we may formally define syntax and semantics for imperative programs with local program variables. Focussing on Hoare Logic, in Section 3.2.4, we present a new rule for initialised local program variables and establish soundness and completeness. Finally, we sketch a similar extension for VDM.

$$\{p\}\ \textbf{call}\ \{q\} \vdash \{p\}\ \textbf{call}\ \{q\}$$

$$\Gamma \vdash \{\lambda\sigma \cdot \textbf{true}\}\ \textbf{skip}\ \{\lambda(\tau,\sigma) \cdot \tau = \sigma\}$$

$$\Gamma \vdash \{\lambda\sigma \cdot \textbf{true}\}\ x := t\ \{\lambda(\tau,\sigma) \cdot \tau = \sigma[x \mapsto \text{eval}(\sigma)(t)]\}$$

$$\frac{\Gamma \vdash \{p_1\}\ S_1\ \{\lambda(\tau,\sigma) \cdot p_2(\tau) \wedge r_1(\tau,\sigma)\} \quad \Gamma \vdash \{p_2\}\ S_2\ \{r_2\}}{\Gamma \vdash \{p_1\}\ S_1;\ S_2\ \{r_2 \circ r_1\}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \{\lambda\sigma \cdot p(\sigma) \wedge \text{eval}(\sigma)(b) = \textbf{true}\}\ S_1\ \{q\} \\ \Gamma \vdash \{\lambda\sigma \cdot p(\sigma) \wedge \text{eval}(\sigma)(b) = \textbf{false}\}\ S_2\ \{q\}\end{array}}{\Gamma \vdash \{p\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{q\}}$$

$$\frac{\Gamma \vdash \{\lambda\sigma \cdot p(\sigma) \wedge \text{eval}(\sigma)(b) = \textbf{true}\}\ S\ \{\lambda(\tau,\sigma) \cdot p(\tau) \wedge \mathit{sofar}(\tau,\sigma)\}}{\Gamma \vdash \{p\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{q\}}$$

where *sofar* is any binary transitive relation on the state space Σ such that

$$\lambda(\tau,\sigma) \cdot \mathit{sofar}(\tau,\sigma) \wedge p(\tau) \text{ is well-founded}$$

and $q(\tau,\sigma) \stackrel{\text{def}}{=} p(\tau) \wedge \text{eval}(\tau)(b) = \textbf{false} \wedge \big(\mathit{sofar}(\tau,\sigma) \vee \tau = \sigma\big)$.

$$\frac{\forall t : W \cdot \{\lambda\sigma \cdot \exists u : W \cdot p(u)(\sigma) \wedge u < t\}\ \textbf{call}\ \{q\} \vdash \{p(t)\}\ S_0\ \{q\}}{\vdash \{\lambda\sigma \cdot \exists t : W \cdot p(t)(\sigma)\}\ \textbf{call}\ \{q\}}$$

where $(W, <)$ is well-founded.

$$\frac{\Gamma \vdash \{p_1\}\ S\ \{q_1\}}{\Gamma \vdash \{p\}\ S\ \{q\}}$$

provided $\forall\sigma \cdot p(\sigma) \Rightarrow p_1(\sigma)$ and $\forall(\tau,\sigma) \cdot p(\sigma) \Rightarrow q_1(\tau,\sigma) \Rightarrow q(\tau,\sigma)$.

Figure 3.4: VDM and Recursive Procedures

3.2.1 Initialisation

We introduce local program variables by the constructor **new** $x := t$ **in** S where x is a program variable, t is an expression with *arbitrary* type T and S is a program. Programs of the above form are also called *blocks*. Intuitively, the expression t is evaluated, its value assigned to the variable x and in this updated context, the program S is executed. At the end, the original value of x is restored.

Many real programming languages also allow blocks of the form **new** x **in** S where a suitable amount of memory is allocated, but x may or may not be initialised. The standard approach in program verification is to treat uninitialised variables as a special case of **new** $x := t$ **in** S where t denotes a new canonical value[7] ω_T (Apt 1981). We only consider (well-formed) initialised local program variables.

> " *After all, forgetting to initialize is one of the most common sources of programming error; and if there is no default initialization, debugging is made difficult by the fact that program behaviour is unpredictable in terms of the program text, and that it may even depend on the previous contents of the computer memory. Uninitialized pointer variables are especially troublesome.* " (Dahl (1992))

Surprisingly, some languages such as C which support initialised variables insist on first allocating the new variable x and only then evaluate the expression t (Schildt et al. 1990). Of course this also leads to undefined values if the variable x is contained in t.

Example 3.19 *Employing the compiler gcc 2.7.2, the program*

```
void main() {
  int c=103; {
    int c=c+1;
    printf ("%i\n",c);
  }}
```

prints out 5 *instead of the expected* 104. *Java is conservative over C in that initialisations which refer to the value of a program variable to be defined are considered as not being well-formed. Perl is more flexible and the program*

```
#!/usr/local/bin/perl

$c=103; {
```

[7] To avoid having to explicitly deal with such undefined values, de Bakker (1980) suggests to restrict the class of programs with uninitialised values to those where program variables may only be referenced *after* a suitable assignment has taken place.

```
local($c) = $c+1;
print "$c\n"; }
```

yields indeed 104.

3.2.2 States with Dynamic Types

With local program variables, representing the memory as a map from program variables to values of corresponding type is no longer adequate, because one may have multiple copies of program variables with identical names, but different values (and scopes). One usually refines the notion of states,

- either by introducing addresses (de Bakker 1980),
- or by explicitly modelling stacks (Clarke Jr. 1979).

3.2.2.1 Addresses

Addresses serve as unique handles for all program variables. Hence, instead of representing states as functions from program variables to values, states are functions from addresses to values. Scoping is controlled by an environment which is simply a map from program variables to addresses.

Let σ be an instance of the state and E an environment. To lookup the value of the program variable x, one needs to combine the two views of the memory i.e. $\sigma(E(x))$. Thus, evaluation has to additionally take scoping into account. To allocate a new local variable **new** $x := t$ **in** S, one needs to find a fresh address l to update the state space $\sigma[l \mapsto \text{eval}(E,\sigma)(t)]$ and the environment $E[x \mapsto l]$. Afterward executing S, one reinstates the previous environment E.

3.2.2.2 Explicit Stacks

With this approach, the state space is a stack where entries are tuples of program variables and their values. To lookup the value of the program variable x, one needs to access the most recent entry in the stack where the first component is x and output the second component. Thus, scoping and evaluation of terms is determined by the order on the stack. To allocate a new local program variable **new** $x := t$ **in** S, one needs to evaluate the term t and push $(x, \text{eval}(\sigma)(t))$ onto the stack. After having executed the block, one needs to remove the last entry in the stack.

3.2.2.3 Implicit Stacks

Without procedures, one can work with a more naive notion of states which only records the value of all program variables in scope. Stack behaviour is achieved implicitly (Sieber 1981, Olderog 1981).

Let σ be such a state. When considering the execution behaviour of a block **new** $x := t$ **in** S commencing in state σ, it suffices to consider running the program S in the state $\sigma[x \mapsto \mathrm{eval}(\sigma)(t)]$, because the old entry for x cannot be accessed by the program S. The final state of the block is identical to that of the program S, except that the entry for the program variable x needs to be updated by its original value $\sigma(x)$.

When adding other language features, implicit stacks may no longer be adequate. In particular, when one considers procedures with call-by-name parameters, due to aliasing, one is forced to abandon the implicit stack encoding[8]. However, restricting our attention to parameterless[9] recursive procedures in this thesis, it suffices to work with minor variants of implicit stacks.

Previously, see Def. 2.2 on page 12, states were given relative to a *fixed* variable declaration $\mathrm{sort} : \mathbf{VAR} \to \mathtt{Type}$. However, in a block **new** $x := t$ **in** S, the sort of x is not determined by a previous variable declaration. Hence, sorts are no longer merely a parameter of states. Instead, states have to be treated as a family of functions over sorts.

➥ p. 171 **Definition 3.20 (State Space with Dynamic Types)** $\Sigma(\mathrm{sort}) \stackrel{\mathrm{def}}{=} \prod x : \mathbf{VAR} \cdot \mathrm{sort}(x)$

In particular, updating may change the sort i.e., for $t : T$ and $\sigma : \Sigma(\mathrm{sort})$ for some $\mathrm{sort} : \mathbf{VAR} \to \mathtt{Type}$, the updated state $\sigma[x \mapsto t]$ inhabits $\Sigma(\mathrm{sort}[x \mapsto T])$. We also need to revise our previous definitions of expressions and assertions:

➥ p. 171 **Definition 3.21 (Expressions)**

$$\mathrm{expression}(\mathrm{sort} : \mathbf{VAR} \to \mathtt{Type})(T : \mathtt{Type}) \stackrel{\mathrm{def}}{=} \big(\Sigma(\mathrm{sort})\big) \to T \quad .$$

➥ p. 172 **Definition 3.22 (Assertions)**

$$\mathrm{Assertion}(\mathrm{sort} : \mathbf{VAR} \to \mathtt{Type})(T : \mathtt{Type}) \stackrel{\mathrm{def}}{=} \big(T \times \Sigma(\mathrm{sort})\big) \to \mathtt{Prop}$$

[8] Ah-kee (1990) has demonstrated that the address mechanism is suitable to handle aliasing.
[9] Call-by-value parameters would not cause any problems.

3.2.3 Programs with Dynamic Sorts

The BNF grammar

$$S ::= \textbf{skip} \mid x := t \mid S_1;\ S_2 \mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \mid \textbf{while } b \textbf{ do } S \mid \textbf{new } x := t \textbf{ in } S$$

does not do justice to the dynamic rôle of program variable declarations. The type-theoretic representation where every constructor is annotated by its type appears to be more informative. One may formulate the empty program and assignment for an arbitrary sort. Apart from the block constructor, all other language constructors simply preserve sorts.

➥ p. 172 **Definition 3.23 (Syntax)** *Imperative programs* `prog` *are defined as an inductive family over* $\text{sort} : \textbf{VAR} \to \texttt{Type}$ *by*

$$\textbf{skip} : \texttt{prog}(\text{sort})$$

$$x := t : \texttt{prog}(\text{sort}) \quad \textit{where } t : \text{expression}(\text{sort})(\text{sort}(x))$$

$$\frac{S_1, S_2 : \texttt{prog}(\text{sort})}{S_1;\ S_2 : \texttt{prog}(\text{sort})}$$

$$\frac{S_1, S_2 : \texttt{prog}(\text{sort})}{\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 : \texttt{prog}(\text{sort})} \quad \textit{where } b : \text{expression}(\text{sort})(\texttt{bool})$$

$$\frac{S : \texttt{prog}(\text{sort})}{\textbf{while } b \textbf{ do } S : \texttt{prog}(\text{sort})} \quad \textit{where } b : \text{expression}(\text{sort})(\texttt{bool})$$

$$\frac{S : \texttt{prog}(\text{sort}[x \mapsto T])}{\textbf{new } x := t \textbf{ in } S : \texttt{prog}(\text{sort})}$$

where $t : \text{expression}(\text{sort})(T)$ *for an arbitrary type* T

➥ p. 173 **Definition 3.24 (Structural Operational Semantics)** *We extend Def. 2.8 on page 16 with a rule for blocks*

$$\frac{\sigma[x \mapsto \text{eval}(\sigma)(t)] \xrightarrow{\quad S \quad} \tau}{\sigma \xrightarrow{\ \textbf{new } x := t \textbf{ in } S\ } \tau[x \mapsto \sigma(x)]} \tag{3.24}$$

where for arbitrary type T *and declaration* $\text{sort} : \textbf{VAR} \to \texttt{Type}$*, we have*

$$x : \textbf{VAR} \qquad \sigma : \Sigma(\text{sort})$$

$$t : \text{expression}(\text{sort})(T) \qquad \tau : \Sigma(\text{sort}[x \mapsto T])$$

$$S : \texttt{prog}(\text{sort}[x \mapsto T])$$

In all other rules, the sort of states is identical in the premiss and the conclusion.

3.2.4 Hoare Logic

Building on ideas of Lauer (1971), Apt (1981) suggests the following rule

$$\frac{\{p[x \mapsto y] \wedge x = \omega\}\ S\ \{q[x \mapsto y]\}}{\{p\}\ \textbf{new}\ x\ \textbf{in}\ S\ \{q\}} \qquad \text{where } y \notin \mathit{free}(p,S,q)$$

where ω stands for a special constant denoting an undefined value.

Rather than substituting a fresh *program* variable y for the newly declared local program variable x, we preserve the original value of x in the assertions by updating both pre- and postcondition with a constant v which we introduce by universal quantification[10] in the premiss. Scoping of the implicitly universally quantified p, S and q ensures that $v \notin \mathit{free}(p,S,q)$. Furthermore, we consider initialised blocks:

$$\frac{\forall v \cdot \{p[x \mapsto v] \wedge x = t[x \mapsto v]\}\ S\ \{q[x \mapsto v]\}}{\{p\}\ \textbf{new}\ x := t\ \textbf{in}\ S\ \{q\}} \tag{3.25}$$

➥ p. 178

Theorem 3.25 (Soundness) *The set of rules in Figure 3.5 on the next page is sound.*

Proof by induction on the derivation of $\vdash_{\text{Hoare}} \{p\}\ S\ \{q\}$. In the new case of blocks, we fix arbitrary $Z : T$ and $\sigma : \Sigma(\text{sort})$ such that $p(Z,\sigma)$ holds. From the induction hypothesis

$$\forall v \cdot \models_{\text{Hoare}} \begin{array}{c} \{\lambda(Z,\sigma) \cdot p(Z,\sigma[x \mapsto v]) \wedge \sigma(x) = \text{eval}(\sigma[x \mapsto v])(t)\} \\ S \\ \{\lambda(Z,\tau) \cdot q(Z,\tau[x \mapsto v])\} \end{array}$$

we have to produce a final state τ such that $\sigma \xrightarrow{\textbf{new}\ x := t\ \textbf{in}\ S} \tau$ and $q(Z,\tau)$ holds. Instantiating the induction hypothesis with $\sigma(x)$ for v and $\sigma[x \mapsto \text{eval}(\sigma)(t)]$ for the initial state yields a final state η such that

$$\sigma[x \mapsto \text{eval}(\sigma)(t)] \xrightarrow{\quad S \quad} \eta \qquad \text{and} \qquad q(Z,\eta[x \mapsto \sigma(x)])$$

holds. Hence, appealing to the operational semantics of blocks, we choose

$$\tau \stackrel{\text{def}}{=} \eta[x \mapsto \sigma(x)] \ .$$

□

[10] Alternatively, just like in the case of loops on page 38, one could have extended the domain of auxiliary variables in the premiss to avoid universal quantification.

$$\vdash_{\text{Hoare}} \{p\}\ \textbf{skip}\ \{p\}$$

$$\vdash_{\text{Hoare}} \left\{\lambda(Z,\sigma)\cdot p\big(Z,\sigma[x\mapsto \text{eval}(\sigma)(t)]\big)\right\}\ x:=t\ \{p\}$$

$$\frac{\vdash_{\text{Hoare}} \{p\}\ S_1\ \{r\} \qquad \vdash_{\text{Hoare}} \{r\}\ S_2\ \{q\}}{\vdash_{\text{Hoare}} \{p\}\ S_1;\ S_2\ \{q\}}$$

$$\frac{\begin{array}{c}\vdash_{\text{Hoare}} \{\lambda(Z,\sigma)\cdot p(Z,\sigma)\wedge \text{eval}(\sigma)(b)=\textbf{true}\}\ S_1\ \{q\}\\ \vdash_{\text{Hoare}} \{\lambda(Z,\sigma)\cdot p(Z,\sigma)\wedge \text{eval}(\sigma)(b)=\textbf{false}\}\ S_2\ \{q\}\end{array}}{\vdash_{\text{Hoare}} \{p\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{q\}}$$

$$\frac{\forall t : W\cdot \vdash_{\text{Hoare}} \{\hat{p}\}\ S\ \{\lambda(Z,\tau)\cdot p(Z,\tau)\wedge u(\tau)<t\}}{\vdash_{\text{Hoare}} \{p\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{\lambda(Z,\tau)\cdot p(Z,\tau)\wedge \text{eval}(\tau)(b)=\textbf{false}\}}$$

where $\hat{p}(Z,\sigma) \stackrel{\text{def}}{=} p(Z,\sigma)\wedge \text{eval}(\sigma)(b)=\textbf{true}\wedge u(\sigma)=t$
and $(W,<)$ is well-founded.

$$\frac{\forall v\cdot\ \vdash_{\text{Hoare}}\quad \begin{array}{c}\{\lambda(Z,\sigma)\cdot p(Z,\sigma[x\mapsto v])\wedge\sigma(x)=\text{eval}(\sigma[x\mapsto v])(t)\}\\ S\\ \{\lambda(Z,\tau)\cdot q(Z,\tau[x\mapsto v])\}\end{array}}{\vdash_{\text{Hoare}} \{p\}\ \textbf{new}\ x:=t\ \textbf{in}\ S\ \{q\}} \tag{3.26}$$

$$\frac{\vdash_{\text{Hoare}} \{p_1\}\ S\ \{q_1\}}{\vdash_{\text{Hoare}} \{p\}\ S\ \{q\}}$$

where $p_1, q_1 : \text{Assertion}(U)$ and $p, q : \text{Assertion}(T)$ for arbitrary types T and U

and $\forall Z\cdot\forall\sigma\cdot p(Z,\sigma)\Rightarrow\left(\exists Z_1\cdot p_1(Z_1,\sigma)\wedge\big(\forall\tau\cdot q_1(Z_1,\tau)\Rightarrow q(Z,\tau)\big)\right)$. (3.27)

Figure 3.5: Hoare Logic for Blocks – Total Correctness

➥ p. 178 **Theorem 3.26 (MGF)** $\vdash_{\text{Hoare}} \left\{ \lambda(Z,\sigma)\cdot\sigma \xrightarrow{\quad S \quad} Z \right\} S \, \{\lambda(Z,\tau)\cdot Z = \tau\}$

Proof by induction on the structure of S. In the case of a block, from the induction hypothesis

$$\vdash_{\text{Hoare}} \left\{ \lambda(Z,\sigma)\cdot\sigma \xrightarrow{\quad S \quad} Z \right\} S \, \{\lambda(Z,\tau)\cdot Z = \tau\}$$

one has to establish

$$\vdash_{\text{Hoare}} \left\{ \lambda(Z,\sigma)\cdot\sigma \xrightarrow{\textbf{new}\ x:=t\ \textbf{in}\ S} Z \right\} \textbf{new}\ x:=t\ \textbf{in}\ S \, \{\lambda(Z,\tau)\cdot Z = \tau\} \quad .$$

Given an arbitrary $v : \text{sort}(x)$, we employ the stronger rule of consequence (3.27) to derive

$$\vdash_{\text{Hoare}} \quad \left\{ \lambda(Z,\sigma)\cdot\sigma[x\mapsto v] \xrightarrow{\textbf{new}\ x:=t\ \textbf{in}\ S} Z \wedge \sigma(x) = \text{eval}(\sigma[x\mapsto v])(t) \right\} \atop S \atop \{\lambda(Z,\tau)\cdot Z = \tau[x\mapsto v]\}$$

from which the rule for blocks (3.26) renders the proof obligation. As a side-condition, given states Z and σ such that

$$\sigma[x\mapsto v] \xrightarrow{\textbf{new}\ x:=t\ \textbf{in}\ S} Z \tag{3.28}$$

$$\sigma(x) = \text{eval}(\sigma[x\mapsto v])(t) \tag{3.29}$$

we have to find a state τ such that $\sigma \xrightarrow{\quad S \quad} \tau$ and $Z = \tau[x\mapsto v]$. Inverting the derivation of (3.28), there must be such a state τ which satisfies

$$\sigma[x\mapsto v]\,[x\mapsto \text{eval}(\sigma[x\mapsto v])(t)] \xrightarrow{\quad S \quad} \tau \atop Z = \tau[x\mapsto \sigma[x\mapsto v](x)]$$

This completes the proof, because (3.29) ensures that the state σ is (extensionally) equal to $\sigma[x\mapsto v]\,[x\mapsto \text{eval}(\sigma[x\mapsto v])(t)]$. □

➥ p. 178 **Corollary 3.27 (Completeness)** *The set of rules in Figure 3.5 on the facing page is complete.*

Proof Identical to the proof of Cor. 2.31 on page 41 for simple imperative programs. □

In practice, is is convenient to additionally be able to employ a *left-constructive* rule for blocks. For the correctness proof of Quicksort in Chapter 4, we have also used the following version

$$\frac{\forall v \cdot \{p\}\ S\ \{q[x \mapsto v]\}}{\{p[x \mapsto t]\}\ \textbf{new}\ x := t\ \textbf{in}\ S\ \{q\}} \tag{3.30}$$

which is to be understood as a pretty-printed version of the rule in Figure 3.6. It essentially assumes that the postcondition q does not mention x. Thus, one cannot expect to retain a complete system when replacing the Bottom-Up version (3.25) with this version. We are not aware of previous proposals of left-constructive rules for blocks.

$$\frac{\forall v \cdot \vdash_{\text{Hoare}} \{p\}\ S\ \{\lambda(Z,\tau) \cdot q(Z,\tau[x \mapsto v])\}}{\vdash_{\text{Hoare}} \left\{\lambda(Z,\sigma) \cdot p\big(Z,\sigma[x \mapsto \text{eval}(\sigma)(t)]\big)\right\}\ \textbf{new}\ x := t\ \textbf{in}\ S\ \{q\}}$$

Figure 3.6: Admissible Left-Constructive Rule for Blocks in Hoare Logic

➥ p. 178

Corollary 3.28 (Left-Constructive Rule for Blocks) *The rule from Figure 3.6 is admissible.*

Proof By soundness and completeness it suffices to establish the semantic counterpart

$$\frac{\forall v \cdot \models_{\text{Hoare}} \{p\}\ S\ \{\lambda(Z,\tau) \cdot q(Z,\tau[x \mapsto v])\}}{\models_{\text{Hoare}} \left\{\lambda(Z,\sigma) \cdot p\big(Z,\sigma[x \mapsto \text{eval}(\sigma)(t)]\big)\right\}\ \textbf{new}\ x := t\ \textbf{in}\ S\ \{q\}}$$

which follows immediately from the operational semantics for blocks. □

3.2.5 The Vienna Development Method

Jones (1990) suggests the right-constructive rule

$$\frac{\{p \wedge x = t\}\ S\ \{q\}}{\{p\}\ \textbf{new}\ x := t\ \textbf{in}\ S\ \{\exists v \cdot q[x \mapsto v]\}} \tag{3.31}$$

to deal with local program variables in VDM. It is, no doubt, useful in practice, but cannot handle shadowed variables. The postcondition of the conclusion cannot record any property of the value of the program variable x which is in scope before and after executing the block. Specifically, one cannot derive $\{\textbf{true}\}\ \textbf{new}\ x := t\ \textbf{in}\ S\ \{x = \overleftarrow{x}\}$. Thus, the set of rules given in Jones's (1990) book on VDM is *incomplete*.

This deficiency has been overcome in the formalisation of Ah-kee (1990). He employs an address mechanism to represent states and assertions. Environments E map

program variables to addresses. Assertions are predicates on *addresses*. Thus, one may selectively update addresses corresponding to the *new* local program variable x. For uninitialised local variables[11], he proposes

$$\frac{E\,[x \mapsto l] \mid \{p\}\ S\ \{q\}}{E \mid \{p\}\ \textbf{new}\ x\ \textbf{in}\ S\ \{\lambda(\tau,\sigma) \cdot \exists v', v \cdot q(\tau\,[l \mapsto v]\,, \sigma\,[l \mapsto v'])\}}$$

where l is a fresh address.

With the implicit stack mechanism, we discard information about intermediate local program variables. As a drawback, adding a variant of rule (3.31) does not lead to a complete verification calculus. Our solution has been to interpret the rule for Hoare Logic (3.25) in the VDM setting. This yields

$$\frac{\forall v \cdot \{p\,[x \mapsto v] \wedge x = t\,[x \mapsto v]\}\ S\ \{q\,[x \mapsto v]\}}{\{p\}\ \textbf{new}\ x := t\ \textbf{in}\ S\ \{q\}}\ .$$

Formally, this corresponds to (3.32).

➥ p. 181 **Theorem 3.29 (Soundness)** *The rules of Figure 3.7 on the following page are sound.*

Proof by induction on the derivation of correctness formulae. □

➥ p. 182 **Theorem 3.30 (Completeness)** *The rules of Figure 3.7 on the next page are complete.*

Proof Similar to the corresponding proof in the setting of Hoare Logic. □

3.3 Recursive Procedures and Static Binding

Combining procedures with blocks requires some care. A procedure invocation may occur inside a block so that, at run-time, some local variables have been allocated which conflict with program variables of the procedure in scope at declaration time. Most programming languages, including Algol, C, Cobol, Fortran, Java and Pascal, adhere to static binding[12] where a procedure invocation triggers an environmental change: A procedure may only refer to either global program variables, formal parameters, or its own local program variables. In particular, since we restrict our attention to parameterless procedures, information can only be passed to procedures via global program variables.

We deal with static binding. Some adjustments are necessary because a naïve combination of the rules of the previous two sections would lead to dynamic binding. A known remedy is to only consider programs where all defining occurrences of program

[11] restricted to the type `int`

[12] also referred to as lexicographical binding

$$\vdash_{\mathrm{VDM}} \{\lambda\sigma \cdot \mathbf{true}\}\ \mathbf{skip}\ \{\lambda(\tau,\sigma) \cdot \tau = \sigma\}$$

$$\vdash_{\mathrm{VDM}} \{\lambda\sigma \cdot \mathbf{true}\}\ x := t\ \{\lambda(\tau,\sigma) \cdot \tau = \sigma[x \mapsto \mathrm{eval}(\sigma)(t)]\}$$

$$\frac{\vdash_{\mathrm{VDM}} \{p_1\}\ S_1\ \{\lambda(\tau,\sigma) \cdot p_2(\tau) \wedge r_1(\tau,\sigma)\} \quad \vdash_{\mathrm{VDM}} \{p_2\}\ S_2\ \{r_2\}}{\vdash_{\mathrm{VDM}} \{p_1\}\ S_1;\ S_2\ \{r_2 \circ r_1\}}$$

$$\frac{\begin{array}{c}\vdash_{\mathrm{VDM}} \{\lambda\sigma \cdot p(\sigma) \wedge \mathrm{eval}(\sigma)(b) = \mathbf{true}\}\ S_1\ \{q\} \\ \vdash_{\mathrm{VDM}} \{\lambda\sigma \cdot p(\sigma) \wedge \mathrm{eval}(\sigma)(b) = \mathbf{false}\}\ S_2\ \{q\}\end{array}}{\vdash_{\mathrm{VDM}} \{p\}\ \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \{q\}}$$

$$\frac{\vdash_{\mathrm{VDM}} \{\lambda\sigma \cdot p(\sigma) \wedge \mathrm{eval}(\sigma)(b) = \mathbf{true}\}\ S\ \{\lambda(\tau,\sigma) \cdot p(\tau) \wedge \mathit{sofar}(\tau,\sigma)\}}{\vdash_{\mathrm{VDM}} \{p\}\ \mathbf{while}\ b\ \mathbf{do}\ S\ \{q\}}$$

where *sofar* is any binary transitive relation on the state space Σ such that

$$\lambda(\tau,\sigma) \cdot \mathit{sofar}(\tau,\sigma) \wedge p(\tau) \text{ is well-founded}$$

and $q(\tau,\sigma) \stackrel{\mathrm{def}}{=} p(\tau) \wedge \mathrm{eval}(\tau)(b) = \mathbf{false} \wedge \big(\mathit{sofar}(\tau,\sigma) \vee \tau = \sigma\big)$.

$$\frac{\begin{array}{c}\forall v \cdot \vdash_{\mathrm{VDM}} \quad \{\lambda\sigma \cdot p(\sigma[x \mapsto v]) \wedge \sigma(x) = \mathrm{eval}(\sigma[x \mapsto v])(t)\} \\ S \\ \{\lambda(\tau,\sigma) \cdot q(\tau[x \mapsto v], \sigma[x \mapsto v])\}\end{array}}{\vdash_{\mathrm{VDM}} \{p\}\ \mathbf{new}\ x := t\ \mathbf{in}\ S\ \{q\}} \tag{3.32}$$

$$\frac{\vdash_{\mathrm{VDM}} \{p_1\}\ S\ \{q_1\}}{\vdash_{\mathrm{VDM}} \{p\}\ S\ \{q\}}$$

provided $\forall\sigma \cdot p(\sigma) \Rightarrow p_1(\sigma)$ and $\forall(\tau,\sigma) \cdot p(\sigma) \Rightarrow q_1(\tau,\sigma) \Rightarrow q(\tau,\sigma)$.

Figure 3.7: VDM and Local Program Variables

variables are denoted differently. In this case, static and dynamic binding coincide (Olderog 1981). We do not make such a simplification. In our treatment, when declaring new variables, one may choose an *arbitrary* program variable name and instantiate it with a value of *arbitrary* type. We cater for static binding by extending implicit stacks with a backup mechanism which records the value of global variables.

We start by annotating the operational semantics with an environment to keep track of the local program variables in scope. We then follow an idea by Ah-kee (1990) to also annotate *correctness formulae* with environments. Without environments, one effectively assumes that all variables in assertions correspond to values of program variables *in scope*. Adding environments leads to more flexibility when formulating verification rules. In particular, since this technique has previously only been applied to VDM, we are able to present an improved set of Hoare Logic rules for recursive procedures and static binding. More precisely, annotating the rules of the previous two sections with environments ensures that a procedure call must take place in the empty environment and the value of all local program variables must be frozen during a call. Without environments, one needs to replace the rule for blocks by a less elegant version to retain a sound verification calculus for static binding (Apt 1981).

We show that our set of rules is sound and complete. We sketch how to achieve similar results for VDM in Sect. 3.3.8.

3.3.1 Static versus Dynamic Binding

The difference between static and dynamic binding is perhaps best explained by means of an example.

➥ p. 205 **Example 3.31 (Scoping)** *Let*

$$S_0 \stackrel{\text{def}}{=} x := z$$
$$S \stackrel{\text{def}}{=} z := 1;\ \textbf{new}\ z := 0\ \textbf{in call}$$

What should be the final value of x? Invoking the procedure S_0, should z refer to the global program variable z which is in scope at declaration time, or the most recently allocated local program variable z? For static binding, the final value of x amounts to 1, whereas dynamic binding yields 0. We restrict our attention to *static* binding which is supported by most programming languages[13]. With the set of rules we are about to introduce, we derive the correctness formula

$$\{\textbf{true}\}\ z := 1;\ \textbf{new}\ z := 0\ \textbf{in call}\ \{x = 1\}$$

in Sect. 3.3.7.6 on page 109.

[13] Exceptions are e.g., Emacs Lisp, Perl (Wall, Christiansen & Schwartz 1996) and TEX(Knuth 1986).

3.3.2 States and Environments

Can we retain the simplicity of implicit stacks if we additionally have to cope with procedure calls? Consider the case where the program **new** $x := t$ **in** S invokes the procedure S_0. In refining

$$\sigma[x \mapsto \mathrm{eval}(\sigma)(t)] \xrightarrow{\quad S \quad} \tau \ ,$$

one is confronted with justifying $\sigma' \xrightarrow{\quad \mathbf{call} \quad} \tau'$. However, it would not be correct to investigate $\sigma' \xrightarrow{\quad S_0 \quad} \tau'$ at this point, because, due to static scoping, an environmental change is required. The program S_0 must be able to access the *global* value of the program variable x which is no longer present in σ'. In particular, combining blocks and procedures, the operational justification employing the previous implementation of implicit stacks is *not* adequate.

To avoid shadowing, Clarke Jr. (1979) suggests to rename local program variable names [14]. Whenever one encounters a block of the form **new** x **in** S, one instead considers the semantics of **new** x' **in** $S[x \mapsto x']$ where x' is fresh. We prefer a semantic account which avoids syntactic modifications. The key idea is to distinguish between global and local program variables, because, for top-level procedure declarations, the procedure body may only inspect *global* program variables, whereas the semantics for blocks only leads to updating of *local* program variables.

Both the address and the explicit stack mechanism would also introduce enough machinery to avoid this problem. But such an approach is redundant in that one does not actually need to worry about all intermediate local program variable allocations. It suffices to access the most recent local program variable as long as we still have access to all global program variables. We duplicate the state space in order to provide a backup of global program variables whenever an environmental change is required.

➥ p. 182 **Definition 3.32 (State Space)** *Relative to a parametrised declaration* gsort *for global program variables, we index the state space by a declaration* lsort

$$\Sigma(\mathrm{lsort}) \stackrel{\mathrm{def}}{=} \left(\prod x \cdot \mathrm{gsort}(x)\right) \times \left(\prod x \cdot \mathrm{lsort}(x)\right)$$

with both global and local declarations mapping program variables to types. Moreover, for $\sigma : \Sigma(\mathrm{lsort})$, *let* $\sigma.\mathrm{global} : \prod x \cdot \mathrm{gsort}(x)$ *and* $\sigma.\mathrm{local} : \prod x \cdot \mathrm{lsort}(x)$ *denote the global and local components of the state* σ.

[14] Sieber's (1981) version of this definition is flawed because he erroneously updates x in the declared procedure body S_0.

Scoping is controlled by an environment $E : \mathbf{VAR} \to \texttt{bool}$. Intuitively, $E(x)$ holds if x is a *local* program variable in the current scope. For convenience we employ set notation. In particular,

$$\begin{aligned} x \in E &\stackrel{\text{def}}{=} E(x) \\ \emptyset &\stackrel{\text{def}}{=} \lambda x \cdot \mathbf{false} \\ E \cup \{x\} &\stackrel{\text{def}}{=} \lambda y \cdot (x = y) \vee (y \in E) \end{aligned}$$

Looking up values of program variables in scope depends on both the current environment and state:

$$\text{lookup}(E, \sigma)(x) \stackrel{\text{def}}{=} \mathbf{if}\ x \in E\ \mathbf{then}\ \sigma.\text{local}(x)\ \mathbf{else}\ \sigma.\text{global}(x) \qquad \text{for } \sigma : \Sigma(\text{lsort}).$$

Its type is given by

$$\text{AccSort}(\text{lsort}, E)(x) \stackrel{\text{def}}{=} \mathbf{if}\ x \in E\ \mathbf{then}\ \text{lsort}(x)\ \mathbf{else}\ \text{gsort}(x)$$

Substitution has to take the environment into account. We define

$$\sigma[x \mapsto_E t] \stackrel{\text{def}}{=} \mathbf{if}\ x \in E\ \mathbf{then}\ (\sigma.\text{global}, \sigma.\text{local}[x \mapsto t])\ \mathbf{else}\ (\sigma.\text{global}[x \mapsto t], \sigma.\text{local})$$

and introduce

$$\text{uL}(\sigma, x, v) \stackrel{\text{def}}{=} (\sigma.\text{global}, \sigma.\text{local}[x \mapsto v]) \tag{3.33}$$

for updating the local entry of a program variable.

3.3.3 Expressions

We retain Def. 3.21 on page 79

$$\text{expression}(\text{sort} : \mathbf{VAR} \to \texttt{Type})(T : \texttt{Type}) \stackrel{\text{def}}{=} \left(\prod x \cdot \text{sort}(x)\right) \to T$$

where sort corresponds to a particular instance of AccSort. A term can only be evaluated in a compatible setting i.e., for a given environment E and a state $\sigma : \Sigma(\text{lsort})$, an expression t must be of type $\text{expression}\big(\text{AccSort}(\text{lsort}, E)\big)(T)$ for some type T. Evaluation proceeds by plugging in the *accessible* fraction of the state space extracted by the lookup function:

➥ p. 183 **Definition 3.33 (Evaluation of Expressions)** *Let* lsort *be an arbitrary sort and E be an arbitrary environment. Given a state $\sigma : \Sigma(\text{lsort})$ and an expression*

$$t : \text{expression}\big(\text{AccSort}(\text{lsort}, E)\big)(T) \ ,$$

we define evaluation by

$$\text{eval}(E, \sigma)(t) \stackrel{\text{def}}{=} t\big(\text{lookup}(E, \sigma)\big)$$

3.3.4 Assertions

The state space keeps track of all global variables and, for each program variable name, the value of the most recently allocated local variable. We provide unrestricted access to the state space in assertions.

➥ p. 185 **Definition 3.34 (Assertions)** $\text{Assertion}(\text{lsort})(T) \stackrel{\text{def}}{=} \big(T \times \Sigma(\text{lsort})\big) \to \texttt{Prop}$

In particular, we can explicitly refer to the value of local and global variables. To support such a view of assertions, one needs to also change the syntactic representation. It is no longer adequate to merely consider standard first-order logic where a (free) variable corresponds to a value of a program variable as this approach does not distinguish between global and local program variables.

Example 3.35 (Syntax of Assertions) *One could restrict the domain of auxiliary variables to the state space and retain the convention that auxiliary variables are represented by upper-case letter. To disambiguate global from local program variables, one could decorate local program variables e.g., with a dot. Then*

$$[\![A = \dot{c} \wedge a = \dot{A}]\!](Z, \sigma)$$

corresponds to

$$Z.\text{global}(a) = \sigma.\text{local}(c) \wedge \sigma.\text{global}(a) = Z.\text{local}(a) \ .$$

As a side-effect of such a richer notion of assertion, it is conceivable that some of the perceived short-comings of verification calculi can be avoided e.g., Cousot (1990), in an attempt to explain the intuition behind Clarke Jr.'s (1979) incompleteness results, writes

> " *But when considering Algol- or Pascal-like languages, the naming conventions in [assertions] P, Q and [programs] C are totally different. For example, objects deeply buried in the runtime stack cannot be accessed by their name although they can be modified using procedure calls!* "

3.3.5 The Programming Language

➥ p. 186 **Definition 3.36 (Syntax)** *We extend Def. 3.23 on page 80 by a constructor for procedure invocations*

$$\textbf{call} : \texttt{prog}(\text{sort})$$

which can be invoked in an arbitrary environment.

Notice that sort is expected to be a mapping of all program variables *in scope* to types. In particular, in the definitions of the operational semantics in the following section, sort must be a particular instance of $\text{AccSort}(\text{lsort}, E)$ for some $\text{lsort} : \textbf{VAR} \to \texttt{Type}$ and environment E. Alternatively, one could directly parametrise the class of programs by a local sort and an environment.

3.3.6 Structural Operational Semantics

Operational semantics has to also take environments into account to ensure that expressions are well-formed. The environment and the sort of local program variables in the final state is always identical to that in the initial state. They may only change for intermediate states, when encountering blocks or procedure invocations.

➥ p. 189 **Definition 3.37 (Structural Operational Semantics)** *The inductive definition of the operational semantics*

$$(\text{lsort}, E) \mid \sigma \xrightarrow{\quad S \quad} \tau$$

depends now explicitly on a sort of local program variables lsort *and an environment* E*, where* $\sigma, \tau : \Sigma(\text{lsort})$ *and* $S : \texttt{prog}\big(\text{AccSort}(\text{lsort}, E)\big)$*. We have collected the set of all rules in Figure 3.8 on the next page.*

3.3.6.1 Procedures with Static Binding

Procedure invocations are legitimate in any environment

$$\textbf{call} : \texttt{prog}\big(\text{AccSort}(\text{lsort}, E)\big) \ ,$$

but local program variables cease to be accessible i.e., for $S_0 : \texttt{prog}(\text{gsort})$, we have

$$\frac{(\text{lsort}, \emptyset) \mid \sigma \xrightarrow{\quad S_0 \quad} \tau}{(\text{lsort}, E) \mid \sigma \xrightarrow{\quad \textbf{call} \quad} \tau} \ . \tag{3.34}$$

Notice that gsort is (extensionally) equal to $\text{AccSort}(\text{lsort}, \emptyset)$. The proposition

$$(\text{lsort}, \emptyset) \mid \sigma \xrightarrow{\quad S_0 \quad} \tau$$

$$(\mathrm{lsort}, E) \mid \sigma \xrightarrow{\textbf{skip}} \sigma$$

$$(\mathrm{lsort}, E) \mid \sigma \xrightarrow{x := t} \sigma[x \mapsto_E \mathrm{eval}(E,\sigma)(t)]$$
$$\text{where } t : \mathrm{expression}\big(\mathrm{AccSort}(\mathrm{lsort}, E)\big)\big(\mathrm{AccSort}(\mathrm{lsort}, E)(x)\big).$$

$$\frac{(\mathrm{lsort}, E) \mid \sigma \xrightarrow{S_1} \eta \quad (\mathrm{lsort}, E) \mid \eta \xrightarrow{S_2} \tau}{(\mathrm{lsort}, E) \mid \sigma \xrightarrow{S_1;\ S_2} \tau}$$

$$\frac{(\mathrm{lsort}, E) \mid \sigma \xrightarrow{S_1} \tau}{(\mathrm{lsort}, E) \mid \sigma \xrightarrow{\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2} \tau} \quad \text{provided } \mathrm{eval}(E,\sigma)(b) = \textbf{true} \ .$$

$$\frac{(\mathrm{lsort}, E) \mid \sigma \xrightarrow{S_2} \tau}{(\mathrm{lsort}, E) \mid \sigma \xrightarrow{\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2} \tau} \quad \text{provided } \mathrm{eval}(E,\sigma)(b) = \textbf{false} \ .$$

$$(\mathrm{lsort}, E) \mid \sigma \xrightarrow{\textbf{while } b \textbf{ do } S} \sigma \quad \text{provided } \mathrm{eval}(E,\sigma)(b) = \textbf{false} \ .$$

$$\frac{(\mathrm{lsort}, E) \mid \sigma \xrightarrow{S} \eta \quad (\mathrm{lsort}, E) \mid \eta \xrightarrow{\textbf{while } b \textbf{ do } S} \tau}{(\mathrm{lsort}, E) \mid \sigma \xrightarrow{\textbf{while } b \textbf{ do } S} \tau}$$
$$\text{provided } \mathrm{eval}(E,\sigma)(b) = \textbf{true} \ .$$

$$\frac{(\mathrm{lsort}, \emptyset) \mid \sigma \xrightarrow{S_0} \tau}{(\mathrm{lsort}, E) \mid \sigma \xrightarrow{\textbf{call}} \tau}$$

$$\frac{\big(\mathrm{lsort}[x \mapsto T], E \cup \{x\}\big) \mid \mathrm{uL}\big(\sigma, x, \mathrm{eval}(E,\sigma)(t)\big) \xrightarrow{S} \tau}{(\mathrm{lsort}, E) \mid \sigma \xrightarrow{\textbf{new } x := t \textbf{ in } S} \mathrm{uL}\big(\tau, x, \sigma.\mathrm{local}(x)\big)}$$

Figure 3.8: Structural Operational Semantics for Static Binding

is not well-formed in (LEGO's) intensional type theory. Instead, one needs to explicitly introduce type coercions i.e., $\sigma \xrightarrow{\text{coerce}(c)(S_0)} \tau$ for a suitable[15] function coerce, where c is a proof of $\forall x \cdot \text{gsort}(x) = \text{AccSort}(\text{lsort}, \emptyset)(x)$. For convenience, we will omit such type coercions in the sequel.

3.3.6.2 Blocks with Backup for Global Program Variables

We extend rule 3.24 on page 80 with a backup mechanism for blocks

$$\frac{\big(\text{lsort}[x \mapsto T], E \cup \{x\}\big) \mid \text{uL}\big(\sigma, x, \text{eval}(E, \sigma)(t)\big) \xrightarrow{\quad S \quad} \tau}{(\text{lsort}, E) \mid \sigma \xrightarrow{\textbf{new}\ x := t\ \textbf{in}\ S} \text{uL}\big(\tau, x, \sigma.\text{local}(x)\big)} \tag{3.35}$$

where, for an arbitrary environment E, an arbitrary type T and a declaration lsort, we have

$$\begin{aligned} &x : \textbf{VAR} && \sigma : \Sigma(\text{lsort}) \\ &t : \text{expression}\big(\text{AccSort}(\text{lsort}, E)\big)(T) && \tau : \Sigma(\text{lsort}[x \mapsto T]) \\ &S : \texttt{prog}\big(\text{AccSort}(\text{lsort}, E \cup \{x\})[x \mapsto T]\big) \ . \end{aligned}$$

By distinguishing between global and local program variables, when commencing execution of S, the initial state still contains the *global* entry for x which may be required if S contains a procedure call. Moreover, observe that

$$\text{AccSort}(\text{lsort}, E \cup \{x\})[x \mapsto T]$$

is (extensionally) equal to

$$\text{AccSort}(\text{lsort}[x \mapsto T], E \cup \{x\}) \ .$$

3.3.6.3 Recursive Depth

To guarantee termination of recursive calls, similar to Def. 3.9 on page 66, we annotate the operational semantics with a recursive depth n which is an upper bound on the recursive depth required in an execution. We refer to Figure 3.9 on the following page for further details.

➥ p. 194

Lemma 3.38 (Bounded Recursive Depth)

1. *If a program terminates, the recursive depth must be finite i.e., if*

$$(\text{lsort}, E) \mid \sigma \xrightarrow{\quad S \quad} \tau$$

then there exists a bound n such that $(\text{lsort}, E) \mid \sigma \xrightarrow{\quad S \quad}_n \tau$.

[15] This is best defined by induction on the structure of programs and exploits substituting terms in types. See Appendix A.

$$(\mathrm{lsort},E) \mid \sigma \xrightarrow{\textbf{skip}}_n \sigma$$

$$(\mathrm{lsort},E) \mid \sigma \xrightarrow{x:=t}_n \sigma[x \mapsto_E \mathrm{eval}(E,\sigma)(t)]$$
$$\text{where } t : \mathrm{expression}\big(\mathrm{AccSort}(\mathrm{lsort},E)\big)\big(\mathrm{AccSort}(\mathrm{lsort},E)(x)\big). \quad (3.36)$$

$$\frac{(\mathrm{lsort},E) \mid \sigma \xrightarrow{S_1}_n \eta \quad (\mathrm{lsort},E) \mid \eta \xrightarrow{S_2}_n \tau}{(\mathrm{lsort},E) \mid \sigma \xrightarrow{S_1;\ S_2}_n \tau}$$

$$\frac{(\mathrm{lsort},E) \mid \sigma \xrightarrow{S_1}_n \tau}{(\mathrm{lsort},E) \mid \sigma \xrightarrow{\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2}_n \tau} \quad \text{provided } \mathrm{eval}(E,\sigma)(b) = \textbf{true} \ .$$

$$\frac{(\mathrm{lsort},E) \mid \sigma \xrightarrow{S_2}_n \tau}{(\mathrm{lsort},E) \mid \sigma \xrightarrow{\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2}_n \tau} \quad \text{provided } \mathrm{eval}(E,\sigma)(b) = \textbf{false} \ .$$

$$(\mathrm{lsort},E) \mid \sigma \xrightarrow{\textbf{while } b \textbf{ do } S}_n \sigma \quad \text{provided } \mathrm{eval}(E,\sigma)(b) = \textbf{false} \ .$$

$$\frac{(\mathrm{lsort},E) \mid \sigma \xrightarrow{S}_n \eta \quad (\mathrm{lsort},E) \mid \eta \xrightarrow{\textbf{while } b \textbf{ do } S}_n \tau}{(\mathrm{lsort},E) \mid \sigma \xrightarrow{\textbf{while } b \textbf{ do } S}_n \tau}$$
$$\text{provided } \mathrm{eval}(E,\sigma)(b) = \textbf{true} \ .$$

$$\frac{(\mathrm{lsort},\emptyset) \mid \sigma \xrightarrow{S_0}_n \tau}{(\mathrm{lsort},E) \mid \sigma \xrightarrow{\textbf{call}}_{n+1} \tau}$$

$$\frac{\big(\mathrm{lsort}[x \mapsto T], E \cup \{x\}\big) \mid \mathrm{uL}\big(\sigma, x, \mathrm{eval}(E,\sigma)(t)\big) \xrightarrow{S}_n \tau}{(\mathrm{lsort},E) \mid \sigma \xrightarrow{\textbf{new } x:=t \textbf{ in } S}_n \mathrm{uL}\big(\tau, x, \sigma.\mathrm{local}(x)\big)} \quad (3.37)$$

Figure 3.9: Recursive Depth for Procedures and Blocks

2. *Bounded recursive depth is a mere annotation of operational semantics i.e., for an arbitrary depth n, $(\mathrm{lsort}, E) \mid \sigma \xrightarrow{S}_n \tau$ implies*

$$(\mathrm{lsort}, E) \mid \sigma \xrightarrow{S} \tau \ .$$

Proof Similar to the proof of Lemma 3.11 on page 67 in the setting without local program variables. □

3.3.6.4 Adequacy

The rule for blocks (3.35) may be comprehensible, but the interaction with procedures is certainly somewhat subtle. Since we *axiomatise* operational semantics, we cannot argue about its soundness and completeness with respect to a more easily comprehensible semantic notion. Rather than taking its adequacy on good faith, one should investigate a number of properties for clarification. One could for example establish that the operational semantics is conservative over the presentations in the previous two sections where we have dealt with procedures and blocks in isolation. Furthermore, one ought to examine how local program variables interact with procedure calls.

For the soundness and completeness proofs, we have found it useful to establish the following properties.

➥ p. 195 **Lemma 3.39 (Determinism)** *The operational semantics of Figure 3.8 and its annotated version of Figure 3.9 is deterministic.*

Proof by induction on the derivation. □

➥ p. 196 **Lemma 3.40 (Procedure Activation and Environments)** *Let E and E' be arbitrary environments.*

$$\frac{(\mathrm{lsort}, E) \mid \sigma \xrightarrow{\mathbf{call}}_n \tau}{(\mathrm{lsort}, E') \mid \sigma \xrightarrow{\mathbf{call}}_n \tau}$$

Proof Inverting the derivation of $(\mathrm{lsort}, E) \mid \sigma \xrightarrow{\mathbf{call}}_n \tau$ yields

$$(\mathrm{lsort}, \emptyset) \mid \sigma \xrightarrow{S_0}_{n-1} \tau \ .$$

Thus, applying rule (3.34) leads to the desired $(\mathrm{lsort}, E') \mid \sigma \xrightarrow{\mathbf{call}}_n \tau$. □

The soundness and completeness proofs are more challenging when combining recursive procedure calls with local program variables because one has to relate the behaviour of procedure activations commencing in states which differ in the value of local program variables. However, if these local program variables are not in scope, their value is preserved.

➥ p. 196 **Theorem 3.41 (Preserving Local Program Variables)** *Let U be any type, y any program variable which is not local i.e., $y \notin E$ and $v : U$. Whenever*

$$(\mathrm{lsort}, E) \mid \sigma \xrightarrow{S}_n \tau$$

then

$$(\mathrm{lsort}[y \mapsto U], E) \mid \mathrm{uL}(\sigma, y, v) \xrightarrow{S}_n \mathrm{uL}(\tau, y, v)$$

holds.

Notice that the type $\mathrm{AccSort}(\mathrm{lsort}, E)$ is (extensionally) equal to

$$\mathrm{AccSort}(\mathrm{lsort}[y \mapsto U], E) \qquad \text{for } y \notin E.$$

Due to Lemma 3.38, such results also hold when one omits the recursive depth annotations. But notice that, with such annotations, the above theorem also establishes that the recursive depth is *preserved*.

Proof by induction on the derivation of $(\mathrm{lsort}, E) \mid \sigma \xrightarrow{S}_n \tau$. The cases where S amounts to an assignment or a block require some attention, because one needs to show that updating is commutative in these circumstances.

Assignment By the axiom for assignments (3.36), it suffices to establish that

$$\mathrm{uL}\big(\sigma[x \mapsto_E \mathrm{eval}(E, \sigma)(t)], y, v\big)$$

is (extensionally) equal to

$$\mathrm{uL}(\sigma, y, v)[x \mapsto_E \mathrm{eval}(E, \mathrm{uL}(\sigma, y, v))(t)] \quad .$$

Since y is not in scope, evaluation of the term t in the environment E cannot depend on the local entry of y in the state σ. Thus, $\mathrm{eval}(E, \mathrm{uL}(\sigma, y, v))(t)$ yields the same result as $\mathrm{eval}(E, \sigma)(t)$. We proceed by case analysis on $x \in E$.

- $x \in E$ The program variables x and y must be different. Hence, updating of x and y is interchangeable.
- $x \notin E$ The order of updating is irrelevant because one updates the *local* entry for y and the *global* entry for x.

Block We abbreviate

$$\begin{aligned} \mathrm{lsort}_x &\stackrel{\text{def}}{=} \mathrm{lsort}[x \mapsto T] \\ \mathrm{lsort}_{x,y} &\stackrel{\text{def}}{=} \mathrm{lsort}[x \mapsto T][y \mapsto U] \\ \mathrm{lsort}_y &\stackrel{\text{def}}{=} \mathrm{lsort}[y \mapsto U] \quad . \end{aligned}$$

From the assumption

$$(\mathrm{lsort}_x, E \cup \{x\}) \mid \mathrm{uL}\big(\sigma, x, \mathrm{eval}(E,\sigma)(t)\big) \xrightarrow{\quad S \quad}_n \tau \tag{3.38}$$

and the induction hypothesis

$$\begin{aligned} &y \notin E \cup \{x\} \Rightarrow \\ &(\mathrm{lsort}_{x,y}, E \cup \{x\}) \mid \mathrm{uL}\Big(\mathrm{uL}\big(x, \mathrm{eval}(E,\sigma)(t), \sigma\big), y, v\Big) \xrightarrow{\quad S \quad}_n \mathrm{uL}(\tau, y, v) \ , \end{aligned}$$

one needs to show

$$(\mathrm{lsort}_y, E) \mid \mathrm{uL}(\sigma, y, v) \xrightarrow{\textbf{new}\ x := t\ \textbf{in}\ S}_n \mathrm{uL}\Big(\mathrm{uL}\big(\tau, x, \sigma.\,\mathrm{local}(x)\big), y, v\Big) \ . \tag{3.39}$$

The induction hypothesis is only applicable if x and y are different program variables. If they are identical, we need to appeal to (3.38).

$x = y$ Since $y \notin E$, evaluation of the term t relative to (E, σ) yields the same result as evaluation with respect to $\big(E, \mathrm{uL}(\sigma, y, v)\big)$. Hence, the initial state of (3.38) is (extensionally) equal to

$$\mathrm{uL}\big(\mathrm{uL}(\sigma, y, v), x, \mathrm{eval}(E, \mathrm{uL}(\sigma, y, v))(t)\big) \ .$$

By the rule for blocks (3.37), we may conclude

$$(\mathrm{lsort}, E) \mid \mathrm{uL}(\sigma, y, v) \xrightarrow{\textbf{new}\ x := t\ \textbf{in}\ S}_n \mathrm{uL}(\tau, x, v) \ . \tag{3.40}$$

This concludes the case $x = y$ because the final states of (3.40) and (3.39) are (extensionally) equal.

$x \neq y$ Since $y \notin E$, we also have $y \notin E \cup \{x\}$ and may exploit the induction hypothesis. For distinct program variables x and y, we may commute the updating operations in the initial state. Thus, the rule for blocks (3.37) leads to

$$(\mathrm{lsort}_y, E) \mid \mathrm{uL}(\sigma, y, v) \xrightarrow{\textbf{new}\ x := t\ \textbf{in}\ S}_n \mathrm{uL}\big(\mathrm{uL}(\tau, y, v), x, \sigma.\,\mathrm{local}(x)\big)$$

Finally, we commute the updating operations in the final state.

□

➥ p. 197 **Corollary 3.42 (Preserving Local Program Variables)**

1. $\big((\mathrm{lsort}, E) \mid \sigma \xrightarrow{S} \tau \wedge y \notin E\big) \Rightarrow \sigma.\mathrm{local}(y) = \tau.\mathrm{local}(y)$
2. *Let U be any type and $v : U$ be an arbitrary value. Given any environments E and E', whenever $(\mathrm{lsort}, E) \mid \sigma \xrightarrow{\textbf{call}}_n \tau$ holds, one may also derive $(\mathrm{lsort}[y \mapsto U], E') \mid \mathrm{uL}(\sigma, y, v) \xrightarrow{\textbf{call}}_n \mathrm{uL}(\tau, y, v)$.*

Proof

1. We apply Theorem 3.41 and instantiate v with $\sigma.\mathrm{local}(y)$. Thus, by determinism, $\tau = \mathrm{uL}\big(\tau, y, \sigma.\mathrm{local}(y)\big)$ from which one may deduce that $\tau.\mathrm{local}(y)$ and $\sigma.\mathrm{local}(y)$ are identical.
2. By Lemma 3.40, we may reduce the problem to a setting without any local program variables. Given $(\mathrm{lsort}, \emptyset) \mid \sigma \xrightarrow{\textbf{call}}_n \tau$, Theorem 3.41 yields

$$(\mathrm{lsort}[y \mapsto U], \emptyset) \mid \mathrm{uL}(\sigma, y, v) \xrightarrow{\textbf{call}}_n \mathrm{uL}(\tau, y, v) \ .$$

□

3.3.7 Hoare Logic

We need to revise Hoare Logic to integrate environments. One could either introduce them at the level of atomic correctness formulae

$$\big((\mathrm{lsort}_1, E_1) \mid \{p_1\}\ \textbf{call}\ \{q_1\}\big) \vdash_{\mathrm{Hoare}} \big((\mathrm{lsort}_2, E_2) \mid \{p_2\}\ S\ \{q_2\}\big)$$

or across a sequent

$$(\mathrm{lsort}, E) \mid \big(\{p_1\}\ \textbf{call}\ \{q_1\} \vdash_{\mathrm{Hoare}} \{p_2\}\ S\ \{q_2\}\big)$$

It turned out that it is easier to design a *complete* verification calculus for sequents with a single environment. Intuitively, it obviates the requirement for a structural rule to mediate between different environments with *different* sorts of local program variables. However, since the rule for blocks needs to modify the environment in the premiss, both the context and the succedent of the premiss needs to be adjusted. Alternatively, one could consider sequents in which the local sort is shared, whereas the set of local program variable names in scope is attached to atomic correctness formulae.

➥ p. 199 **Definition 3.43 (Semantics of Hoare Logic)**

Similar to Def. 3.5 on page 62, we define $(\mathrm{lsort}, E) \mid \Gamma \models_{\mathrm{Hoare}} \{p\}\ S\ \{q\}$ by case analysis on the structure of the context Γ.

- *In the empty context, we have*

$$(\mathrm{lsort}, E) \mid \emptyset \models_{\mathrm{Hoare}} \{p\}\ S\ \{q\} \stackrel{\mathrm{def}}{=}$$
$$\forall Z \cdot \forall \sigma \cdot p(Z,\sigma) \Rightarrow \exists \tau \cdot \big((\mathrm{lsort}, E) \mid \sigma \xrightarrow{\quad S \quad} \tau\big) \wedge q(Z,\tau)$$

- *For* $\Gamma = \{p'\}$ **call** $\{q'\}$*, we define validity relative to the empty context i.e.,*

$$(\mathrm{lsort}, E) \mid \Gamma \models_{\mathrm{Hoare}} \{p\}\ S\ \{q\} \stackrel{\mathrm{def}}{=}$$
$$\big((\mathrm{lsort}, E) \mid \emptyset \models_{\mathrm{Hoare}} \{p'\}\ \mathbf{call}\ \{q'\}\big) \Rightarrow$$
$$\big((\mathrm{lsort}, E) \mid \emptyset \models_{\mathrm{Hoare}} \{p\}\ S\ \{q\}\big)$$

3.3.7.1 Procedure Invocation

Turning to procedure calls, in resolving a correctness formula which invokes a procedure in an environment (lsort, E), we may assume the correctness of this formulae while proving the correctness of the unfolded procedure body. Since procedure calls cause context switching, we investigate the triggered proof obligation relative to the empty environment. Assertions may inspect all entries of the state space irrespective of scoping restrictions. We may therefore reuse the same assertions as in the rule (3.8) for recursive procedures invocations without local program variables on page 62.

$$\frac{\forall t : W \cdot (\mathrm{lsort}, \emptyset) \mid \{\hat{p}(t)\}\ \mathbf{call}\ \{q\} \vdash_{\mathrm{Hoare}} \{p(t)\}\ S_0\ \{q\}}{(\mathrm{lsort}, E) \mid \emptyset \vdash_{\mathrm{Hoare}} \{\lambda(Z,\sigma) \cdot \exists t : W \cdot p(t)(Z,\sigma)\}\ \mathbf{call}\ \{q\}}$$
where $\hat{p}(t)(Z,\sigma) \stackrel{\mathrm{def}}{=} \exists u : W \cdot p(u)(Z,\sigma) \wedge u < t$ and $(W, <)$ is well-founded. (3.41)

3.3.7.2 Blocks

If one were to insist on assertions being simply first-order logical formulae where some free variables are interpreted as the value of program variables, one would effectively limit scoping in assertions to dynamic binding. Specifically, the rule (3.25) from page 81 would be unsound with respect to static binding (Apt 1981). One can avoid scoping intrusion by substituting program variables in *programs* instead of assertions. For uninitialised variables, Gorelick (1975) proposes

$$\frac{\{p \wedge y = \omega\}\ S[x \mapsto y]\ \{q\}}{\{p\}\ \mathbf{new}\ x\ \mathbf{in}\ S\ \{q\}} \qquad \text{where } y \notin \mathit{free}(p, S, q). \tag{3.42}$$

However, modifying the program text during the process of verification may not be appreciated, because software engineers may have chosen variable names carefully to reflect their intended usage. Furthermore, Apt (1981) admits that rule (3.25) is easier to use and handle than rule (3.42).

We follow Ah-kee's (1990) technique of annotating correctness formulae with environments and allow assertions to inspect the complete state space and not only the value of variables in scope. This has previously only been applied to VDM. In assertions we may thus explicitly refer to the value of a local and a global program variable[16]. We propose the rule

$$\frac{\forall v \cdot \big(\mathrm{lsort}[x \mapsto T], E \cup \{x\}\big) \mid \mathrm{Pr}(\Gamma, x, v) \vdash_{\mathrm{Hoare}} \quad \begin{matrix}\{\mathrm{In}(p,x,v,t,E)\} \\ S \\ \{\mathrm{uL}(q,x,v)\}\end{matrix}}{(\mathrm{lsort}, E) \mid \Gamma \vdash_{\mathrm{Hoare}} \{p\}\ \mathbf{new}\ x := t\ \mathbf{in}\ S\ \{q\}}$$

$$\text{with} \quad \mathrm{Pr}(\Gamma, x, v) \stackrel{\mathrm{def}}{=} \begin{cases} \emptyset & \text{for } \Gamma = \emptyset, \\ \{\mathrm{Pr}(p', x, v)\}\ \mathbf{call}\ \{\mathrm{Pr}(q', x, v)\} & \text{for } \Gamma = \{p'\}\ \mathbf{call}\ \{q'\} \end{cases}$$

$$\mathrm{Pr}(p, x, v)\big((Z, V), \sigma\big) \stackrel{\mathrm{def}}{=} \mathrm{uL}(p, x, v)(Z, \sigma) \wedge \sigma.\mathrm{local}(x) = V \tag{3.43}$$

$$\mathrm{In}(p, x, v, t, E)(Z, \sigma) \stackrel{\mathrm{def}}{=} \mathrm{uL}(p, x, v)(Z, \sigma) \wedge \sigma.\mathrm{local}(x) = \mathrm{eval}(E, \mathrm{uL}(\sigma, x, v))(t)$$

In the premiss of the rule, we reduce the complexity by considering the derivation of a correctness formula concerning the program S in the updated environment. We have to freeze all references to the local variable x in all assertions of the premiss. Let $v : \mathrm{lsort}(x)$ be a (fresh) constant[17]. We overload notation and exploit our previous definition (3.33) for updating the local entry in the state space. Employing

$$\mathrm{uL}(p, x, v)(Z, \sigma) \stackrel{\mathrm{def}}{=} p\big(Z, \mathrm{uL}(\sigma, x, v)\big)$$

we replace all references to the previous local value of x in all assertions of the premiss by the same v. Since v is fresh, it corresponds to the previous value of the local program variable x when reaching the block. In particular, in the initial state of the succedent in the premiss, for $\sigma : \Sigma(\mathrm{lsort}[x \mapsto T])$, the state $\mathrm{uL}(\sigma, x, v)$ amounts to the initial state of the succedent in the conclusion. Thus, we may add $\sigma.\mathrm{local}(x) = \mathrm{eval}(E, \mathrm{uL}(\sigma, x, v))(t)$ in the precondition for S to ensure that the local value of x is equal to the value of the expression t when evaluated in the original environment E and state σ.

However, due to static scoping, the value of the local program variable x must be preserved across a procedure call in the context $\mathrm{Pr}(\Gamma, x, v)$. The auxiliary variable $V : T$ records the value of the current local program variable x at the time of invoking a procedure. In the context of the rule's premiss, we have extended the domain of

[16] A less powerful extension to the notion of assertions to merely add propositions of the form $x \in E$ to check if x is a local program variable (Trakhtenbrot et al. 1984) does not seem to suffice.

[17] In the completeness proof, we will see that, due to our choice of permitting (local) program variables with *arbitrary* sorts, it is useful to share the same reference v across the sequent. Thus, v cannot be an auxiliary variable, because they range only over atomic correctness formulae. For a sound and complete calculus where v is represented as an auxiliary variable, one would need to restrict the sort of all (local) program variables to types $SORT \subsetneq \texttt{Type}$ in which, for every type in $SORT$, one can effectively extract an element of that type.

$$(\mathrm{lsort},E) \mid \{p\}\ \mathbf{call}\ \{q\} \vdash_{\mathrm{Hoare}} \{p\}\ \mathbf{call}\ \{q\}$$

$$(\mathrm{lsort},E) \mid \Gamma \vdash_{\mathrm{Hoare}} \{p\}\ \mathbf{skip}\ \{p\}$$

$$(\mathrm{lsort},E) \mid \Gamma \vdash_{\mathrm{Hoare}} \left\{\lambda(Z,\sigma)\cdot p\big(Z,\sigma[x\mapsto_E \mathrm{eval}(E,\sigma)(t)]\big)\right\}\ x:=t\ \{p\}$$

$$\frac{(\mathrm{lsort},E) \mid \Gamma \vdash_{\mathrm{Hoare}} \{p\}\ S_1\ \{r\} \qquad (\mathrm{lsort},E) \mid \Gamma \vdash_{\mathrm{Hoare}} \{r\}\ S_2\ \{q\}}{(\mathrm{lsort},E) \mid \Gamma \vdash_{\mathrm{Hoare}} \{p\}\ S_1;\ S_2\ \{q\}}$$

$$\frac{\begin{array}{c}(\mathrm{lsort},E) \mid \Gamma \vdash_{\mathrm{Hoare}} \{\lambda(Z,\sigma)\cdot p(Z,\sigma)\wedge \mathrm{eval}(\sigma)(b)=\mathbf{true}\}\ S_1\ \{q\}\\ (\mathrm{lsort},E) \mid \Gamma \vdash_{\mathrm{Hoare}} \{\lambda(Z,\sigma)\cdot p(Z,\sigma)\wedge \mathrm{eval}(\sigma)(b)=\mathbf{false}\}\ S_2\ \{q\}\end{array}}{(\mathrm{lsort},E) \mid \Gamma \vdash_{\mathrm{Hoare}} \{p\}\ \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \{q\}}$$

$$\frac{\forall t : W\cdot (\mathrm{lsort},E) \mid \Gamma \vdash_{\mathrm{Hoare}} \{\hat{p}\}\ S\ \{\lambda(Z,\tau)\cdot p(Z,\tau)\wedge u(\tau)<t\}}{(\mathrm{lsort},E) \mid \Gamma \vdash_{\mathrm{Hoare}} \{p\}\ \mathbf{while}\ b\ \mathbf{do}\ S\ \{\lambda(Z,\tau)\cdot p(Z,\tau)\wedge \mathrm{eval}(\tau)(b)=\mathbf{false}\}}$$
$$\text{where } \hat{p}(Z,\sigma) \stackrel{\mathrm{def}}{=} p(Z,\sigma)\wedge \mathrm{eval}(\sigma)(b)=\mathbf{true}\wedge u(\sigma)=t$$
$$\text{and } (W,<) \text{ is well-founded.} \quad (3.44)$$

$$\frac{\forall t : W\cdot (\mathrm{lsort},\emptyset) \mid \{\hat{p}(t)\}\ \mathbf{call}\ \{q\} \vdash_{\mathrm{Hoare}} \{p(t)\}\ S_0\ \{q\}}{(\mathrm{lsort},E) \mid \emptyset \vdash_{\mathrm{Hoare}} \{\lambda(Z,\sigma)\cdot \exists t : W\cdot p(t)(Z,\sigma)\}\ \mathbf{call}\ \{q\}}$$
$$\text{where } \hat{p}(t)(Z,\sigma) \stackrel{\mathrm{def}}{=} \exists u : W\cdot p(u)(Z,\sigma)\wedge u<t \text{ and } (W,<) \text{ is well-founded.}$$

$$\frac{\forall v\cdot \big(\mathrm{lsort}[x\mapsto T],E\cup\{x\}\big) \mid \mathrm{Pr}(\Gamma,x,v) \vdash_{\mathrm{Hoare}} \begin{array}{c}\{\mathrm{In}(p,x,v,t,E)\}\\ S\\ \{\mathrm{uL}(q,x,v)\}\end{array}}{(\mathrm{lsort},E) \mid \Gamma \vdash_{\mathrm{Hoare}} \{p\}\ \mathbf{new}\ x:=t\ \mathbf{in}\ S\ \{q\}}$$
$$\text{with} \quad \mathrm{Pr}(\Gamma,x,v) \stackrel{\mathrm{def}}{=} \begin{cases}\emptyset & \text{for } \Gamma=\emptyset,\\ \{\mathrm{Pr}(p',x,v)\}\ \mathbf{call}\ \{\mathrm{Pr}(q',x,v)\} & \text{for } \Gamma=\{p'\}\ \mathbf{call}\ \{q'\}\end{cases}$$
$$\mathrm{Pr}(p,x,v)\big((Z,V),\sigma\big) \stackrel{\mathrm{def}}{=} \mathrm{uL}(p,x,v)(Z,\sigma)\wedge \sigma.\,\mathrm{local}(x)=V$$
$$\mathrm{In}(p,x,v,t,E)(Z,\sigma) \stackrel{\mathrm{def}}{=} \mathrm{uL}(p,x,v)(Z,\sigma)\wedge \sigma.\,\mathrm{local}(x)=\mathrm{eval}(E,\mathrm{uL}(\sigma,x,v))(t)$$

$$\frac{(\mathrm{lsort},E) \mid \Gamma \vdash_{\mathrm{Hoare}} \{p_1\}\ S\ \{q_1\}}{(\mathrm{lsort},E) \mid \Gamma \vdash_{\mathrm{Hoare}} \{p\}\ S\ \{q\}}$$
where $p_1,q_1 : \mathrm{Assertion}(U)$ and $p,q : \mathrm{Assertion}(T)$ for arbitrary types T and U
$$\text{and } \forall Z\cdot\forall\sigma\cdot p(Z,\sigma)\Rightarrow\Big(\exists Z_1\cdot p_1(Z_1,\sigma)\wedge\big(\forall\tau\cdot q_1(Z_1,\tau)\Rightarrow q(Z,\tau)\big)\Big). \quad (3.45)$$

Figure 3.10: Hoare Logic and Static Binding – Total Correctness

auxiliary variables to generate a fresh constant $V : T$. Therefore, in general, one may no longer a priori fix the domain of auxiliary variables as the state space.

An alternative solution would be to restrict the class of assertions for procedure specifications. As Tarlecki (1985) has pointed out, one usually wants to use a procedure in several different contexts. Therefore, auxiliary variables, which play the rôle of a local reference point, ought to coincide with the value of the variables in the initial state. In other words, one may consider restricting the specification for a procedure activation to the VDM format where the precondition $p'(Z,\sigma)$ is of the form $\sigma = Z \wedge p_{\mathrm{VDM}}(\sigma)$ for some predicate $p_{\mathrm{VDM}} \subseteq \Sigma(\mathrm{lsort})$ and the precondition q' is of the form $\mathrm{Assertion}(\mathrm{lsort})\big(\Sigma(\mathrm{lsort})\big)$. Then, for $Z : \Sigma(\mathrm{lsort}[x \mapsto T])$, one may refer to $Z.\mathrm{local}(x)$ instead of V. Hence, the context in the premiss could be simplified to

$$\mathrm{Pr}(\Gamma,x,v) = \begin{cases} \emptyset & \text{for } \Gamma = \emptyset, \\ \{\mathrm{uL}(p',x,v)\}\ \mathbf{call}\ \{\mathrm{Pr}(q',x,v)\} & \text{for } \Gamma = \{p'\}\ \mathbf{call}\ \{q'\} \end{cases}$$
$$\text{where}\ \mathrm{Pr}(q',x,v)(Z,\tau) \stackrel{\mathrm{def}}{=} q\big(\mathrm{uL}(Z,x,v),\mathrm{uL}(\tau,x,v)\big) \wedge \tau.\mathrm{local}(x) = Z.\mathrm{local}(x)$$

In a setting where the scope of local program variables is provided relative to atomic correctness formulae, one may leave the context (and its environment) invariant in the rule for blocks. When looking up correctness formulae in the context, the difference in environments may then be reconciled in a single step e.g.,

$$\big((\mathrm{lsort},\emptyset) \mid \{p\}\ \mathbf{call}\ \{q\}\big) \vdash_{\mathrm{Hoare}} \left(\left(\mathrm{lsort}\left[\vec{x} \mapsto \vec{T}\right], \vec{x}\right) \mid \{\mathrm{Pr}(p)\}\ S\ \{\mathrm{Pr}(q)\}\right)$$
$$\text{where}\ \mathrm{Pr}(p)\big((Z,\vec{v},\vec{V}),\sigma\big) \stackrel{\mathrm{def}}{=} \mathrm{uL}(p,\vec{x},\vec{v})(Z,\sigma) \wedge \bigwedge_{i\in I} \sigma.\mathrm{local}(x_i) = V_i.$$

3.3.7.3 Soundness

➥ p. 204

Theorem 3.44 (Soundness) *The verification calculus defined as the least relation satisfying the rules summarised in Figure 3.10 on the previous page is sound.*

Proof by induction on the derivation of $(\mathrm{lsort},E) \mid \Gamma \vdash_{\mathrm{Hoare}} \{p\}\ S\ \{q\}$. Consider the case where we have applied the rule for blocks as the last step of the derivation. We abbreviate $\mathrm{lsort}_x \stackrel{\mathrm{def}}{=} \mathrm{lsort}[x \mapsto T]$. From the induction hypothesis

$$\forall v : \mathrm{lsort}(x) \cdot$$
$$(\mathrm{lsort}_x, E \cup \{x\}) \mid \mathrm{Pr}(\Gamma,x,v) \models_{\mathrm{Hoare}} \{\mathrm{In}\,(p,x,v,t,E)\}\ S\ \{\mathrm{uL}\,(q,x,v)\} \quad (3.46)$$

we have to establish

$$(\mathrm{lsort},E) \mid \Gamma \models_{\mathrm{Hoare}} \{p\}\ \mathbf{new}\ x := t\ \mathbf{in}\ S\ \{q\} \quad .$$

Let Z be an arbitrary auxiliary variable and $\sigma : \Sigma(\mathrm{lsort})$ be an arbitrary initial state such that $p(Z,\sigma)$ holds. In the case where the context Γ contains a correctness formula $\{p'\}$ **call** $\{q'\}$, the induction hypothesis is no longer directly applicable[18]. As a new aspect in the proof, given

$$(\mathrm{lsort}, E) \mid \emptyset \models_{\mathrm{Hoare}} \left\{p'\right\} \textbf{ call } \left\{q'\right\} \quad , \tag{3.47}$$

one needs to establish

$$(\mathrm{lsort}_x, E \cup \{x\}) \mid \emptyset \vdash_{\mathrm{Hoare}} \left\{\mathrm{Pr}(p', x, v)\right\} \textbf{ call } \left\{\mathrm{Pr}(q', x'v)\right\} \quad . \tag{3.48}$$

Let Z', V be arbitrary auxiliary variables and $\sigma' : \Sigma(\mathrm{lsort}_x)$ be an arbitrary initial state such that $p'(Z', \mathrm{uL}(\sigma', x, v))$ and $\sigma'.\mathrm{local}(x) = V$ hold. We need to find a final state $\tau' : \Sigma(\mathrm{lsort}_x)$ such that one has

$$(\mathrm{lsort}_x, E \cup \{x\}) \mid \sigma' \xrightarrow{\textbf{call}} \tau' \tag{3.49}$$

$$q'\left(Z', \mathrm{uL}(\tau', x, v)\right) \tag{3.50}$$

$$\tau'.\mathrm{local}(x) = V \tag{3.51}$$

Appealing to (3.47) we may extract a final state τ which satisfies

$$(\mathrm{lsort}, E) \mid \mathrm{uL}(\sigma', x, v) \xrightarrow{\textbf{call}} \tau \tag{3.52}$$

and $q'(Z', \tau)$. By Cor. 3.42.2, the derivation (3.52) can be lifted to the updated environment (3.49) with $\tau' \stackrel{\mathrm{def}}{=} \mathrm{uL}\left(x, \sigma'.\mathrm{local}(x), \tau\right)$, because σ' is (extensionally) equal to $\mathrm{uL}\left(x, \sigma'.\mathrm{local}(x), \mathrm{uL}(\sigma', x, v)\right)$. Cor. 3.42.1 guarantees that in the derivation of (3.52), $\tau = \mathrm{uL}(\tau, x, v)$. Thus, $\mathrm{uL}(\tau', x, v) = \tau$ and we may conclude (3.50). Finally, in the light of $\sigma'.\mathrm{local}(x) = V$, applying Cor. 3.42.1 to the derivation (3.49) ensures the desired (3.51). □

3.3.7.4 Completeness

The proof of completeness follows the structure in the setting with recursive procedures without local program variables in Section 3.1.6 on page 65. We abbreviate

$$\begin{aligned}
p_0(S)(\mathrm{lsort}, E)(n)(Z,\sigma) &\stackrel{\mathrm{def}}{=} (\mathrm{lsort}, E) \mid \sigma \xrightarrow{S}_n Z \\
\psi(\mathrm{lsort}, E)(n)(Z,\sigma) &\stackrel{\mathrm{def}}{=} p_0(\textbf{call})(n+1)(\mathrm{lsort}, E)(Z,\sigma) \\
\hat{\psi}(\mathrm{lsort}, E)(n)(Z,\sigma) &\stackrel{\mathrm{def}}{=} \exists u : \mathtt{nat} \cdot \psi(u)(\mathrm{lsort}, E)(Z,\sigma) \wedge u < n \\
q(Z,\tau) &\stackrel{\mathrm{def}}{=} Z = \tau \quad .
\end{aligned}$$

[18] for $v = \sigma.\mathrm{local}(x)$

The subtle interaction between procedures and local variables is dealt with in the following theorem. It established the pseudo induction hypothesis of the MGF theorem in the case of a recursive procedure call.

➥ p. 204 **Theorem 3.45 (Completeness Step)**

$$(\mathrm{lsort}, \emptyset) \mid \{\hat{\psi}(\mathrm{lsort}, \emptyset)(n)\}\ \mathbf{call}\ \{q\} \vdash_{\mathrm{Hoare}} \{\psi(\mathrm{lsort}, \emptyset)(n)\}\ S_0\ \{q\} \quad .$$

Proof Just like the proof of the corresponding Lemma 3.12 on page 67, appealing to the consequence rule and the definition of bounded recursive depth, we equivalently transform the proof obligation into

$$(\mathrm{lsort}, \emptyset) \mid \{\hat{\psi}(\mathrm{lsort}, \emptyset)(n)\}\ \mathbf{call}\ \{q\} \vdash_{\mathrm{Hoare}} \{p_0(S_0)(\mathrm{lsort}, \emptyset)(n)\}\ S_0\ \{q\} \quad . \tag{3.53}$$

However, a direct proof by induction on the structure of the procedure body S_0 would not go through, because

- the context would be unaffected and
- programs would not be able to refer to local program variables.

Thus, the new rule for blocks (3.43) would not be applicable. Instead, we have to consider arbitrary environments E. Moreover, we strengthen the proof obligation and replace $\hat{\psi}(\mathrm{lsort}, E)(n)$ and q in the context by arbitrary assertions p_1, q_1 such that, according to the rule of consequence,

$$\frac{(\mathrm{lsort}, E) \mid \emptyset \vdash_{\mathrm{Hoare}} \{p_1\}\ \mathbf{call}\ \{q_1\}}{(\mathrm{lsort}, E) \mid \emptyset \vdash_{\mathrm{Hoare}} \{\hat{\psi}(\mathrm{lsort}, E)(n)\}\ \mathbf{call}\ \{q\}} \quad .$$

Hence, we pursue

$$\forall p_1, q_1 \cdot \mathrm{KSC}\big(\hat{\psi}(\mathrm{lsort}, E)(n), q, p_1, q_1\big) \Rightarrow (\mathrm{lsort}, E) \mid \{p_1\}\ \mathbf{call}\ \{q_1\} \vdash_{\mathrm{Hoare}} \{p_0(S)(\mathrm{lsort}, E)(n)\}\ S\ \{q\}$$

where

$$\mathrm{KSC}(p, q, p_1, q_1) \stackrel{\mathrm{def}}{=} \forall Z \cdot \forall \sigma \cdot p(Z, \sigma) \Rightarrow \Big(\exists Z_1 \cdot p_1(Z_1, \sigma) \wedge \big(\forall \tau \cdot q_1(Z_1, \tau) \Rightarrow q(Z, \tau)\big)\Big)$$

by induction on the structure of S. In comparison to the corresponding proof without blocks, one needs to additionally appeal to the rule of consequence in each case to deal with the arbitrary assertions p_1, q_1. As an example, we spell out the case of a procedure invocation. The challenging part is the new case when we encounter a block.

Procedure Call Let p_1, q_1 be arbitrary assertions. Given the assumption

$$\mathrm{KSC}\big(\hat{\psi}(\mathrm{lsort}, E)(n), q, p_1, q_1\big) \quad ,$$

we need to show

$$(\mathrm{lsort}, E) \mid \{p_1\}\ \mathbf{call}\ \{q_1\} \vdash_{\mathrm{Hoare}} \{p_0(\mathbf{call})(\mathrm{lsort}, E)(n)\}\ \mathbf{call}\ \{q\} \quad . \tag{3.54}$$

From being able to access the context, we may derive

$$(\mathrm{lsort}, E) \mid \{p_1\}\ \mathbf{call}\ \{q_1\} \vdash_{\mathrm{Hoare}} \{p_1\}\ \mathbf{call}\ \{q_1\} \quad .$$

An appeal to the rule of consequence yields

$$(\mathrm{lsort}, E) \mid \{p_1\}\ \mathbf{call}\ \{q_1\} \vdash_{\mathrm{Hoare}} \{\hat{\psi}(\mathrm{lsort}, E)(n)\}\ \mathbf{call}\ \{q\} \quad .$$

Applying the consequence rule once more leads to (3.54). For a justification of the side-condition, see the corresponding proof in the scenario without blocks on page 67.

Blocks We abbreviate $\mathrm{lsort}_x \stackrel{\mathrm{def}}{=} \mathrm{lsort}[x \mapsto T]$. From the induction hypothesis

$$\begin{aligned} &\forall p_1, q_1 \cdot \mathrm{KSC}\big(\hat{\psi}(\mathrm{lsort}, E)(n), q, p_1, q_1\big) \Rightarrow \\ &\qquad (\mathrm{lsort}_x, E \cup \{x\}) \mid \{p_1\}\ \mathbf{call}\ \{q_1\} \vdash_{\mathrm{Hoare}} \{p_0(S)(\mathrm{lsort}, E)(n)\}\ S\ \{q\} \end{aligned}$$

and arbitrary assertions p_1, q_1 satisfying

$$\mathrm{KSC}(\hat{\psi}(\mathrm{lsort}, E)(n), q, p_1, q_1) \tag{3.55}$$

we have to establish

$$(\mathrm{lsort}, E) \mid \Gamma \vdash_{\mathrm{Hoare}} \{p_0(\mathbf{new}\ x := t\ \mathbf{in}\ S)(\mathrm{lsort}, E)(n)\}\ \mathbf{new}\ x := t\ \mathbf{in}\ S\ \{q\}$$

where $\Gamma = \{p_1\}\ \mathbf{call}\ \{q_1\}$. Appealing to the rule for blocks (3.43), it suffices to show

$$\begin{aligned} (\mathrm{lsort}_x, E \cup \{x\}) \mid \mathrm{Pr}(\Gamma, x, v) \vdash_{\mathrm{Hoare}} &\left\{\mathrm{In}\left(p', x, v, t, E\right)\right\}\ S\ \{\mathrm{uL}\,(q, x, v)\} \\ &\text{where } p' \stackrel{\mathrm{def}}{=} p_0(\mathbf{new}\ x := t\ \mathbf{in}\ S)(\mathrm{lsort}, E)(n) \end{aligned} \tag{3.56}$$

for an arbitrary constant $v : \mathrm{lsort}(x)$. From now on in the proof, we may refer to v because we had chosen to universally quantify v at the level of a sequent. We now exploit the induction hypothesis and instantiate the precondition with $\mathrm{Pr}(p_1, x, v)$ and the postcondition with $\mathrm{Pr}(q_1, x, v)$. This yields

$$(\mathrm{lsort}_x, E \cup \{x\}) \mid \mathrm{Pr}(\Gamma, x, v) \vdash_{\mathrm{Hoare}} \{p_0(S)(\mathrm{lsort}, E)(n)\}\ S\ \{q\} \tag{3.57}$$

provided the side-condition

$$\mathrm{KSC}\big(\hat{\psi}(\mathrm{lsort},E)(n),q,\mathrm{Pr}(p_1,x,v),\mathrm{Pr}(p_2,x,v)\big) \tag{3.58}$$

is satisfied. The rule of consequence mediates between (3.57) and (3.56). Thus, in addition to (3.58), we are left with having to establish

$$\mathrm{KSC}\big(\mathrm{In}(p',x,v,t,E),\mathrm{uL}(q,x,v),p_0(S)(\mathrm{lsort},E)(n),q\big) \quad . \tag{3.59}$$

(3.59) Expanding the definitions of KSC, p', In, p_0 and uL, after some straightforward simplifications, this side-condition is similar to the corresponding one in the proof of the MGF theorem on page 83.

(3.58) Given states $Z,\sigma,\tau : \Sigma(\mathrm{lsort}_x)$ such that for some $u < n$, we have

$$(\mathrm{lsort}_x, E\cup\{x\}) \mid \sigma \xrightarrow{\textbf{call}}_{u+1} Z \quad , \tag{3.60}$$

we have to extract auxiliary variables Z' and v' such that

$$p_1\big(Z',\mathrm{uL}(\sigma,x,v')\big) \tag{3.61}$$

and

$$\Big(q_1\big(Z',\mathrm{uL}(\tau,x,v')\big)\wedge\tau.\mathrm{local}(x)=\sigma.\mathrm{local}(x)\Big)\Rightarrow Z=\tau \quad . \tag{3.62}$$

Having universally quantified v in the premiss of the rule for blocks, we still have a suitable candidate for v' in scope. Otherwise, one would need to restrict the class of sorts so that one can effectively generate an element for every data type.

In order to utilise (3.55), we have to lift the assumption (3.60) to

$$(\mathrm{lsort},E) \mid \mathrm{uL}(\sigma,x,v) \xrightarrow{\textbf{call}}_{n} \mathrm{uL}(\tau,x,v)$$

by appealing to Cor. 3.42.2.

From (3.55), we extract the desired auxiliary variable Z'. It satisfies (3.61) and

$$q_1\big(Z',\mathrm{uL}(\tau,x,v)\big)\Rightarrow \mathrm{uL}(Z,x,v)=\mathrm{uL}(\tau,x,v) \quad . \tag{3.63}$$

Dealing with the remaining proof obligation (3.62), assume

$$q_1\big(Z',\mathrm{uL}(\tau,x,v)\big) \qquad \text{and} \qquad \tau.\mathrm{local}(x)=\sigma.\mathrm{local}(x) \quad .$$

By (3.63), it suffices to show that the local entry of x in the state spaces Z and σ are identical. This can be established by considering the derivation (3.60). According to Lemma 3.40, the same derivation holds in the empty environment. Thus, one may conclude the proof with the help of Cor. 3.42.1.

□

➥ p. 204 **Theorem 3.46 (MGF)**

$$(\mathrm{lsort}, E) \mid \emptyset \vdash_{\mathrm{Hoare}} \left\{ \lambda(Z,\sigma) \cdot (\mathrm{lsort}, E) \mid \sigma \xrightarrow{\quad S \quad} Z \right\} S \, \{q\}$$

Proof by induction on the structure of S.

Procedure Call We need to show

$$(\mathrm{lsort}, E) \mid \emptyset \vdash_{\mathrm{Hoare}} \left\{ \lambda(Z,\sigma) \cdot (\mathrm{lsort}, E) \mid \sigma \xrightarrow{\quad \mathbf{call} \quad} Z \right\} \mathbf{call} \, \{q\} \quad (3.64)$$

Applying the rule for procedure invocations (3.41) to Theorem 3.45 yields

$$(\mathrm{lsort}, E) \mid \emptyset \vdash_{\mathrm{Hoare}} \left\{ \lambda(Z,\sigma) \cdot \exists n \cdot (\mathrm{lsort}, E) \mid \sigma \xrightarrow{\quad \mathbf{call} \quad}_{n+1} Z \right\} \mathbf{call} \, \{q\} \quad . \quad (3.65)$$

We employ the rule of consequence to mediate between (3.65) and (3.64). For a justification of the side-condition, see the corresponding proof without blocks on page 68.

Blocks see the corresponding proof in the scenario without procedures on page 83.

□

➥ p. 204 **Corollary 3.47 (Completeness)**
Whenever $(\mathrm{lsort}, E) \mid \emptyset \vdash_{\mathrm{Hoare}} \{p\} \, S \, \{q\}$ *is derivable from the axioms and rules of Figure 3.10 on page 101,* $(\mathrm{lsort}, E) \mid \emptyset \models_{\mathrm{Hoare}} \{p\} \, S \, \{q\}$ *holds.*

Proof Completeness follows immediately from the MGF Theorem. See the proof of Corollary 2.31 on page 41 for further details. □

3.3.7.5 Admissible Rules

In examples, due to the left-constructive nature of the assignment axiom, it is useful to be able to appeal to admissible *left-constructive* versions of the consequence rule, see Figure 3.1.8 on page 72, and the rule for blocks, see Figure 3.6 on page 84. Moreover, we derive a rule for non-recursive procedure calls. When combining recursive procedures with local program variables, we need to annotate the rules with contexts and environments.

$$\frac{(\mathrm{lsort},\emptyset) \mid \emptyset \vdash_{\mathrm{Hoare}} \{p\}\ S_0\ \{q\}}{(\mathrm{lsort},E) \mid \emptyset \vdash_{\mathrm{Hoare}} \{p\}\ \mathbf{call}\ \{q\}} \tag{3.66}$$

$$\frac{(\mathrm{lsort},E) \mid \Gamma \vdash_{\mathrm{Hoare}} \{p_1\}\ S\ \{q_1\}}{(\mathrm{lsort},E) \mid \Gamma \vdash_{\mathrm{Hoare}} \{\hat{p}\}\ S\ \{q\}} \quad \text{with } \hat{p}(Z,\sigma) \stackrel{\mathrm{def}}{=} \exists Z_1 \cdot p_1(Z_1,\sigma) \wedge \forall \tau \cdot q_1(Z_1,\tau) \Rightarrow q(Z,\tau). \tag{3.67}$$

$$\frac{\forall v \cdot \big(\mathrm{lsort}[x \mapsto T], E \cup \{x\}\big) \mid \mathrm{Pr}(\Gamma,x,v) \vdash_{\mathrm{Hoare}} \{p\}\ S\ \{\mathrm{uL}(q,x,v)\}}{(\mathrm{lsort},E) \mid \Gamma \vdash_{\mathrm{Hoare}} \big\{\lambda(Z,\sigma) \cdot \mathrm{uL}\big(p,x,\mathrm{eval}(E,\sigma)(t)\big)\big\}\ \mathbf{new}\ x := t\ \mathbf{in}\ S\ \{q\}} \tag{3.68}$$

Figure 3.11: Admissible Rules for Static Binding in Hoare Logic

➥ p. 205

Corollary 3.48 (Soundness of Admissible Rules for Static Binding)
The rules from Figure 3.11 are admissible.

Proof

Procedure Call By soundness and completeness it suffices to show that

$$(\mathrm{lsort},\emptyset) \mid \sigma \xrightarrow{S_0} \tau$$

implies

$$(\mathrm{lsort},E) \mid \sigma \xrightarrow{\mathbf{call}} \tau\ .$$

This follows immediately from the operational semantics.

Adaptation The rule (3.67) is a trivial instance of the consequence rule (3.45).

Blocks Given

$$\forall v \cdot \big(\mathrm{lsort}[x \mapsto T], E \cup \{x\}\big) \mid \mathrm{Pr}(\Gamma,x,v) \vdash_{\mathrm{Hoare}} \{p\}\ S\ \{\mathrm{uL}(q,x,v)\}$$

we need to establish

$$(\mathrm{lsort},E) \mid \Gamma \vdash_{\mathrm{Hoare}} \big\{\lambda(Z,\sigma) \cdot \mathrm{uL}\big(p,x,\mathrm{eval}(E,\sigma)(t)\big)\big\}\ \mathbf{new}\ x := t\ \mathbf{in}\ S\ \{q\} \tag{3.69}$$

By the rule of consequence, we may replace p by

$$\lambda(Z,\sigma) \cdot \mathrm{In}\Big(\mathrm{uL}\big(p,x,\mathrm{eval}(E,\sigma)(t)\big),x,v,t,E\Big)\ .$$

Then, the rule for blocks (3.43) is applicable and yields the desired (3.69).

□

3.3.7.6 Example

We return to Example 3.31 on page 87 to illustrate how the new set of rules disambiguates global and local program variables.

➥ p. 206

Lemma 3.49 (Static Binding)
Let E be an arbitrary environment (lsort, E) *such that* $E \cap \{x, z\} = \emptyset$ *i.e., x and z are* global *program variables. Given the procedure declaration* $S_0 \stackrel{\text{def}}{=} x := z$*, one may derive* $(\mathrm{lsort}, E) \mid \emptyset \vdash_{\text{Hoare}} \{\mathbf{true}\}\ z := 1;\ \mathbf{new}\ z := 0\ \mathbf{in\ call}\ \{x = 1\}$.

Proof Semantically, the postcondition $[\![x = 1]\!](Z, \tau)$ corresponds to

$$\mathrm{lookup}(E, \tau)(x) = 1$$

which expands to $\tau.\,\mathrm{global}(x) = 1$. Thus, updating the value of local program variables has no effect. We follow the convention introduced in Example 3.35 on page 90 that $[\![x]\!](Z, \sigma) = \sigma.\,\mathrm{global}(x)$ and $[\![\dot{x}]\!](Z, \sigma) = \sigma.\,\mathrm{local}(x)$.

Appealing to the rule of sequential composition, we split the proof obligation into

$$(\mathrm{lsort}, E) \mid \emptyset \vdash_{\text{Hoare}} \{\mathbf{true}\}\ z := 1\ \{?i_1\} \tag{3.70}$$

$$(\mathrm{lsort}, E) \mid \emptyset \vdash_{\text{Hoare}} \{?i_1\}\ \mathbf{new}\ z := 0\ \mathbf{in\ call}\ \{x = 1\} \tag{3.71}$$

where $?i_1$ is an intermediate proof obligation which can be synthesised courtesy of the *left-constructive* version of the rule for blocks. Refining the second correctness formula (3.71) by the rule (3.68) yields

$$(\mathrm{lsort}[z \mapsto \mathtt{nat}], E \cup \{z\}) \mid \emptyset \vdash_{\text{Hoare}} \{?i_2\}\ \mathbf{call}\ \{x = 1\}$$

where

$$?i_1 = \mathrm{uL}(?i_2, z, 0)\ . \tag{3.72}$$

We then use rule (3.66) to cater for the (non-recursive) procedure invocation. This gives us the new subgoal $(\mathrm{lsort}[z \mapsto \mathtt{nat}], \emptyset) \mid \emptyset \vdash_{\text{Hoare}} \{?i_2\}\ x := z\ \{x = 1\}$. By the axiom of assignments, the precondition $?i_2$ gets bound to $z = 1$ where, due to the environmental change caused by the procedure call, z refers to the value of the *global* program variable. Hence, synthesising $?i_1$ according to (3.72) leads to $z = 1$ (and not $0 = 1$). With this instantiation, it is straight-forward to derive the remaining proof obligation (3.70). □

3.3.8 The Vienna Development Method

We extend the rules of Figure 3.4 on page 76 with environments and a rule for blocks. In the procedure invocation rule, the premiss is restricted to the empty environment to enforce static scoping. The conclusion holds for an arbitrary number of local variables. In the rule for blocks, we need to also adjust the environment of the context. We had already discussed the required modifications for VDM in Sect. 3.3.7.2 on page 99.

➥ p. 213 **Theorem 3.50 (Soundness)** *The rules of Figure 3.12 on the next page are sound.*

Proof by induction on the derivation of correctness formulae. □

➥ p. 214 **Theorem 3.51 (Completeness)** *The rules of Figure 3.12 on the facing page are complete.*

Proof Similar to the corresponding proof in the setting of Hoare Logic. □

3.4 Parameter Passing

For practical applications, it is clearly important to support procedures with parameters. In this section we discuss call-by-value and call-by-name parameters.

Let $S_0(T\ \mathbf{val}\ a, U\ \mathbf{var}\ x)$ be a procedure declaration in which the program S_0 may refer to the program variables a and x with sorts T and U. One refers to a as a formal value parameter and to x as a formal variable parameter. The procedure is invoked by $\mathbf{call}\ (t, v)$ where t is an expression and v a program variable. One refers to t as the actual value parameter and to v as the actual variable parameter (Dahl 1992).

On procedure entry, the expression t is evaluated in the current environment and its value is assigned to the variable a. With call-by-name passing, the variable x is aliased to v i.e., in executing S_0, all occurrences of x are effectively replaced by v. To adequately deal with call-by-name parameters, it seems advisable to abandon the implicit stack encoding of states. Ah-kee (1990) develops VDM proof rules for parameter passing based on the address mechanism.

Call-by-value parameters are easier to deal with because this mechanism can be simulated in the presence of local program variables (Morgan 1988). One allocates a new variable a, initialises it with the value of t and proceeds with executing S_0:

$$\frac{(\mathrm{lsort}[a \mapsto T], \{a\}) \mid \mathrm{uL}\big(\sigma, a, \mathrm{eval}(E, \sigma)(t)\big) \xrightarrow{\quad S_0 \quad} \tau}{(\mathrm{lsort}, E) \mid \sigma \xrightarrow{\ \mathbf{call}\ t\ } \mathrm{uL}\big(\tau, a, \sigma.\mathrm{local}(a)\big)}$$

However, one needs to take care to evaluate the expression t in the correct environment E. Let us first consider the case where all procedure invocations employ the

$$(\mathrm{lsort}, E) \mid \{p\}\ \mathbf{call}\ \{q\} \vdash \{p\}\ \mathbf{call}\ \{q\}$$

$$(\mathrm{lsort}, E) \mid \Gamma \vdash \{\lambda\sigma \cdot \mathbf{true}\}\ \mathbf{skip}\ \{\lambda(\tau,\sigma) \cdot \tau = \sigma\}$$

$$(\mathrm{lsort}, E) \mid \Gamma \vdash \{\lambda\sigma \cdot \mathbf{true}\}\ x := t\ \{\lambda(\tau,\sigma) \cdot \tau = \sigma[x \mapsto \mathrm{eval}(\sigma)(t)]\}$$

$$\frac{(\mathrm{lsort}, E) \mid \Gamma \vdash \{p_1\}\ S_1\ \{\lambda(\tau,\sigma) \cdot p_2(\tau) \wedge r_1(\tau,\sigma)\} \quad (\mathrm{lsort}, E) \mid \Gamma \vdash \{p_2\}\ S_2\ \{r_2\}}{(\mathrm{lsort}, E) \mid \Gamma \vdash \{p_1\}\ S_1;\ S_2\ \{r_2 \circ r_1\}}$$

$$\frac{\begin{array}{c}(\mathrm{lsort}, E) \mid \Gamma \vdash \{\lambda\sigma \cdot p(\sigma) \wedge \mathrm{eval}(\sigma)(b) = \mathbf{true}\}\ S_1\ \{q\} \\ (\mathrm{lsort}, E) \mid \Gamma \vdash \{\lambda\sigma \cdot p(\sigma) \wedge \mathrm{eval}(\sigma)(b) = \mathbf{false}\}\ S_2\ \{q\}\end{array}}{(\mathrm{lsort}, E) \mid \Gamma \vdash \{p\}\ \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \{q\}}$$

$$\frac{(\mathrm{lsort}, E) \mid \Gamma \vdash \{\lambda\sigma \cdot p(\sigma) \wedge \mathrm{eval}(\sigma)(b) = \mathbf{true}\}\ S\ \{\lambda(\tau,\sigma) \cdot p(\tau) \wedge \mathit{sofar}(\tau,\sigma)\}}{(\mathrm{lsort}, E) \mid \Gamma \vdash \{p\}\ \mathbf{while}\ b\ \mathbf{do}\ S\ \{q\}}$$

where *sofar* is any binary transitive relation on the state space Σ such that

$\lambda(\tau,\sigma) \cdot \mathit{sofar}(\tau,\sigma) \wedge p(\tau)$ is well-founded

and $q(\tau,\sigma) \stackrel{\mathrm{def}}{=} p(\tau) \wedge \mathrm{eval}(\tau)(b) = \mathbf{false} \wedge \big(\mathit{sofar}(\tau,\sigma) \vee \tau = \sigma\big)$.

$$\frac{\forall t : W \cdot (\mathrm{lsort}, \emptyset) \mid \{\lambda\sigma \cdot \exists u \cdot p(u)(\sigma) \wedge u < t\}\ \mathbf{call}\ \{q\} \vdash \{p(t)\}\ S_0\ \{q\}}{(\mathrm{lsort}, E) \mid \emptyset \vdash \{\lambda\sigma \cdot \exists t : W \cdot p(t)(\sigma)\}\ \mathbf{call}\ \{q\}}$$

where $(W, <)$ is well-founded.

$$\frac{\forall v \cdot \big(\mathrm{lsort}[x \mapsto T], E \cup \{x\}\big) \mid \mathrm{Pr}(\Gamma, x, v) \vdash_{\mathrm{VDM}} \begin{array}{c}\{\mathrm{In}(p,x,v,t,E)\} \\ S \\ \{\mathrm{uL}(q,x,v)\}\end{array}}{(\mathrm{lsort}, E) \mid \Gamma \vdash \{p\}\ \mathbf{new}\ x := t\ \mathbf{in}\ S\ \{q\}}$$

with $\mathrm{Pr}(\Gamma, x, v) \stackrel{\mathrm{def}}{=} \begin{cases} \emptyset & \text{for } \Gamma = \emptyset, \\ \{\mathrm{uL}(p', x, v)\}\ \mathbf{call}\ \{\mathrm{Pr}(q', x, v)\} & \text{for } \Gamma = \{p'\}\ \mathbf{call}\ \{q'\} \end{cases}$

$$\mathrm{Pr}(q', x, v)(\tau,\sigma) \stackrel{\mathrm{def}}{=} \mathrm{uL}(q', x, v)(\tau,\sigma) \wedge \tau.\mathrm{local}(x) = \sigma.\mathrm{local}(x)$$

$$\mathrm{In}(p, x, v, t, E)(\sigma) \stackrel{\mathrm{def}}{=} \mathrm{uL}(p, x, v)(\sigma) \wedge \sigma.\mathrm{local}(x) = \mathrm{eval}(E, \mathrm{uL}(\sigma, x, v))(t)\ .$$

$$\frac{(\mathrm{lsort}, E) \mid \Gamma \vdash \{p_1\}\ S\ \{q_1\}}{(\mathrm{lsort}, E) \mid \Gamma \vdash \{p\}\ S\ \{q\}}$$

provided $\forall\sigma \cdot p(\sigma) \Rightarrow p_1(\sigma)$ and $\forall(\tau,\sigma) \cdot p(\sigma) \Rightarrow q_1(\tau,\sigma) \Rightarrow q(\tau,\sigma)$.

Figure 3.12: VDM and Static Binding

same call-by-value parameter t. One may simulate all occurrences of **call** t by the procedure **new** $a := t$ **in** S_0 if the actual value parameter t refers only to *global* program variables. In particular, restricting our attention to non-recursive procedures and partial correctness, in the case where the expression t amounts to a global program variable a', call-by-value parameter passing can be dealt with by the rule

$$\frac{\forall v \cdot \big(\mathrm{lsort}\,[a \mapsto T]\,, \{a\}\big) \mid \{\mathrm{In}(p,a,v,a',E)\}\ S_0\ \{\mathrm{uL}(q,a,v)\}}{(\mathrm{lsort}, E) \mid \{p\}\ \mathbf{call}\ a'\ \{q\}} \qquad \text{for } a' \notin E.$$

Dahl (1992) proposes dealing with the general case of arbitrary expressions t as actual value parameters by considering the translation

$$\begin{aligned} S_0 &\rightsquigarrow \mathbf{new}\ a := a'\ \mathbf{in}\ S_0 \\ \mathbf{call}\ t &\rightsquigarrow a' := t;\ \mathbf{call}\ a' \end{aligned} \tag{3.73}$$

This leads to the rule

$$\frac{\forall v \cdot \big(\mathrm{lsort}\,[a \mapsto T]\,, \{a\}\big) \mid \{\mathrm{In}(p,a,v,a',E)\}\ S_0\ \{\mathrm{uL}(q,a,v)\}}{(\mathrm{lsort}, E) \mid \big\{\lambda(Z,\sigma) \cdot p\big(Z, (\sigma.\,\mathrm{global}\,[a' \mapsto t]\,, \sigma.\,\mathrm{local})\big)\big\}\ \mathbf{call}\ t\ \{q\}} \qquad \text{for } a' \notin E.$$

It would be an interesting further research contribution to mechanically establish soundness and completeness for verification calculi dealing with parameter passing. In the following chapter, in verifying Quicksort, we avoid call-by-value parameter passing by directly translating all procedure calls and procedure bodies according to (3.73). Moreover, we replace call-by-name parameters by global program variables.

Chapter 4

Case Study: Quicksort

In the previous chapters, we have demonstrated that it is feasible to conduct soundness and completeness proofs for verification calculi on a machine. As a by-product, we have gained a new set of verification rules for imperative programs dealing with recursive (parameterless) procedures and local variables. Soundness and completeness are important meta-theoretical properties. However, it remains to be shown that the new verification calculi are suitable in practice. Clearly, we cannot really expect to verify industrial-size programs at this stage. Our results would need to be extended to better support modular program development. In particular, we cannot directly deal with multiple procedures or parameter passing.

In this chapter, using Hoare's (1961) Quicksort algorithm as an example, we show that our Hoare Logic rules lend themselves to proving the correctness of a concrete recursive procedure with local and global variables.

Verification systems such as Hoare Logic themselves do not actually impose a strict methodology on how to verify programs. At one end of the spectrum, one can derive a Verification Condition Generator (VCG) from the set of rules. Given a program annotated with loop invariants and specifications for recursive procedures, a VCG automatically produces a set of propositions, which, if proven correct, guarantee the correctness of the desired program. Using the B method (Abrial 1996) as the underlying verification system, this approach has for example been employed by *GEC Alsthom* to verify the speed- and switching-control software used in all of France's 6000 electric trains. On the other end of the spectrum, Dijkstra (1990), Gries (1981) and others advocate developing the program hand-in-hand[1] with its proof of correctness.

Our specification follows Reynolds's (1981) presentation. The arguments in the correctness proof are well-known. Foley & Hoare had already published a proof of partial correctness for Quicksort in 1971. In our presentation, besides also considering

[1]Formally, such an approach may be described by first-order deliverables (Burstall & McKinna 1993).

termination, we highlight the rôle of the strengthened consequence rule and the new rules required in the presence of recursive procedures and local variables.

The formal rigour of the correctness proof is an orthogonal issue. Current industrial applications seem to be aided by some tool support, but purely logical propositions (arising from a consequence rule) are mostly dealt with on paper only. We have conducted the complete verification using the LEGO system.

The outline of this chapter is as follows. We first sketch the required background theory of sorted arrays. We then present Reynolds's (1981) specification and discuss the modifications required so that the program can be expressed in our toy language framework. At the heart of the chapter lies the verification proof.

4.1 A Theory of Sorted Arrays

In his thesis, McKinna (1992) investigates correctness proofs of *functional* programs in LEGO. We chose to reuse a substantial amount of his theory of sorted lists, which McKinna formalised to justify the correctness of *Insert Sort*.

In an imperative scenario, we have to deal with arrays. These are quite different from lists. There is no recursive structure and any element can be accessed in constant time. For efficiency, the algorithms rely on call-by-name passing (or global variables) of arrays. An imperative sorting algorithm requires three arguments: the array, the lower and the upper bound. The verification requires a little more care, because one has to ensure that a recursive call only manipulates the array within the specified bounds. We decided to represent arrays as (total) function spaces. We chose $\mathtt{nat} \to \mathtt{nat}$ as the sort for the program variable x storing the array. Any entry in an array $x[c]$ corresponds to $x(c)$. An assignment to a particular field in an array can then be represented by updating, see Section 2.6.1 on page 15 for details.

As a disadvantage of reusing McKinna's (1992) scripts, we have to convert functions to lists relative to some array bounds. Thus, we can no longer directly exploit the structure of arrays. It might have been more elegant to develop a theory of sorting based directly on arrays. However, we believe this would have taken more time and decided it would not be worth the effort. We will keep the presentation short and informal.

4.1.1 Ordered Lists

Given an ordering relation such as $<$ on natural numbers, one can lift this relation pointwise to lists. We overload notation e.g., if k and l are lists of natural numbers, $k < l$ denotes that every element in the list k is less than every element in the list l.

Lifting is also useful to define the notion of an ordered list. For every element x in the list, every element appearing later than x in the list must be greater than or equal to x.

Definition 4.1 (Ordered Lists) *Let $R \subseteq T \times T$ be an arbitrary relation.*

$$\begin{aligned} \mathrm{ord}(R)([]) &= \textbf{true} \\ \mathrm{ord}(R)(a :: l) &= [a]\ R\ l \wedge \mathrm{ord}(R)(l) \end{aligned}$$

4.1.2 Subarrays

In sorting algorithms such as Quicksort, one needs to assert that a subarray, characterised by array boundaries, is sorted. Representing arrays as functions, we have implemented a function $\mathrm{natl}(x)(a,b)$, which, given a function x with domain `nat` and boundaries a, b : `nat`, returns the list of all elements between a and b, intuitively, $[x(a), \ldots, x(b)]$.

The main idea behind Quicksort is to recursively partition the array into two subarrays such that all elements in the first are less than or equal to those in the second. This involves swapping of elements. Due to the assignment axiom, on the level of assertions, one needs to update array elements. One therefore has to investigate how updating interacts with the translation from functions to lists. Either updating has no effect, because it affects elements outside the array bounds, or it changes an intermediate value and one may factorise the array accordingly.

➥ p. 217 **Lemma 4.2 (External Update)** *Let a, b and c be natural number such that either $c < a$ or $b < c$, then* $\mathrm{natl}(x[c \mapsto t])(a,b) = \mathrm{natl}(x)(a,b)$.

Proof by induction on the array bounds a and b for a fixed element t. □

➥ p. 217 **Lemma 4.3 (Factorisation)** *Let a, b and c be natural numbers such that $a \leq b \leq c$, then*

$$\begin{aligned} &\mathrm{natl}(x[b \mapsto t])(a,c) = \\ &\qquad (\textbf{if } b = 0 \textbf{ then } [] \textbf{ else } \mathrm{natl}(x)(a,b-1)) \mathbin{@} (t :: (\mathrm{natl}(x)(b+1,c))) \ . \end{aligned}$$

Notice that the conditional is necessary, because, for $a = b = 0$, we have by definition that $b - 1 = 0$: `nat`. Thus, $\mathrm{natl}(x)(a,b-1)$ reduces to $x(0)$ and not the required empty list.

Proof by induction on the array bounds a and c. The proof has an interesting twist in that one needs to generalise over an arbitrary term t. □

Having explored the effect of updating, we can show that swapping two elements via updating preserves the permutation property.

Definition 4.4 (Permutation) *McKinna defines $\sim$ as the smallest equivalence relation which identifies lists up to the commutativity of the append function, and is closed under adding a new element at the front:*

$$l \sim l \qquad \frac{k \sim l}{l \sim k} \qquad \frac{k \sim l \quad l \sim m}{k \sim m} \qquad k \mathbin{@} l \sim l \mathbin{@} k \qquad \frac{k \sim l}{(a :: k) \sim (a :: l)}$$

➥ p. 217 **Lemma 4.5 (Swapping Elements)** *Let a, b, c and d be natural numbers with $a \leq c \leq b$ and $a \leq d \leq b$. Then* $\mathrm{natl}(x[d \mapsto x(c)][c \mapsto x(d)])(a,b) \sim \mathrm{natl}(x)(a,b)$.

Proof by case analysis on whether $c < d$, $c = d$ or $c > d$. □

New in the imperative scenario is the desired property that elements outside the boundaries of the array are not affected by the sorting algorithm. Hence, we need to express that two arrays X and x are permutations within a specified region and identical otherwise.

➥ p. 219 **Definition 4.6 (Permutation on Subarrays)**

$$X \ {}_a{\sim}^b \ x \stackrel{\text{def}}{=} \\ \big(\mathrm{natl}(X)(a,b) \sim \mathrm{natl}(x)(a,b)\big) \wedge \big(\forall c \cdot (c < a \vee b < c) \Rightarrow X(c) = x(c)\big)$$

4.1.3 High-Level Correctness Argument

The array to be sorted undergoes a series of transformations during the process of Quicksort. First, it is reordered to form a partition such that elements in the first are smaller than or equal to all elements in the second. Then, the algorithm is applied recursively to the two partitions.

Starting with the original array X_0, let X_1 and X_2 denote the two intermediate arrays, and X_3 the final array. From the progress achieved in the individual steps, one needs to establish that X_3 is ordered, a permutation of X_0 and that none of the elements outside of the array boundaries have been modified.

➥ p. 219 **Definition 4.7 (Sorted Arrays)** $X \ {}_a\mathrm{Sorted}^b \ x \stackrel{\text{def}}{=} \mathrm{ord}(\leq)\big(\mathrm{natl}(x)(a,b)\big) \wedge X \ {}_a{\sim}^b \ x$

➥ p. 219 **Theorem 4.8 (Quicksort)** *Let a and b be the array boundaries, and c the intermediate position computed by the partitioning process such that $a < c \leq b$. Given*

$$X_0 \ {}_a{\sim}^b \ X_1 \tag{4.1}$$

$$\mathrm{natl}(X_1)(a, c-1) \leq \mathrm{natl}(X_1)(c, b) \tag{4.2}$$

$$X_1 \ {}_a\mathrm{Sorted}^{c-1} \ X_2 \tag{4.3}$$

$$X_2 \ {}_c\mathrm{Sorted}^{b} \ X_3 \tag{4.4}$$

one can justify $X_0 \ {}_a\mathrm{Sorted}^b \ X_3$.

Proof Combining the stability conditions outside the array boundaries of (4.3) and (4.4), we may exploit

$$\mathrm{natl}(X_1)(c, b) = \mathrm{natl}(X_2)(c, b) \tag{4.5}$$

$$\mathrm{natl}(X_2)(a, c-1) = \mathrm{natl}(X_3)(a, c-1) \tag{4.6}$$

We show how to establish $\mathrm{ord}(\leq)\big(\mathrm{natl}(X_3)(a, b)\big)$ in more detail. Appealing to the permutation properties of (4.3) and (4.4), as well as the equalities (4.5) and (4.6), from the partitioning (4.2), we may conclude

$$\mathrm{natl}(X_3)(a, c-1) \leq \mathrm{natl}(X_3)(c, b) \ . \tag{4.7}$$

Moreover, rewriting (4.3) with the help of (4.6) yields

$$\mathrm{ord}(\leq)\big(\mathrm{natl}(X_3)(a, c-1)\big) \tag{4.8}$$

Hence, combining (4.7), (4.8) and the order property of (4.4) established that the list $\mathrm{natl}(X_3)(a, b)$ is sorted. □

4.2 Specification of Quicksort

We follow Reynolds's (1981) top-down presentation. Relative to a program Prepare which partitions the array into two fragments k and l with $k \leq l$, we recursively sort k and l. A Hoare triple which characterises the behaviour of *Prepare* is

$\{a < b \wedge a = A \wedge b = B \wedge x = X\}$
Prepare
$\{A < c \leq B \wedge \mathrm{natl}(X)(A, c-1) \leq \mathrm{natl}(X)(c, B) \wedge x \ {}_A{\sim}^B \ X \wedge a = A \wedge b = B\}$

Lemma 4.9 (Rigorous Correctness of Quicksort) *We pass the boundaries a, b as call-by-values and the array x as call-by-name.*

procedure *quicksort*(nat **val** a,b; nat **array** **var** x);
$\{a = A \wedge b = B \wedge x = X\}$
if $a < b$ **then begin** nat c;
Prepare;
quicksort$(a, c-1, x)$; *quicksort*(c, b, x)
end
$\{X\ {}_A\text{Sorted}^B\ x\}$

The Prepare fragment builds on a general partition procedure

procedure *partition*(nat **val** a,b,r; nat **var** c; nat **array** **var** x);
$\{a = A \wedge b = B \wedge x = X\}$
begin nat d;
$c := a$; $d := b$
while $c \leq d$ **do**
if $x[c] \leq r$ **then** $c := c+1$ **else**
if $x[d] > r$ **then** $d := d-1$ **else**
begin
begin nat t; $t := x[c]$; $x[c] := x[d]$; $x[d] := t$ **end**;
$c := c+1$; $d := d-1$
end
end
$\{A \leq c \leq d+1 \wedge \text{natl}(x)(A, c-1) \leq r \wedge r < \text{natl}(x)(c, B) \wedge x\ {}_A\sim^B X \wedge$
$a = A \wedge b = B\}$

If Prepare were to simply call *partition*(a,b,r,c,x), there would be no guarantee that both partitions are strictly smaller. Without this additional property, the above Quicksort procedure may not terminate. Reynolds (1981) overcomes this problem by analysing the outermost elements $x[a]$ and $x[b]$ and swaps them in case $x[a] > x[b]$. By narrowing the partitioning algorithm to the subarray $\text{natl}(x)(a+1, b-1)$ for some intermediate pivot element r which lies between $x[a]$ and $x[b]$, one can guarantee that Prepare finds a value c such that $a < c \leq b$:

Definition 4.10 (Prepare)

$\{a < b \wedge a = A \wedge b = B \wedge x = X\}$
begin nat r;
if $x[a] > x[b]$ **then begin** nat t; $t := x[a]$; $x[a] := x[b]$; $x[b] := t$ **end**;
$r := (x[a] + x[b]) \div 2$;
partition$(a+1, b-1, r, c, x)$

end
$\{A < c \leq B \wedge \mathrm{natl}(X)(A, c-1) \leq \mathrm{natl}(X)(c, B) \wedge x \ {}_A{\sim}^B \ X \wedge a = A \wedge b = B\}$

4.2.1 Mechanisation

Being primarily interested in the meta-theory of verification calculi, we have chosen a shallow embedding for assertions. Unfortunately, this renders examples illegible. For the purpose of this chapter, we present assertions at the syntactic level. These have been translated by hand from the actual LEGO scripts.

All local program variables have sort `nat`. We therefore annotate correctness formulae only with a set of local program variables in scope E instead of an environment (lsort, E).

We only consider assertions where the domain of auxiliary variables coincides with the state space. In assertions, lowercase (free) variable names correspond to the value of program variables in the current state, whereas uppercase variable names correspond to auxiliary variables. Intuitively, auxiliary variables capture the initial value of program variables.

Example 4.11 *In an environment E recording all local variables, the assertion*

$$a = A \wedge b = B \wedge a < b$$

is to be parsed as

$$\begin{aligned}\lambda(Z, \sigma) \cdot \mathrm{lookup}(E, \sigma)(a) = \mathrm{lookup}(E, Z)(a) \wedge \\ \mathrm{lookup}(E, \sigma)(b) = \mathrm{lookup}(E, Z)(b) \wedge \mathrm{lookup}(E, \sigma)(a) < \mathrm{lookup}(E, \sigma)(b) \ .\end{aligned}$$

Our toy language does not support multiple procedures, nor parameter passing. Therefore, we need to reformulate the algorithm in more primitive terms. To circumvent parameter passing in Def. 4.9, we replace the call-by-name parameter x by a global program variable.

We eliminate call-by-value parameters by following the recipe of Sect. 3.4 on page 110. We declare the parameters a' and b' as global program variables. At the beginning of the procedure, we allocate two *local* variables a and b and initialise them with the values of a' and b' at procedure invocation time. Moreover, we have to take care of uninitialised local program variables, either by rearranging code, or, by choosing some arbitrary initial value.

➥ p. 221 **Definition 4.12 (Quicksort)**

$$
\begin{array}{l}
S_0 \stackrel{\text{def}}{=} \textbf{new } a := a' \textbf{ in new } b := b' \textbf{ in} \\
\quad \textbf{if } a < b \textbf{ then new } c := 0 \textbf{ in} \\
\quad\quad \textbf{begin} \\
\quad\quad\quad \text{Prepare}; \\
\quad\quad\quad a' := a;\ b' := c - 1 \\
\quad\quad\quad \textbf{call}; \\
\quad\quad\quad a' := c;\ b' := b; \\
\quad\quad\quad \textbf{call} \\
\quad\quad \textbf{end}
\end{array}
$$

As the partitioning algorithm is non-recursive, we macro-expand its invocation in Def. 4.10. As a further simplification, we set the pivot element to $x[a]$ instead of $(x[a]+x[b]) \div 2$, because, given $x[a] \leq x[b]$, it it easier to show $x[a] \leq x[a] \leq x[b]$ than $x[a] \leq (x[a]+x[b]) \div 2 \leq x[b]$.

➥ p. 220 **Definition 4.13** (Prepare)

$$
\begin{array}{l}
\text{Prepare} \stackrel{\text{def}}{=} \textbf{if } x[a] > x[b] \textbf{ then new } t := x[a] \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in begin } x[a] := x[b];\ x[b] := t \textbf{ end}; \\
\quad \textbf{new } r := x[a] \textbf{ in} \\
\quad\quad \textbf{begin} \\
\quad\quad\quad c := a + 1; \\
\quad\quad\quad \textbf{new } d := b - 1 \textbf{ in} \\
\quad\quad\quad\quad \textbf{while } c \leq d \textbf{ do} \\
\quad\quad\quad\quad\quad\quad \textbf{if } x[c] \leq r \textbf{ then } c := c + 1 \textbf{ else} \\
\quad\quad\quad\quad\quad\quad\quad \textbf{if } x[d] > r \textbf{ then } d := d - 1 \textbf{ else} \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{begin} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{new } t := x[c] \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{in begin} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad x[c] := x[d];\ x[d] := t \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{end}; \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad c := c + 1;\ d := d - 1 \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{end} \\
\quad\quad \textbf{end}
\end{array}
$$

The specification for the Quicksort algorithm remains the same. However, due to our encoding of parameter passing, we have to ensure that the program variables a', b' and x are global.

➥ p. 235 **Lemma 4.14 (Correctness of Quicksort)** *Let E be an arbitrary environment such that $E \cap \{a', b', x\} = \emptyset$. We can then derive*

$$E \mid \emptyset \vdash_{\text{Hoare}} \{a' = A' \wedge b' = B' \wedge x = X\} \\ \textbf{call} \\ \{X \ {}_{A'}\text{Sorted}^{B'} \ x\}$$

In order to prove this, we establish the intermediate result:

➥ p. 234 **Lemma 4.15 (Correctness of** Prepare**)** *Let E be an arbitrary environment*[2] *and Γ be an arbitrary context.*

$$E \mid \Gamma \vdash_{\text{Hoare}} \{a < b \wedge a = A \wedge b = B \wedge x = X\} \\ \text{Prepare} \\ \{\text{variant}_{\text{Prepare}}(x)(c) \wedge a = A \wedge b = B\}$$

where

➥ p. 222 **Definition 4.16 (Variant for** Prepare **Fragment)**

$$\text{variant}_{\text{Prepare}}(x)(c) \stackrel{\text{def}}{=} \\ A < c \leq B \wedge \text{natl}(X)(A, c-1) \leq \text{natl}(X)(c, B) \wedge x \ {}_A{\sim}^B \ X$$

An important and challenging aspect of the specification is to provide a suitable loop invariant. Based on Reynolds's (1981) version for the partition algorithm, we have employed the following assertion:

➥ p. 222 **Definition 4.17 (Invariant for** Prepare **Fragment)**

$$\text{inv}_{\text{Prepare}}(c, d) \stackrel{\text{def}}{=} A < c \leq d+1 \wedge d < B \wedge \\ \text{natl}(x)(A, c-1) \leq r \wedge r < \text{natl}(x)(d+1, B-1) \wedge r \leq x[B] \wedge \\ x \ {}_A{\sim}^B \ X \wedge a = A \wedge b = B$$

4.3 A Guided Tour of the Verification Proof

Our formalised verification calculus does not directly permit multiple procedures. The declaration of the procedure in Def. 4.12 contains only procedure activations of itself. We have used LEGO's support for definitions to emphasise that the Prepare fragment corresponds to a procedure invocation. In the correctness proof, we exploit LEGO's

[2] For simplicity, in the mechanised proof, we only consider the particular environment $E = \{a, b, c\}$ arising in the proof of Quicksort.

support for structuring proofs to *independently* verify the correctness of Prepare and Quicksort.

We commence by assuming Lemma 4.15 and show Lemma 4.14 i.e., that the skeleton of the Quicksort algorithm is correct. Afterwards, we complete the proof by verifying the Prepare implementation.

The main purpose of this section is to clarify how to use our set of Hoare Logic rules in practice. It is well-known that the whole process can be automated with a Verification Condition Generator.

4.3.1 Verification of Quicksort Skeleton

Our proof strategy is to employ the rule for well-founded recursion (3.41) on page 99. To use this we need to plant a termination measure in the precondition. For Quicksort, every call works on a smaller subarray. Its length is determined by the expression $(b'-a')+1$. We therefore employ the consequence rule (3.45) to equivalently transform the precondition to $\exists n \cdot n = (b'-a')+1 \wedge a' = A' \wedge b' = B' \wedge x = X$. Specialising rule (3.41) to the usual well-founded order $<$ on `nat` results in the subgoal

$$\begin{array}{l} \emptyset \mid \Gamma \vdash_{\text{Hoare}} \{n = (b'-a')+1 \wedge a' = A' \wedge b' = B' \wedge x = X\} \\ \qquad\qquad S_0 \\ \qquad\qquad \{X\ {}_{A'}\text{Sorted}^{B'}\ x\} \end{array}$$

where $n : \texttt{nat}$ is an arbitrary reference and

$$\begin{array}{l} \Gamma \stackrel{\text{def}}{=} \{\exists m \cdot m = (b'-a')+1 \wedge a' = A' \wedge b' = B' \wedge x = X \wedge m < n\} \\ \quad \textbf{call} \\ \quad \{X\ {}_{A'}\text{Sorted}^{B'}\ x\} \end{array}$$

The outermost constructors of the program S_0 successively determine which verification rules to apply. We commence by compensating for the local variable a. Since the pre- and postcondition do not mention another (local) program variable a, the rule for blocks (3.43) on page 100 only adds $a = a'$ in the precondition and $a = A$ in both assertions of the context. In particular, we have to derive

$$\begin{array}{l} \{a\} \mid \text{Pr}(\Gamma, a, v_a) \vdash_{\text{Hoare}} \{n = (b'-a')+1 \wedge a' = A' \wedge b' = B' \wedge x = X \wedge a = a'\} \\ \qquad\qquad \textbf{new}\ b := b'\ \textbf{in} \\ \qquad\qquad\quad \textbf{if}\ a < b\ \textbf{then new}\ c := 0\ \textbf{in} \\ \qquad\qquad\qquad \textbf{begin} \\ \qquad\qquad\qquad\quad \text{Prepare}; \\ \qquad\qquad\qquad\quad a' := a;\ b' := c-1 \\ \qquad\qquad\qquad\quad \textbf{call}; \end{array}$$

$$
\begin{array}{l}
\quad a' := c;\ b' := b; \\
\quad \textbf{call} \\
\textbf{end} \\
\{X\ {}_{A'}\mathrm{Sorted}^{B'}\ x\}
\end{array}
$$

Similarly, we remove the other local variable declarations and split the conditional. The case where the array to be sorted contains at most one element can be resolved easily. Otherwise, we have to show

$$
\begin{array}{l}
\{a,b,c\} \mid\ \Gamma_{a,b,c} \vdash_{\mathrm{Hoare}} \{n = (b'-a')+1 \wedge a' = A' \wedge b' = B' \wedge x = X \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad a = a' \wedge b = b' \wedge a < b \wedge c = 0\} \\
\qquad\qquad \mathrm{Prepare}; \\
\qquad\qquad a' := a;\ b' := c-1 \\
\qquad\qquad \textbf{call}; \\
\qquad\qquad a' := c;\ b' := b; \\
\qquad\qquad \textbf{call} \\
\qquad\qquad \{X\ {}_{A'}\mathrm{Sorted}^{B'}\ x\}
\end{array}
$$

where for

$$
\begin{array}{l}
p_1(A,A',B,B',C,X)(a,a',b,b',c,x) \stackrel{\mathrm{def}}{=} \\
\qquad\qquad \exists m \cdot m = (b'-a')+1 \wedge a' = A' \wedge b' = B' \wedge x = X \wedge m < n \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad a = A \wedge b = B \wedge c = C
\end{array}
$$

and

$$
\begin{array}{l}
q_1(A,A',B,B',C,X)(a,a',b,b',c,x) \stackrel{\mathrm{def}}{=} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad X\ {}_{A'}\mathrm{Sorted}^{B'}\ x \wedge a = A \wedge b = B \wedge c = C
\end{array}
$$

we define

$$
\begin{array}{rcl}
\Gamma_{a,b,c} & \stackrel{\mathrm{def}}{=} & \mathrm{Pr}\Big(\mathrm{Pr}\big(\mathrm{Pr}(\Gamma,a,v_a),b,v_b\big),c,v_c\Big) \\
& = & \{p_1(A,A',B,B',C,X)(a,a',b,b',c,x)\} \\
& & \quad \textbf{call} \\
& & \{q_1(A,A',B,B',C,X)(a,a',b,b',c,x)\}
\end{array} \quad .
$$

The assignment axiom can cope with an arbitrary postcondition. Hence, it is advisable, to break down the sequence of statements in reverse order. Splitting off the last procedure invocation yields

$$
\{a,b,c\} \mid\ \Gamma_{a,b,c} \vdash_{\mathrm{Hoare}} \{n = (b-a)+1 \wedge a' = A' \wedge b' = B' \wedge x = X \wedge
$$

$$a' = a \wedge b' = b \wedge a' < b' \wedge c = 0\}$$

Prepare;
$a' := a;\ b' := c - 1$
call
$\{?i[b \mapsto b'][a \mapsto c]\}$

and

$$\{a,b,c\} \mid\ \Gamma_{a,b,c} \vdash_{\text{Hoare}}\ \{?i\}\,\mathbf{call}\,\{X\ {}_{A'}\text{Sorted}^{B'}\ x\} \tag{4.9}$$

where $?i$ is an intermediate assertion to be synthesised.

Ideally, $?i$ should be as weak as possible. It can be computed with the help of the rule of adaptation (3.67) applied to (4.9) and the correctness formula in the context $\Gamma_{a,b,c}$. This yields the precondition

$$\begin{aligned}&\exists A_1, A_1', B_1, B_1', C_1, X_1 \cdot p_1(A_1, A_1', B_1, B_1', C_1, X_1)(a, a', b, b', c, x) \wedge \\ &\quad \forall a_1, a_1', b_1, b_1', c_1, x_1 \cdot \big(q_1(A_1, A_1', B_1, B_1', C_1, X_1)(a_1, a_1', b_1, b_1', c_1, x_1) \Rightarrow \\ &\qquad\qquad X\ {}_{A'}\text{Sorted}^{B'}\ x_1\big)\end{aligned}$$

The existential quantifiers are uniquely determined by the assertion p_1. Furthermore, we can simplify q_1 because the variables a_1, a_1', b_1, b_1' and c_1 do not occur in the assertion $X\ {}_{A'}\text{Sorted}^{B'}\ x_1$. Thus, we may instantiate

$$?i = (b' - a') + 1 < n \wedge \big(\forall X_3 \cdot x\ {}_{A'}\text{Sorted}^{B'}\ X_3 \Rightarrow X\ {}_{A'}\text{Sorted}^{B'}\ X_3\big)$$

Adopting the same strategy, we remove the first recursive procedure invocation and are left with having to show

$$\begin{aligned}\{a,b,c\} \mid\ \Gamma_{a,b,c} \vdash_{\text{Hoare}}\ &\{n = (b' - a') + 1 \wedge a' = A' \wedge b' = B' \wedge x = X \wedge \\ &\qquad a = a' \wedge b = b' \wedge a < b \wedge c = 0\}\end{aligned}$$

Prepare
$\{((c-1) - a') + 1 < n \wedge (b' - c) + 1 < n \wedge$
$\big(\forall X_2 \cdot x\ {}_{a'}\text{Sorted}^{c-1}\ X_2 \Rightarrow$
$\quad \forall X_3 \cdot X_2\ {}_{c}\text{Sorted}^{b'}\ X_3 \Rightarrow X\ {}_{A'}\text{Sorted}^{B'}\ X_3\big)\}$

One could continue this strategy and push the cumulative postcondition through the Prepare fragment. Instead, we follow Reynolds's specification in which Prepare is a procedure and proceed by stepwise-refinement i.e., we employ the consequence rule (3.45) to resolve the open goal with the help of the postulated correctness of the Prepare fragment according to Lemma 4.15. We then independently establish the correctness of the partitioning for a concrete implementation.

Notice that we require the strengthened consequence rule (3.45) to exploit $n = (b' - a') + 1$, a property exclusively recorded in the *precondition* when resolving the termination guarantees $\big((c-1)-a'\big)+1 < n$ and $(b'-c)+1 < n$ of the *postcondition*. More precisely, modulo some trivial simplifications, our strengthened consequence rule generates the following side-condition:

$$
\begin{array}{l}
n = (b'-a')+1 \wedge a' = A' \wedge b' = B' \wedge x = X \wedge a = A' \wedge b = b' \wedge a < b \Rightarrow \\
\quad \forall c, X_1 \cdot \mathrm{variant}_{\mathrm{Prepare}}(X_1)(c) \Rightarrow \\
\qquad \big((c-1)-a'\big)+1 < n \wedge (b'-c)+1 < n \wedge \\
\qquad \big(\forall X_2 \cdot x\ {}_{a'}\mathrm{Sorted}^{c-1}\ X_2 \Rightarrow \forall X_3 \cdot X_2\ {}_{c}\mathrm{Sorted}^{b'}\ X_3 \Rightarrow X\ {}_{A'}\mathrm{Sorted}^{B'}\ X_3\big)\big)
\end{array}
$$

The inequalities are easily resolved. The last part of the conjunction corresponds to Theorem 4.8 on page 116 in the presence of $\mathrm{variant}_{\mathrm{Prepare}}(X_1)(c)$.

4.3.2 Verification of the Prepare fragment

The only interesting aspect in this proof revolves around the loop. Using the sequential rule, we first split the proof obligation into

$$
\begin{array}{l}
E \mid \Gamma \vdash_{\mathrm{Hoare}} \{a < b \wedge a = A \wedge b = B \wedge x = X\} \\
\qquad\qquad \textbf{if } x[a] > x[b] \textbf{ then new } t := x[a] \\
\qquad\qquad\qquad\qquad\qquad \textbf{in begin } x[a] := x[b];\ x[b] := t \textbf{ end} \\
\qquad\qquad \{?i_1\}
\end{array}
$$

and

$$
\begin{array}{l}
E \mid \Gamma \vdash_{\mathrm{Hoare}} \{?i_1\} \\
\qquad\qquad \textbf{new } r := x[a] \textbf{ in} \\
\qquad\qquad\quad \textbf{begin} \\
\qquad\qquad\qquad c := a+1; \\
\qquad\qquad\qquad \textbf{new } d := b-1 \textbf{ in} \\
\qquad\qquad\qquad\quad \textbf{while } c \le d \textbf{ do} \\
\qquad\qquad\qquad\qquad\quad \textbf{if } x[c] \le r \textbf{ then } c := c+1 \textbf{ else} \\
\qquad\qquad\qquad\qquad\qquad \textbf{if } x[d] > r \textbf{ then } d := d-1 \textbf{ else} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{begin} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{new } t := x[c] \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in begin} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad x[c] := x[d];\ x[d] := t \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{end;} \\
\qquad\qquad\qquad\qquad\qquad\qquad c := c+1;\ d := d-1 \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{end}
\end{array}
$$

end

$\{\text{variant}_{\text{Prepare}}(x)(c) \wedge a = A \wedge b = B\}$

where the intermediate assertion $?i_1$ is eventually uniquely determined by the loop invariant and can be synthesised, provided we employ the *left-constructive* version of the rule for dismantling blocks (3.68) which unifies

$$?i_1 = \text{uL}(?i_2, r, x[a])$$

and yields the proof obligation

$E \cup \{r\} \mid \text{Pr}(\Gamma, r, v_r) \vdash_{\text{Hoare}} \{?i_2\}$

$c := a + 1;$
new $d := b - 1$ **in**
while $c \leq d$ **do**
if $x[c] \leq r$ **then** $c := c + 1$ **else**
if $x[d] > r$ **then** $d := d - 1$ **else**
begin
new $t := x[c]$
in begin
$x[c] := x[d];\ x[d] := t$
end;
$c := c + 1;\ d := d - 1$
end

$\{\text{variant}_{\text{Prepare}}(x)(c) \wedge a = A \wedge b = B\}$

We employ the sequential, assignment and left-constructive rule for blocks to successively decompose the program fragment until we reach the loop. This process yields another intermediate assertion $?i_3$ such that

$$?i_2 = ?i_3\,[c \mapsto a + 1]$$
$$?i_3 = \text{uL}\big(\text{inv}_{\text{Prepare}}(c, d), d, b - 1\big)$$

and

$E \cup \{r, d\} \mid \Gamma_{r,d} \vdash_{\text{Hoare}}$ **while** $c \leq d$ **do**

if $x[c] \leq r$ **then** $c := c + 1$ **else**
if $x[d] > r$ **then** $d := d - 1$ **else**
begin
new $t := x[c]$
in begin

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad x[c] := x[d];\ x[d] := t \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{end}; \\
\qquad\qquad\qquad\qquad\quad c := c+1;\ d := d-1 \\
\qquad\qquad\qquad\qquad \textbf{end} \\
\qquad\qquad\qquad \{\mathrm{variant}_{\mathrm{Prepare}}(x)(c) \wedge a = A \wedge b = B\}
\end{array}
$$

with

$$\Gamma_{r,d} \stackrel{\text{def}}{=} \mathrm{Pr}\big(\mathrm{Pr}(\Gamma, r, v_r), d, v_d\big) \quad .$$

Dealing with the loop, we apply the consequence rule to strengthen the postcondition to $\mathrm{inv}_{\mathrm{Prepare}}(c,d) \wedge \neg(c \leq d)$. Progress is achieved by extending the first segment, the second segment, or both. A suitable termination measure keeps track of the length of the intermediate segment $(d+1) - c$. Thus, applying the rule for well-founded loops (3.44), for a constant $w : (\mathtt{nat}, <)$, we are left with having to establish

$$
\begin{array}{l}
E \cup \{r,d\} \ \mid\ \Gamma_{r,d} \ \vdash_{\mathrm{Hoare}} \{\mathrm{inv}_{\mathrm{Prepare}}(c,d) \wedge c \leq d \wedge (d+1) - c = w\} \\
\qquad\qquad\qquad\qquad \textbf{if } x[c] \leq r \textbf{ then } c := c+1 \textbf{ else} \\
\qquad\qquad\qquad\qquad\quad \textbf{if } x[d] > r \textbf{ then } d := d-1 \textbf{ else} \\
\qquad\qquad\qquad\qquad\qquad \textbf{begin} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{new } t := x[c] \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in begin} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad x[c] := x[d];\ x[d] := t \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{end}; \\
\qquad\qquad\qquad\qquad\qquad\quad c := c+1;\ d := d-1 \\
\qquad\qquad\qquad\qquad\qquad \textbf{end} \\
\qquad\qquad\qquad\qquad \{\mathrm{inv}_{\mathrm{Prepare}}(c,d) \wedge (d+1) - c < w\}
\end{array}
$$

The cases where either $x[c] \leq r$ or $x[d] > r$ are straightforward. Otherwise, we have to show

$$
\begin{array}{l}
E \cup \{r,d\} \ \mid\ \Gamma_{r,d} \ \vdash_{\mathrm{Hoare}} \{\mathrm{inv}_{\mathrm{Prepare}}(c,d) \wedge c \leq d \wedge (d+1) - c = w \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad \neg(x[c] \leq r) \wedge \neg(x[d] > r)\} \\
\qquad\qquad\qquad\qquad \textbf{new } t := x[c] \\
\qquad\qquad\qquad\qquad\quad \textbf{in begin} \\
\qquad\qquad\qquad\qquad\qquad\quad x[c] := x[d];\ x[d] := t \\
\qquad\qquad\qquad\qquad\qquad \textbf{end} \\
\qquad\qquad\qquad\qquad \{\mathrm{inv}_{\mathrm{Prepare}}(c+1)(d-1) \wedge ((d-1)+1) - (c+1) < w\}
\end{array}
$$

To complete the proof, we need to appeal once more to the rule of consequence. We could either

1. commence by applying the top-down version of the rule for blocks (3.43) and appeal to the consequence rule when dealing with the assignment $x[c] := x[d]$.

2. first apply the rule of consequence so that the left-constructive version of the rule for blocks (3.11) becomes applicable. One can then immediately resolve $x[c] := x[d]$ by the axiom for assignments.

We have pursued the first option. Given

$$\begin{gathered} \mathrm{inv}_{\mathrm{Prepare}}(c,d) \\ c \leq d \wedge \neg(x[c] \leq r) \wedge \neg(x[d] > r) \\ (d+1) - c = w \\ t = x[c] \end{gathered} \tag{4.10}$$

we have to show

$$\mathrm{inv}_{\mathrm{Prepare}}(c+1)(d-1)\,[x \mapsto x[d \mapsto t]]\,[x \mapsto x[c \mapsto x[d]]] \qquad \text{(Invariant)}$$

$$\big((d-1)+1\big) - (c+1) < w \qquad \text{(Termination)}$$

Invariant Having represented arrays as functions, we need to remove redundant updates by appealing to the Lemmata 4.2 and 4.5.

Termination The premiss (4.10) is equivalent to $c < d$.

4.4 Conclusions

Much of the discussion in the above proofs has been focussed on choosing appropriate verification rules at strategic points and explaining the resulting proof obligations. Our main motivation has been to clarify the effect of the new set of rules by way of an example.

The only challenging aspect is to devise suitable invariants for loops. Furthermore, one needs to have a succinct specification of the recursive procedure. In an early proof attempt, we had forgotten to specify that array elements outside the array bounds remain invariant.

Given such invariants and specifications of recursive procedures, it is well-known that the complete verification can be automated by a VCG. This produces the same set of purely logical obligations arising from side-conditions of the consequence rule which we had to deal with after manually applying the axioms and rules of Hoare Logic.

Our proof has been unnecessarily complicated, because we have no support for parameter passing at the level of Hoare Logic. We had to introduce further global and local program variables. In particular, it caused more proof obligations involving values of the additional program variables.

LEGO's implementation of existential variables (such as ?*i*) has been very helpful when applying verification rules. It allowed us to apply them without having to specify the assertions. Later in the correctness proof, LEGO synthesised these automatically.

Unfortunately, it turned out to be time consuming to prove the arising (purely logical) verification conditions in LEGO. Due to our choice of a shallow embedding of assertions, the proof tool has no syntactic guidance to reduce the resulting excessively large proof obligations[3].

To check the correctness of the Quicksort algorithm, LEGO had to run for more than 37 hours requiring more than 80MB on a SUN SPARC station 20 with sufficient physical memory to avoid swapping[4]. To speed up the process, one could either employ a deep embedding of assertions, or directly guide the system to simplify proof obligations.

[3] See pages 222–234.

[4] In comparison, on the same architecture, the completeness proof for Hoare Logic dealing with recursive procedures and local variables could be dealt with in less than 15 minutes requiring less than 25MB. In both cases, we started LEGO in the empty environment.

Chapter 5

Conclusions and Future Work

If a verification calculus is unsound, one may derive properties of a program which do not hold. If it is incomplete, one may not be able to derive a desired property of a program despite it being satisfied. It is well known that establishing soundness and completeness for verification calculi is a challenging task. Many incorrect results based on doing proofs by hand have been published in the past.

We have taken advantage of the LEGO system to interactively derive machine-checked soundness and completeness proofs for Hoare Logic and the operation decomposition rules of the Vienna Development Method (VDM). We have dealt with parameterless recursive procedures and local variables in the context of total correctness. Previously, machine-checked completeness results had only been established for simple imperative programs without recursive procedures in a scenario of partial correctness.

With computer-aided proof systems, one attains a high-level of confidence concerning the correctness of machine-checked proofs e.g., in LEGO, less than 200 lines of SML code are responsible for checking that a given derivation is correct. Today's proof tools are sufficiently advanced to be of significant help in investigating meta-theory of verification calculi. In our opinion, soundness and completeness are difficult concepts which have to be tackled by appealing to intricate induction principles. A computer-aided proof system is very good at managing such tasks.

5.1 Syntax of Assertions

To prove completeness, one needs to be able to construct assertions which express semantic properties of the programming language. One usually simply assumes that the assertion language is sufficiently expressive. Both soundness and completeness proofs can be simplified if one does not worry about the actual syntactic representation of assertions.

A disadvantage of this approach is that examples are more difficult to read. Moreover, verifying the Quicksort algorithm, we found that the resulting proof obligations arising from the side-condition of the rule of consequence become too large for the LEGO system to efficiently process. Without syntactic structure, the proof tool has little guidance on how to reduce such proof obligations.

5.2 The Rôle of Auxiliary Variables in Hoare Logic

The required formal detail in a completely machine-checked proof encourages one to more critically analyse the theory under investigation. Often, simplifications are vital for a machine-checked development to be feasible. As our main contribution, we have illuminated the rôle of auxiliary variables in Hoare Logic. They are required to relate the value of program variables in *different* states. We felt that, in previous Hoare Logic calculi, some of the encountered difficulties arose because the rôle of auxiliary variables had not been adequately reflected in the rules.

We have followed a proposal by Apt & Meertens (1980) to interpret assertions as relations on the state space *and* a domain of auxiliary variables. Moreover, we have proposed a new consequence rule to adjust auxiliary variables while strengthening preconditions and weakening postconditions. This rule is stronger than all previously suggested structural rules, including Hoare's (1969) rule of consequence and rules of adaptation. As a direct consequence

- we were able to show that, contrary to common belief, VDM is more restrictive than Hoare Logic in that every derivation in VDM can be naturally embedded in Hoare Logic.
- We have clarified how to uniformly establish completeness as a corollary of Gorelick's (1975) Most General Formula (MGF) theorem which focusses on deriving a specific correctness formula. One may appeal to our new consequence rule to derive completeness in a single step from the MGF.
- No further structural rules are required to cater for recursive procedures.

The new approach has already been taught by Hofmann (1997) to undergraduate students as part of a course on semantics and verification.

5.3 Future Work

Following Apt (1981) and America & de Boer (1990), we have restricted our attention to a single procedure declaration without parameters. We have however proposed a new

rule to handle mutually recursive procedures which may conceivably be easier to apply than previous suggestions. Furthermore, we have sketched extensions to procedures with call-by-value parameters.

We were able to work with a simple notion of states which maps program variables to values. To extend this work to call-by-name parameters, and thus deal with aliasing, it seems advisable to consider a more elaborate mechanism for representing the state space such as an address mechanism. It would be interesting and rewarding to employ a computer-aided proof tool to investigate soundness and completeness for such verification calculi.

Appendix A

Some Remarks on Working with the LEGO System

The LEGO system was developed at the University of Edinburgh by Pollack (1994) with contributions by Claire Jones and McBride (1998). We recommend Underwood's (1997) lecture notes as an introduction to the system. LEGO implements Luo's (1994) Unifying Theory of dependent Types (UTT) which has been established as being consistent by Goguen (1994). UTT combines an extension of the Calculus of Constructions (Luo 1990) with (intensional) Martin-Löf type theory (Coquand, Nordström, Smith & von Sydow 1994). UTT has a well-established theory which consists of less than 100 rules. The algorithm which checks proof terms is relatively small. It has been programmed in SML in less than 200 lines. Thus confidence in the system is high (Pollack 1998).

In this chapter, we describe in more detail some of the peculiarities of working in type theory, with particular emphasis on the LEGO implementation. We commence with introducing the notation for term structure to clarify the outline of the scripts in the following chapter. In Sect. A.2, we introduce the system's facility for defining recursive functions. In Sect A.3, we present the encoded well-founded induction principle. Next, we explain in detail how we have encoded an equality such that one may replace terms in any type. Thus we may implement substitution on dependent functions.

A.1 Notation

Functional abstraction $\lambda x : T \cdot U$ is represented by `[x:T]U`. Its type $\prod x : T \cdot U$ is rendered as `{x:T}U`. If one replaces the colon with a vertical bar, LEGO attempts to synthesise the value of such implicit variables in an application (Pollack 1992).

Example A.1 (Implicit Arguments) *The* `seq` *constructor for sequential composition of imperative programs inhabits the type*

`{sort|SORT(VAR)}{S1,S2:prog(sort)}prog(sort)`

It expects three elements, a sort and two imperative programs where program variables adhere to this sort. It returns a program with the same sort. However, the implicit argument `sort` *of the compound program can be synthesised from the other arguments. In particular, the term* `seq(S1)(S2)`*, where* `S1` *and* `S2` *are programs of type* `prog(sort)`*, is well-formed.*

A.2 Recursion

Recursion is restricted to higher-order *primitive* recursion. Every inductively defined type is equipped with an elimination principle which enforces that the new type is closed by its constructors.

Example A.2 *Polymorphic Lists are defined by*

```
Inductive    [list:Type->Type]
Constructors [nil :{T:Type}list(T)]
             [cons:{T|Type}T->list(T)->list(T)]
```

From such a definition, LEGO generates the elimination principle

$$\frac{\prod T \cdot C\big(\mathbf{nil}(T)\big) \qquad \prod T \cdot \prod x \cdot \prod xs \cdot C(xs) \to C\big(\mathbf{cons}(x)(xs)\big)}{\prod T \cdot \prod z \cdot C(z)} \qquad (\mathtt{list}_{\mathtt{elim}})$$

and the two computation rules

$$\mathtt{list}_{\mathtt{elim}}(C)(f)(g)\big(\mathbf{nil}(T)\big) \rhd_\delta f(T)$$
$$\mathtt{list}_{\mathtt{elim}}(C)(f)(g)\big(\mathbf{cons}(x)(xs)\big) \rhd_\delta g(x)(xs)\big(\mathtt{list}_{\mathtt{elim}}(C)(f)(g)(xs)\big)$$

where C is a dependent function inhabiting $\prod T \mid \mathtt{Type} \cdot \mathtt{list}(T) \to \mathtt{Type}$. In particular, the constant $\mathtt{list}_{\mathtt{elim}}$ expects as arguments a function C and descriptions of the behaviour in the base and step cases. It returns a function mapping an arbitrary list z to $C(z)$.

Consider the definition of the **append** function in the programming language SML (Paulson 1990)

$$\begin{aligned} &\textbf{fun append}(\mathbf{nil})(y) = y \\ &\quad |\mathbf{append}(x :: xs)(y) = x :: (\mathbf{append}(xs)(y)) \end{aligned}$$

where :: denotes the uncurried version of **cons**. It follows a primitive-recursive pattern and one may immediately reflect its definition in LEGO by applying $\mathtt{list}_{\mathtt{elim}}$ to the three arguments

1. $\lambda T \cdot \lambda_ : \texttt{list}(T) \cdot \texttt{list}(T) \to \texttt{list}(T)$
2. $\lambda T \cdot \lambda y \cdot y$
3. $\lambda T \cdot \lambda x \cdot \lambda_ : \texttt{list}(T) \cdot \lambda append_xs : \texttt{list}(T) \to \texttt{list}(T) \cdot \lambda y \cdot$
   $\mathbf{cons}(x)(append_xs(y))$

A.3 Well-Founded Relations

A relation $R \subseteq T \times T$ is well-founded if, and only if, there are no infinite decreasing sequences $x_0\ R^{-1}\ x_1\ R^{-1} \cdots$ e.g., $< \subseteq \texttt{nat} \times \texttt{nat}$ is well-founded, whereas $< \subseteq \texttt{int} \times \texttt{int}$ is not. Well-founded relations play an important rôle in specifying a termination measure required in proofs of termination (Dershowitz & Manna 1979). If a verification rule refers to a well-founded relation R to cater for termination, its soundness proof has to appeal to well-founded induction:

$$\frac{\forall n \cdot \big(\forall x \cdot x\ R\ n \Rightarrow P(x)\big) \Rightarrow P(n)}{\forall z \cdot P(z)} \qquad \text{(WF)}$$

One may consider the induction scheme as a constructive version of the definition of well-foundedness.

A.4 Equality

Representing equality in type theory is a delicate subject. In this section, we show in detail how one can set up a powerful notion of extensional equality which supports dependent substitutions. Thus, we may implement updating of dependent functions which is required to represent a model of the state space.

LEGO's tactics can be configured to work for an arbitrary equality, provided it is reflexive and substitutive. It uses these ingredients to derive, amongst other properties, a proof that the equality is also symmetric. We chose Hofmann's (1995) encoding of an extensional equality. It is based on Leibniz equality which equates two terms if they fulfil the same property:

➥ p. 235 **Definition A.3 (Leibniz Equality)** $x = y \stackrel{\text{def}}{=} \forall P : T \to \texttt{Prop} \cdot P(x) \Rightarrow P(y)$

Leibniz equality is obviously reflexive and substitutive. Let $\texttt{Eq_refl}(x)$ be an abbreviation for the proof of reflexivity $\lambda P \cdot \lambda p : P(x) \cdot p$.

Substitutivity is somewhat limited. One may only replace two equal terms within a *proposition*. We therefore axiomatise[1]

[1] Notice that we cannot employ `Eq_subst` to *define* equality, because $\texttt{Eq_subst}(x, y)$ is a type and not a proposition.

➥ p. 235 **Axiom 1 (Substitutivity)**

$$\prod x, y : T \cdot x = y \to \prod C : T \to \texttt{Type} \cdot C(x) \to C(y) \qquad \text{(Eq_subst)}$$

$$\forall C : T \to \texttt{Type} \cdot \forall a : T \cdot \forall b : C(a) \cdot \texttt{Eq_subst}(a,a)(\texttt{Eq_refl}(a))(C)(b) = b \qquad \text{(Eq_subst_comp)}$$

The axiom `Eq_subst` allows one to substitute within a *type*. The second axiom permits removing redundant substitutions such as substituting a for a, provided that the equality $a = a$ has been established by the canonical proof of reflexivity. In practice, we also want to remove redundant substitutions if the equality has been established by other means, in particular, by appeal to an assumption. In such a scenario, it would help to assume that there is only one canonical proof of two terms being equal. We go further and adopt the stronger proof irrelevance principle stating that regarding any proposition, proofs are unique.

➥ p. 236 **Axiom 2 (Proof Irrelevance)** $\forall P : \texttt{Prop} \cdot \forall x, y : P \cdot x = y$

With proof irrelevance, we can strengthen substitutivity to replace terms in a dependent context:

➥ p. 237 **Lemma A.4 (Dependent Substitution)** *Let A be any type, B a type depending on elements of the type A, and C inhabiting $\prod a : A \cdot B(a) \to \texttt{Type}$. Then, for any two elements a and $\hat{a}$ in A, given a proof p that they are propositionally equal, for any* dependent *term $b : B(a)$, we may replace $C(a)(b)$ by $C(\hat{a})(\texttt{Eq_subst}(a,\hat{a})(p)(B)(b))$.*

Proof Let p be a proof of $a = \hat{a}$. We show

$$\prod b : B(a) \cdot C(\hat{a})\big(\texttt{Eq_subst}(a,\hat{a})(p)(B)(b)\big) \to C(a)(b)$$

by applying `Eq_subst` to the proof p and

$$\lambda a \cdot \prod p : a = \hat{a} \cdot \prod b : B(a) \cdot C(\hat{a})\big(\texttt{Eq_subst}(a,\hat{a})(p)(B)(b)\big) \to C(a)(b) \quad .$$

This leaves us with having to establish

$$\prod b : B(\hat{a}) \cdot C(\hat{a})\big(\texttt{Eq_subst}(\hat{a},\hat{a})(p')(B)(b)\big) \to C(\hat{a})(b)$$

where p' is a witness for $\hat{a} = \hat{a}$. Due to proof irrelevance, we may replace p' with the canonical proof $\texttt{Eq_refl}(\hat{a})$. Appealing to Axiom 1, the substitution collapses. □

Finally, we assume that two (dependent) functions are equal if their input/output behaviour is identical. This is required in the soundness proof for the verification rule for blocks on page 102 to replace a function f by $f[x \mapsto t][x \mapsto f(x)]$.

➥ p. 236 **Axiom 3 (Extensionality)** *Let T be a type and $C : T \to \texttt{Type}$ be an indexed type. Consider any dependent functions $f, g : \prod x : T \cdot C(x)$. If, for all inputs $x : T$, $f(x) = g(x)$ holds, one may conclude $f = g$.*

In his thesis, Hofmann (1995) has shown that it is consistent to assume all of these axioms.

A.5 Updating Dependent Functions

We use dependent functions to model the state space. An assignment to a variable then corresponds to updating the function. Unlike non-dependent functions, updating requires some care and we have to resort to the first two axioms to achieve the goal.

In a constructive framework, we can only effectively update a (dependent) function if equality on its domain is decidable. For the purpose of this section, let T be such a domain. We characterise that equality is decidability by relying on a concrete implementation $eq : T \times T \to \texttt{bool}$ such that the function eq reflects equality i.e.,

$$\forall x, y : T \cdot x = y \Leftrightarrow eq(x, y) = \textbf{true} \quad . \tag{DecSetoid$_{\text{char}}$}$$

Assume we have a suitable candidate for eq and let $f : \prod y : T \cdot C(y)$ be the function we want to update at position $x : T$ with $v : C(x)$ i.e. $f[x \mapsto v]$. Let y be an arbitrary element of the domain. Our task is to extract the appropriate element with type $C(y)$. We proceed by case analysis on whether x coincides with y. More precisely, we derive

$$\forall c : \texttt{bool} \cdot (eq(x, y) = c) \to C(y)$$

by induction on c with the help of the elimination principle

$$\frac{C : \texttt{bool} \to \texttt{Type} \quad C(\textbf{false}) \quad C(\textbf{true})}{\forall c : \texttt{bool} \cdot C(c)} \tag{$\texttt{bool}_{\texttt{elim}}$}$$

Consider the case that $c = \textbf{false}$. We ignore the additional input $eq(x, y) = \textbf{false}$ and retrieve the original value $f(y)$.

The case $c = \textbf{true}$ is more subtle, because the naïve solution v does not inhabit $C(y)$. But, in the presence of a proof of $eq(x, y) = \textbf{true}$, eq's characterisation lemma (DecSetoid$_{\text{char}}$) yields a proof that $x = y$ and we can use the axiom of substitutivity to coerce the type $C(y)$ to $C(x)$.

Thus, the recursion corresponds to the following elimination on booleans[2]:

$$\texttt{bool}_{\texttt{elim}}\ \big(\overbrace{\lambda c : \texttt{bool} \cdot (eq(x,y) = c)}^{\text{need to be cut}} \to C(y)\big) \tag{A.1}$$

$$\Big(\lambda H : eq(x,y) = \textbf{true} \cdot \underbrace{\texttt{Eq_subst}(x,y)\big(\text{snd}(\text{DecSetoid}_{\text{char}}(x,y))(H)\big)}_{\text{coercing } v : C(x) \text{ to } C(y)}(v)\Big) \tag{A.2}$$

$$\big(\lambda_ : eq(x,y) = \textbf{false} \cdot f(y)\big) \tag{A.3}$$

$$\big(eq(x,y)\big) \tag{A.4}$$

$$\big(\texttt{Eq_refl}(eq(x,y))\big) \tag{A.5}$$

Expressions (A.2) and (A.3) correspond to the cases $eq(x,y)$ and $\neg eq(x,y)$, respectively. The terms (A.4) and (A.5) serve as a witness in order to cut the assumptions in term (A.1)

We can now establish the correctness of this implementation and show that both $f[x \mapsto v](x) = v$ and $f[x \mapsto v](y) = f(y)$ hold where $x \neq y$. Both proofs are easy to conduct by appealing to dependent substitution, replacing $eq(x,x)$ by **true**, and $eq(x,y)$ by **false**. We focus on the first property to clarify why ordinary substitution fails. The term $f[x \mapsto v](x)$ is obtained by replacing y with x yielding

$$\texttt{bool}_{\texttt{elim}}\ \big(\overbrace{\lambda c : \texttt{bool} \cdot (eq(x,x) = c)}^{\text{need to be cut}} \to C(x)\big) \tag{A.6}$$

$$\Big(\lambda H : eq(x,x) = \textbf{true} \cdot \underbrace{\texttt{Eq_subst}(x,y)\big(\text{snd}(\text{DecSetoid}_{\text{char}}(x,x))(H)\big)}_{\text{coercing } v : C(x) \text{ to } C(x)}(v)\Big) \tag{A.7}$$

$$\big(\lambda_ : eq(x,x) = \textbf{false} \cdot f(x)\big) \tag{A.8}$$

$$\big(eq(x,x)\big) \tag{A.9}$$

$$\big(\texttt{Eq_refl}(eq(x,x))\big) \tag{A.10}$$

The substitutivity axioms are not strong enough to replace **true** for $eq(x,x)$ because, in case (A.7), the term $\texttt{Eq_subst}(x,y)\big(\text{snd}(\text{DecSetoid}_{\text{char}}(x,x))(H)\big)$ requires an argument inhabiting the *type* $(eq(x,x)) = \textbf{true}$. Replacing $eq(x,x)$ by **true** would lead to an illegal term. As a consequence, we cannot replace $eq(x,x)$ in the term (A.6) either, for otherwise, the type of the argument (A.7) would violate the typing discipline of the

[2]Notice that in *extensional* type theory (Martin-Löf 1984), we may omit the coercion function $\texttt{Eq_subst}(x,y)\big(\text{snd}(\text{DecSetoid}_{\text{char}}(x,y))(H)\big)$ because the equality reflection rule forces the identity of propositional and definitional equality. As a drawback, assuming this rule renders type-checking undecidable. We work in an intensional type theory. Type checking is decidable, but we cannot avoid equality bookkeeping in the constructed terms. Unfortunately, the current implementation of LEGO offers no help with this.

$\texttt{bool}_{\texttt{elim}}$ rule. Furthermore, if we were to replace $eq(x,x)$ with **true** in (A.9), the last argument must be a proof that $eq(x,x) = \textbf{true}$, which cannot be fulfilled by a proof of reflexivity. We need dependent substitution to perform the desired substitution for the term (A.9) modifying the type of the term (A.6)–(A.8) accordingly.

Appendix B

Outline of LEGO Scripts

This appendix contains outlines of the actual LEGO scripts in which most of the proofs have been omitted. The full scripts and further modules from LEGO's library are available on-line (Lego 1998).

B.1 Simple Imperative Programs

```
Module state Import DepUpdate;

[VAR|DecSetoid][sort|SORT VAR];

(*  Corresponds to Definition 2.2 on page 12. *)
[STATE = FS|VAR sort];
[lookupSTATE [s:STATE] = s];

(*  Corresponds to Definition 2.5 on page 14. *)
[expression [T:Type(0)] = STATE -> T]
[eval [sigma:STATE][T|Type(0)][t:expression T] = t sigma];

[STATE_update = FS_update];
[STATE_lookup_update = FS_update_lemma];
```

```
Module syntax Import lib_bool state;

(*  Corresponds to Definition 2.6 on page 15. *)
Inductive [prog:Type]
   Theorems
Constructors
  [skip:prog]
  [assign:{x:VAR.DecSetoid_carrier}{t:expression (sort x)}prog]
  [seq:{S1,S2:prog}prog]
  [ifthenelse:{b:expression bool}{S1,S2:prog}prog]
  [while:{b:expression bool}{body:prog}prog];

Goal ifloop : {T|Type}{S:prog}{then:T}{else:T}T;
...

Goal ifassign : {T|Type}{S:prog}{then:T}{else:T}T;
...

Goal prog_domain : {S:prog}Type;
...

[P : {S:prog}Prop];
```

```
Goal prog_predicate : {S:prog}Prop;
...

Discharge P;

Goal assign_update :
  {T|Type}
  {S:prog}{x:VAR.DecSetoid_carrier}{t:expression (sort x)}
  {f:{x:VAR.DecSetoid_carrier}{t:expression (sort x)}T}
  T;
...

Goal prog_assign_injective' :
   {ix0,iy0|VAR.DecSetoid_carrier}
   {ix1|expression (sort ix0)}{iy1|expression (sort iy0)}
   (Eq  (assign iy0 iy1) (assign ix0 ix1))->
   {P:{ix0:VAR.DecSetoid_carrier}{ix1:expression (sort ix0)}Prop}
   (P ix0 ix1) -> P iy0 iy1;
...


Goal loop_extract_b : {default:bool.expression}{S:prog}bool.expression;
...

Goal loop_extract_body : {S:prog}prog;
...




Module operate Import syntax;

(*  Corresponds to Definition 2.8 on page 16. *)
Inductive [prog_operate : {sigma:STATE}{S:prog}{tau:STATE}Prop] Relation
Inversion NoReductions
Constructors
  [skip_operate
    : {sigma:STATE}prog_operate sigma skip sigma]
  [assign_operate
    : {x:VAR.DecSetoid_carrier}{t:x.sort.expression}{sigma:STATE}
      prog_operate sigma (assign x t) (STATE_update x (t sigma) sigma)]
  [seq_operate
    : {S1,S2:prog}{sigma,tau,eta:STATE}
      (prog_operate sigma S1 tau) -> (prog_operate tau S2 eta)->
      (*-------------------------------------------------------*)
      prog_operate sigma (seq S1 S2) eta]
  [ifthen_operate
    : {b:bool.expression}{S1,S2:prog}{sigma,tau:STATE}
      (is_true (b (lookupSTATE sigma))) ->
      (prog_operate sigma S1 tau) ->
      prog_operate sigma (ifthenelse b S1 S2) tau]
  [ifelse_operate
    : {b:bool.expression}{S1,S2:prog}{sigma,tau:STATE}
      (is_false (b (lookupSTATE sigma))) ->
      (prog_operate sigma S2 tau) ->
      prog_operate sigma (ifthenelse b S1 S2) tau]
  [while_exit_operate
    : {b:bool.expression}{body:prog}{sigma:STATE}
      (is_false (b (lookupSTATE sigma))) ->
      prog_operate sigma (while b body) sigma]
  [while_body_operate
    : {b:bool.expression}{body:prog}{sigma,tau,eta:STATE}
      (is_true (b (lookupSTATE sigma))) ->
      (prog_operate sigma body eta) ->
      (prog_operate eta (while b body) tau) ->
      prog_operate sigma (while b body) tau];
```

```
Module operate_thms Import operate lib_bool_thms;

Goal oper_Inversion : {S:prog}{sigma,tau:STATE}Prop;

Induction S;

  Intros; Refine Eq sigma tau; (* skip *)

  (* assign *)
  Intros; Refine Eq (STATE_update x (t sigma) sigma) tau;

  (* seq *)
  Intros s1 s2 ____;
  Refine Ex [rho:STATE]and (prog_operate sigma s1 rho)
                           (prog_operate rho s2 tau);

  (* conditional *)
  Intros b s1 s2 ____;
    Refine prog_operate sigma (if (b sigma) s1 s2) tau;

  (* loop *)
  Intros b s ___;
    Refine if (b sigma);
      Refine Ex [rho:STATE]and (prog_operate sigma s rho)
                       (prog_operate rho (while b s) tau);
Refine Eq sigma tau;
Save oper_Inversion;

Goal oper_inversion : {sigma|STATE}{S|prog}{tau|STATE}
{D:prog_operate sigma S tau}(oper_Inversion S sigma tau);
...

Goal oper_inversion_ifthenelse :
  {b|expression bool}{S1,S2|prog}{sigma,tau|STATE}{c|bool}
  {eq:Eq (b sigma) c}(prog_operate sigma (ifthenelse b S1 S2) tau) ->
  prog_operate sigma (if c S1 S2) tau;
...

Goal oper_inversion_while :
  {c|bool}{b|expression bool}{S|prog}{sigma,tau|STATE}
  (Eq (b sigma) c)->(prog_operate sigma (while b S) tau) ->
  if c
     (Ex [rho:STATE]and (prog_operate sigma S rho)
                        (prog_operate rho (while b S) tau))
     (Eq sigma tau);
...

(*  Corresponds to Lemma 2.9 on page 16. *)
Goal oper_determ :
  {sigma|STATE}{S|prog}{tau|STATE}
  {Prem_tau|prog_operate sigma S tau}{C:STATE->Prop}{eta:STATE}
  {Prem_eta:prog_operate sigma S eta}
  {C_eta:C eta}C tau;
...



Module DECLexp Import lib_int_basics decsetoid;

Inductive [VARexp:Type]
Constructors [x,y,r:VARexp];

[sortexp = [_:VARexp]zed];

Goal VARexp_eq : {a,b:VARexp}bool;
...

Goal VARexp_eq_refl: {a:VARexp}is_true (VARexp_eq a a);
...
```

```
Freeze Eq;

Goal VARexp_char : Eqchar VARexp_eq;
...

Goal VARexp_setoid : DecSetoid;
...


Module exp Import DECLexp lib_bool_funs lib_int_bool operate
                        lib_nat_Le;

(** belongs in lib_nat_Le **)
Goal Le_zero_Eq_zero : {a|nat}(Le a zero) -> Eq a zero;
...

Goal Le_resp_pred_left : {a,b|nat}(Le (suc a) b) -> Le a (pred b);
...
(**************************)

(** belongs to lib_int_funs **)
Goal Eq_zed_suc : {a,b:nat}Eq_zed ((suc a,suc b)) (a,b);
...

Goal Le_zed_resp_suc_right :
  {a|zed}{b,c|nat}(Le_zed a (b,c)) -> Le_zed a (suc b,suc c);
...

Goal Le_zed_resp_pred_right :
  {a|zed}{b,c|nat}(Le_zed a (suc b,suc c)) -> Le_zed a (b,c);
...

Goal abs_zed : zed -> nat;
intros input;
Refine nat_elim [m:nat]{n:nat}nat; Refine -0 input.2; Refine -0 input.1;

  Refine Id; (* |(0,n)| = n *)
  intros m abs_zed_m;
  Induction n;
    Refine suc m; (* |(suc m,0)| = suc m *)
    intros n _;
    Refine abs_zed_m n; (* |(suc m,suc n)| = |(m,n)| *)
Save abs_zed;

(Normal abs_zed (four,two));
(Normal abs_zed (two,four));

Goal abs_zed_zero : {a|zed}(Eq (abs_zed a) zero) -> Eq_zed a zero_zed;
...

Freeze Eq;

Goal abs_zed_suc : {a|zed}(is_suc (abs_zed a)) -> not (Eq_zed a zero_zed);
...

Goal exp_zed : {a:zed}{b:nat}zed;
Induction b;
  intros; Refine one_zed;
  intros b exp_zed_b a; Refine times_zed a (exp_zed_b a);
Save exp_zed;

Configure Infix * left 6; [op* = times_zed];
Configure Infix - left 5; [op- = minus_zed];

Goal Le_zed_pos_pred :
  {a|zed}(Le_zed zero_zed a) -> (not (Eq_zed a zero_zed)) ->
  Le_zed zero_zed (a - one_zed);
...

Goal abs_zed_pos_can : {a:nat}Eq (abs_zed (a,zero)) a;
```

```
...

Goal Le_zed_Lt_zed_pred : {a,b|zed}(Le_zed a (b-one_zed)) -> Lt_zed a b;
...

Goal abs_pos_suc_pred :
  {a|zed}(Lt_zed zero_zed a) -> Eq (suc (abs_zed (a-one_zed))) (abs_zed a);
...

Goal exp_zed_pred :
  {a:zed}{b:nat}(is_suc b) -> Eq (exp_zed a b) (a * exp_zed a (pred b));
...

Goal Le_is_suc : {a,b:nat}(Le (suc a) b) -> is_suc (minus b a);
...

Goal Eq_Eq_zed : {a,b|zed}(Eq a b)->Eq_zed a b;
...

Cut [VAR=VARexp_setoid];
Cut [sort=sortexp];

Goal prog_loop_exp : prog;
Refine while [sigma:STATE]inv (int_eq_bool (sigma y) zero_zed);
    Refine seq;
      Refine assign r [sigma:STATE](sigma r) * (sigma x);
      Refine assign y [sigma:STATE]minus_zed (sigma y) one_zed;
Save prog_loop_exp;

Goal prog_exp : prog;
Refine seq;
  Refine assign r [sigma:STATE]one_zed;
  Refine prog_loop_exp;
Save prog_exp;

[sofar [sigma,tau:STATE] =
  [R=sigma r][r=tau r]
  [X=sigma x][x=tau x]
  [Y=sigma y][y=tau y]
  Eq_zed r (R * (exp_zed X (abs_zed Y)))];

Goal induction_exp :
  {n:nat}{sigma:STATE}
  {pre:(Le_zed zero_zed (sigma y)) ∧ (Eq (abs_zed (sigma y)) n)}
  Ex [tau:STATE] (prog_operate sigma prog_loop_exp tau) ∧
                  ((Eq_zed (tau y) zero_zed) ∧ sofar sigma tau);
...

(*  Corresponds to Lemma 2.10 on page 18. *)
Goal correct_exp :
  {sigma:STATE}{pre:(Le_zed zero_zed (sigma(y)))}
  Ex [tau:STATE](prog_operate sigma prog_exp tau) ∧
                (Eq_zed (tau(r)) (exp_zed (sigma x) (abs_zed ((sigma y)))));
...

Unfreeze Eq;



Module VDMsemantics Import lib_rel operate;

(*  Corresponds to Definition 2.12 on page 22. *)
[VDMSem [p:STATE.Pred][S:prog][r:Rel STATE STATE]
  = {sigma:STATE}
    (p sigma) -> Ex [tau:STATE]and (prog_operate sigma S tau) (r tau sigma)];



Module VDM Import lib_rel syntax;

[VDM_Pred_to_Rel [P:STATE.Pred] = [tau,_:STATE]P tau : Rel STATE STATE];
```

```
[reflclose [R:Rel STATE STATE] =
  [tau,sigma:STATE](R tau sigma) ∨ (Eq tau sigma)];

(*  Corresponds to Definition 2.13 on page 23. *)
Inductive
  [VDMder : {P:Pred STATE}{S:prog}{Q:Rel STATE STATE}Prop]

Relation NoReductions

Constructors
  [VDMSkip :
    VDMder  ([_:STATE]trueProp) skip  ([tau,sigma:STATE]Eq tau sigma)]

  [VDMAssign :
    {x:VAR.DecSetoid_carrier}{t:x.sort.expression}
    VDMder ([_:STATE]trueProp)
           (assign x t)
   ([tau,sigma:STATE]Eq tau (STATE_update x (t(sigma)) sigma))]

  [VDMSeq :

    {p1,p2:STATE.Pred}{r1,r2:Rel STATE STATE}{S1,S2:prog}
    (VDMder p1 S1 [tau,sigma:STATE]and (p2 tau) (r1 tau sigma)) ->
    (VDMder p2 S2 r2) ->
    VDMder p1 (seq S1 S2) (composeRel r2 r1)]

  [VDMIfthenelse :

    {pre:STATE.Pred}{post:Rel STATE STATE}
    {b:bool.expression}{TH,EL:prog}
    (VDMder ([sigma:STATE](pre sigma) ∧ (is_true (b sigma.lookupSTATE)))
            TH post) ->
    (VDMder ([sigma:STATE](pre sigma) ∧ (is_false (b sigma.lookupSTATE)))
            EL post) ->
    VDMder pre (ifthenelse b TH EL) post]

  [VDMWhile :

   {P:STATE.Pred}
   {sofar:Rel STATE STATE}
     {sofar_trans: trans sofar}
     {sofar_wf: WF (andRel sofar (VDM_Pred_to_Rel P))}
   {b:bool.expression}{body:prog}
   {Prem:VDMder
               ([sigma:STATE](P sigma) ∧ (is_true (b sigma.lookupSTATE)))
               body
        ([tau,sigma:STATE](P tau) ∧ (sofar tau sigma))}
   VDMder P (while b body)
          ([tau,sigma:STATE](P tau) ∧ (is_false (b tau.lookupSTATE)) ∧
                           (reflclose sofar tau sigma))]

  [VDMConsequence :

    {p,p1:Pred STATE}{q,q1:Rel STATE STATE}
    {S:prog}
    {ConsPre:{sigma|STATE}(p sigma) -> p1 sigma}
    {ConsPost:{sigma|STATE}{tau:STATE}(p sigma) ->
              (q1 tau sigma) -> q tau sigma}
    {ConsPrem:VDMder p1 S q1}
    VDMder p S q];



Module  VDMtheorems Import VDM;

Goal VDMPre :
  {pre|STATE.Pred}{S|prog}{post|Rel STATE STATE}
  {Prem:VDMder pre S post}
```

```
  VDMder pre S [tau,sigma:STATE]and (pre sigma) (post tau sigma);
...

Goal VDMConsequenceLeft :
            {pre_s,pre : STATE.Pred}{post:Rel STATE STATE}{S:prog}
            {strengthen : {sigma:STATE}(pre_s sigma) -> pre sigma}
            {prem : VDMder pre S post}
    VDMder pre_s S post;
...

Goal VDMConsequenceRight:

  {pre : STATE.Pred}{post,post_w:Rel STATE STATE}{S:prog}
  {weaken : {tau,sigma:STATE}(post tau sigma) -> post_w tau sigma}
  {prem : VDMder pre S post}
  VDMder pre S post_w;
...

Goal VDMSkipBU :
  {p:Rel STATE STATE}{p':STATE.Pred}
  {p'p:{sigma:STATE}(p' sigma) -> p sigma sigma}
  VDMder p' skip p;
...

Goal VDMAssignBU :
  {x:VAR.DecSetoid_carrier}{t:x.sort.expression}
  {p:Rel STATE STATE}{p':STATE.Pred}
  {p'p:{sigma:STATE}
       (p' sigma) ->  p (STATE_update x (eval sigma t) sigma) sigma}
  VDMder p' (assign x t) p;
...

Goal VDMSeqBU :
  {p1,p2:STATE.Pred}{r,r1,r2:Rel STATE STATE}{S1,S2:prog}
  {weakening: SubRel (composeRel r2 r1) r}
  (VDMder p1 S1 [tau,sigma:STATE]and (p2 tau) (r1 tau sigma)) ->
  (VDMder p2 S2 r2) ->
  VDMder p1 (seq S1 S2) r;
...

Goal VDMLoopBU :
   {P,I:STATE.Pred}
   {Q,sofar:Rel STATE STATE}
     {sofar_trans: trans sofar}
     {sofar_wf: WF (andRel sofar (VDM_Pred_to_Rel I))}
   {b:bool.expression}{body:prog}
   {ScPre:SubPred P I}
   {ScPost:{tau,sigma:STATE}(I sigma) -> (I tau) ->
           (is_false (b tau.lookupSTATE)) ->
   (reflclose sofar tau sigma) -> Q tau sigma}
   {Prem:VDMder
                ([sigma:STATE](I sigma) ∧ (is_true (b sigma.lookupSTATE)))
                body
         ([tau,sigma:STATE](I tau) ∧ (sofar tau sigma))}
   VDMder P (while b body) Q;
...



Module VDMexp Import lib_nat_Le lib_bool_funs lib_int_bool DECLexp VDM;


(** belongs in lib_nat_Le **)
Goal Le_zero_Eq_zero : {a|nat}(Le a zero) -> Eq a zero;
...

Goal Le_resp_pred_left : {a,b|nat}(Le (suc a) b) -> Le a (pred b);
...

Goal Lt_zed_Le_pred : {a,b|zed}(Lt_zed a b) -> Le_zed a (minus_zed b one_zed);
```

```
...
(**************************)

Goal Lt_imp_Le_zed : {a,b|zed}(Lt_zed a b) -> Le_zed a b;
...

[pred_zed [a:zed] = minus_zed a one_zed];

Goal pred_zed_resp : {a,b:zed}(Eq_zed a b) -> Eq_zed (pred_zed a) (pred_zed b);
...

Goal Lt_zed_pred : {a:zed}Lt_zed (pred_zed a) a;
...

(** belongs to lib_int_funs **)

Goal Eq_zed_suc : {a,b:nat}Eq_zed ((suc a,suc b)) (a,b);
...

Goal Le_zed_resp_suc_right :
  {a|zed}{b,c|nat}(Le_zed a (b,c)) -> Le_zed a (suc b,suc c);
...

Goal Le_zed_resp_pred_right :
  {a|zed}{b,c|nat}(Le_zed a (suc b,suc c)) -> Le_zed a (b,c);
...

Goal Le_zed_resp_pred_left :
  {a|zed}{b,c|nat}(Le_zed (suc b,suc c) a) -> Le_zed (b,c) a;
...

Goal Lt_zed_resp_pred_right :
  {a|zed}{b,c|nat}(Lt_zed a (suc b,suc c)) -> Lt_zed a (b,c);
...

Goal abs_zed : zed -> nat;
...

Goal abs_zed_pos_can : {a:nat}Eq (abs_zed (a,zero)) a;
...

Goal abs_zed_resp : {a,b:zed}(Eq_zed a b) -> Eq (abs_zed a) (abs_zed b);
...

Goal abs_zed_zero : {a|zed}iff (Eq (abs_zed a) zero) (Eq_zed a zero_zed);
...

Goal abs_zed_suc : {a:zed}iff  (is_suc (abs_zed a)) (not (Eq_zed a zero_zed));
...

Goal abs_zed_pred :
  {a|zed}(Lt_zed zero_zed a) ->
  Lt (abs_zed (minus_zed a one_zed)) (abs_zed a);
...

Goal Lt_resp_abs_zed_pos :
  {a,b:zed}(Le_zed zero_zed a) -> (Lt_zed a b) -> Lt (abs_zed a) (abs_zed b);
...

Goal exp_zed : {a:zed}{b:nat}zed;
...


Configure Infix * left 6; [op* = times_zed];
Configure Infix - left 5; [op- = minus_zed];

Goal Eq_Eq_zed : {a,b|zed}(Eq a b)->Eq_zed a b;
...

Goal exp_zed_times :
```

```
  {a:zed}{b,c:nat}Eq_zed (exp_zed a b * exp_zed a c) (exp_zed a (plus b c));
…

Goal Le_zed_pos_pred :
  {a|zed}(Le_zed zero_zed a) -> (not (Eq_zed a zero_zed)) ->
  Le_zed zero_zed (a - one_zed);
…


Goal Le_zed_Lt_zed_pred : {a,b|zed}(Le_zed a (b-one_zed)) -> Lt_zed a b;
…

Goal abs_pos_suc_pred :
  {a|zed}(Lt_zed zero_zed a) -> Eq (suc (abs_zed (a-one_zed))) (abs_zed a);
…

Goal abs_zed_minus :
  {a,b:zed}(Le_zed zero_zed a) -> (Le_zed a b) ->
           Eq (abs_zed (minus_zed b a)) (minus (abs_zed b) (abs_zed a));
…

Goal abs_zed_pred' :
  {a|zed}(Lt_zed zero_zed a) -> Eq (abs_zed (pred_zed a)) (pred (abs_zed a));
…

Goal exp_zed_pred :
  {a:zed}{b:nat}(is_suc b) -> Eq (exp_zed a b) (exp_zed a (pred b) * a);
…

Goal Le_is_suc : {a,b:nat}(Le (suc a) b) -> is_suc (minus b a);
…

Goal int_eq_bool_character :
  {a,b:zed}iff (is_true (int_eq_bool a b)) (Eq_zed a b);
…

Goal int_eq_bool_character' :
  {a,b:zed}iff (is_false (int_eq_bool a b)) (not (Eq_zed a b));
…

Cut [VAR=VARexp_setoid];
Cut [sort=sortexp];

Goal minus_zed_zero : {a:zed}Eq_zed (a-zero_zed) a;
…

Goal prog_loop_exp : prog;
Refine while [sigma:STATE]inv (int_eq_bool (sigma y) zero_zed);
    Refine seq;
      Refine assign r [sigma:STATE](sigma r) * (sigma x);
      Refine assign y [sigma:STATE]minus_zed (sigma y) one_zed;
Save prog_loop_exp;

Goal prog_exp : prog;
Refine seq;
  Refine assign r [sigma:STATE]one_zed;
  Refine prog_loop_exp;
Save prog_exp;

[pre = [sigma:STATE]Le_zed zero_zed (sigma(y))];

[invariant_term [sigma:STATE] =
  sigma(r) * exp_zed (sigma(x)) (abs_zed (sigma(y)))];

[sofar [tau,sigma:STATE] =
  (Eq_zed (invariant_term sigma) (invariant_term tau)) ∧
  (Lt_zed (tau(y)) (sigma(y)))];

(*  Corresponds to Lemma 2.14 on page 23. *)
Goal correct_exp :
```

```
  VDMder pre
         prog_exp
 ([tau,sigma:STATE]
  Eq_zed (tau(r)) (exp_zed (sigma x) (abs_zed ((sigma y)))));
…
```

```
Module VDMsound Import VDMsemantics VDM operate_thms;

(*  Corresponds to Theorem 2.15 on page 25. *)
Goal VDMsoundness :
  {P|Pred STATE}{S|prog}{Q|Rel STATE STATE}
  {der:VDMder P S Q}VDMSem P S Q;
…
```

```
Module termination Import operate_thms lib_rel;

(*  Corresponds to Definition 2.16 on page 27. *)
[Termin [S:prog] =
  [sigma:STATE]Ex [tau:STATE]prog_operate sigma S tau : STATE.Pred];

Goal seq_termin1 :
  {S1,S2|prog}{sigma|STATE}(Termin (seq S1 S2) sigma) -> Termin S1 sigma;
…

Goal seq_termin2 :
  {S1,S2|prog}{sigma,tau|STATE}
  (Termin (seq S1 S2) sigma) -> (prog_operate sigma S1 tau) ->
  Termin S2 tau;
…

Goal ifthenelse_termin :
  {b|expression bool}{S1,S2|prog}{sigma|STATE}{c|bool}
  {eq:Eq (b sigma) c}
  (Termin (ifthenelse b S1 S2) sigma) ->
  if c (Termin S1 sigma) (Termin S2 sigma);
…


Goal while_termin :
  {b|expression bool}{S|prog}{sigma,tau|STATE}
  (Termin (while b S) sigma) ->
  (is_true (b sigma)) ->
  (prog_operate sigma S tau) ->
  Termin (while b S) tau;
…

Goal while_termin_body :
  {b|expression bool}{S|prog}{sigma|STATE}
  (is_true (b sigma)) -> (Termin (while b S) sigma) -> Termin S sigma;
…
```

```
Module VDMcomplete Import VDMsemantics VDMtheorems termination WF;

(** pre(S,q) is equivalent to Termin(S) **)

[post [S:prog][p:STATE.Pred] =
  [tau,sigma:STATE](p(sigma)) ∧ (prog_operate sigma S tau) : Rel STATE STATE];

(*  Corresponds to Definition 2.17 on page 27. *)
[VDM_Most_General [S:prog] =
  VDMder (Termin S) S ([tau,sigma:STATE]prog_operate sigma S tau)];
```

```
Goal VDM_most_general_seq :
  {S1,S2:prog}{S1_ih:VDM_Most_General S1}{S2_ih:VDM_Most_General S2}
              (VDM_Most_General (seq S1 S2));
…

Goal VDM_most_general_conditional :
  {b:expression bool}{S1,S2:prog}
  {S1_ih:VDM_Most_General S1}{S2_ih:VDM_Most_General S2}
  VDM_Most_General (ifthenelse b S1 S2);
…

[b:expression bool][body:prog];

$Goal VDMWFsofar :
  WF (andRel (sofar b body) (VDM_Pred_to_Rel (Termin (while b body))));
…

Freeze Eq;

Goal VDM_most_general_loop :
  {ih:VDM_Most_General body}VDM_Most_General (while b body);
…

Discharge b;

(*  Corresponds to Theorem 2.18 on page 27. *)
Goal VDM_most_general : {S:prog}VDM_Most_General S;
…

(*  Corresponds to Corollary 2.19 on page 28. *)
Goal VDM_complete :
  {S|prog}{p|STATE.Pred}{q|Rel STATE STATE}
  {semant:VDMSem p S q}VDMder p S q;
…
```

```
Module semantics Import syntax assertion;

(*  Corresponds to Definition 2.20 on page 30. *)
[semantics [p:Assertion][S:prog][q:Assertion]
  = {sigma|STATE}(p sigma) ->
    Ex [tau:STATE] (prog_operate sigma S tau) ∧ (q tau)];
```

```
Module hoare Import syntax assertion lib_nat_Lt lib_rel;

(*  Corresponds to Definition 2.21 on page 31. *)
Inductive [HD : {p:Assertion}{S:prog}{q:Assertion}Prop]
Relation Inversion
Constructors

(* skip *)
[Inv : {p:Assertion}HD p skip p]

(* := *)
[Assign : {x:VAR.DecSetoid_carrier}{t:x.sort.expression}{p:Assertion}
          HD ([sigma:STATE][sigma' = STATE_update x (t sigma) sigma]
       p sigma') (assign x t) p]

(* ; *)
[Seq : {p,q,r:Assertion}{S1,S2:prog}
       (HD p S1 r) -> (HD r S2 q) -> HD p (seq S1 S2) q]

(* ifthenelse *)
[Ifthenelse : {p,q:Assertion}{b:STATE->bool}{S1,S2:prog}
              (HD ([sigma:STATE](p sigma) ∧  (is_true (b sigma))) S1 q) ->
              (HD ([sigma:STATE](p sigma) ∧ (is_false (b sigma))) S2 q) ->
```

```
              HD p (ifthenelse b S1 S2) q]

(* while *)
[While :
   {T|Type}{Lt':Rel T T}{is_wf : WF Lt'}
   {u:expression(T)} (* convergence function *)
   {p:Assertion}{b:STATE->bool}{body:prog}
   {While_step:{n:T}
     HD ([sigma:STATE](p(sigma)) ∧ (is_true(b sigma)) ∧ (Eq (u sigma) n))
        body ([tau:STATE](p(tau)) ∧ (Lt' (u tau) n))}
   HD p (while b body) ([tau:STATE](p(tau)) ∧ (is_false(b tau)))]

[Con: {p,p1,q,q1:Assertion}{S:prog}
      {ConSC : {sigma,tau:STATE}((p sigma) -> p1 sigma) ∧ ((q1 tau) -> q tau)}
      (HD p1 S q1)->HD p S q]
NoReductions;

Goal Conl : {p,p1,q:Assertion}{S:prog}
      (p .Assertion_implies p1) ->
      (HD p1 S q)->HD p S q;
...


Goal Conr:{p,q,q1:Assertion}{S:prog}
      (q1 .Assertion_implies q) ->
      (HD p S q1)->HD p S q;
...

Goal Inv_lemma : {p,p':Assertion}
                 (p' .Assertion_implies p) -> HD p' (skip) p;
...

Goal Assign_lemma :
  {x:VAR.DecSetoid_carrier}{t:x.sort.expression}{p,p':Assertion}
  (p' .Assertion_implies
   (compose p ([sigma:STATE]STATE_update x (t sigma) sigma))) ->
  HD p' (assign x t) p;
...



Module soundness Import operate semantics hoare;

(* The consequence rule preserves soundness *)
$Goal soundness_consequence :
  {p,p1,q,q1:Assertion}{S:prog}
  {ConSC : {sigma,tau:STATE}((p sigma) -> p1 sigma) ∧ ((q1 tau) -> q tau)}
  (semantics p1 S q1)->semantics p S q;
...

(*  Corresponds to Theorem 2.22 on page 32. *)
Goal soundness :
  {P:Assertion}{S:prog}{Q:Assertion}(HD P S Q) ->
  (semantics P S Q);
...



Module completeness Import operate_thms hoare semantics WF;

[pre [S:prog][q:Assertion]
  = [rho:STATE]Ex[tau:STATE]and (prog_operate rho S tau) (q tau)
  : Assertion]
[post [p:Assertion][S:prog]
  = [tau:STATE]Ex [rho:STATE](p rho) ∧ (prog_operate rho S tau)
  : Assertion];

(*  Corresponds to Definition 2.23 on page 32. *)
Goal completeness :
```

```
  {P|Assertion}{S|prog}{Q|Assertion}
  (semantics P S Q) -> HD P S Q;
...
```

```
(* Hoare Logic Assertions over extended state space *)
Module HAuxAssertion Import state;

(*  Corresponds to Definition 2.24 on page 36. *)
[Assertion [T:Type] = T -> STATE -> Prop];

[T,U|Type]

[UpdateAssertion
  [x:VAR.DecSetoid_carrier][t:expression (sort x)][p:Assertion T]
  = [z:T][sigma:STATE]
    [sigma' = STATE_update x (t sigma) sigma]p z sigma']

[ExAssertion [P:U->Assertion T]
  = [z:T][sigma:STATE]Ex ([u:U]P u z sigma)];

[KSC [p,q:Assertion T][p1,q1:Assertion U]
  = {Z|T}{sigma|STATE}(p Z sigma) ->
     Ex [Z1:U](p1 Z1 sigma ∧
 ({tau:STATE}(q1 Z1 tau) -> q Z tau))];

Discharge T;


(* Semantics of Hoare Logic *)
Module HAuxSem Import operate HAuxAssertion;

[T|Type];

(*  Corresponds to Definition 2.26 on page 36. *)
[HAuxSem [p:Assertion(T)][S:prog][q:Assertion(T)] =
  {Z|T}{sigma|STATE}(p Z sigma) ->
  Ex [tau:STATE](prog_operate sigma S tau) ∧ (q Z tau)];

Discharge T;


Module HAux Import syntax HAuxSem lib_nat_Lt lib_rel;

(*  Corresponds to Definition 2.27 on page 37. *)
Inductive [HD : {T|Type}{p:Assertion(T)}{S:prog}{q:Assertion(T)}Prop]
Relation
Constructors

(* skip *)
[Inv : {T|Type}{p:Assertion(T)}HD p skip p]

(* := *)
[Assign :
  {T|Type}{x:VAR.DecSetoid_carrier}{t:x.sort.expression}
  {p:Assertion(T)}
  HD ([Z:T][sigma:STATE][sigma' = STATE_update x (t sigma) sigma]
       p Z sigma') (assign x t) p]

(* ; *)
[Seq : {T|Type}{p,q,r:Assertion(T)}{S1,S2:prog}
       (HD p S1 r) -> (HD r S2 q) -> HD p (seq S1 S2) q]

[Ifthenelse :
  {T|Type}{p,q:Assertion(T)}{b:STATE->bool}{S1,S2:prog}
  (HD ([Z:T][sigma:STATE](p Z sigma) ∧  (is_true (b sigma))) S1 q) ->
```

```
  (HD ([Z:T][sigma:STATE](p Z sigma) ∧ (is_false (b sigma))) S2 q) ->
  HD p (ifthenelse b S1 S2) q]

(* while *)
[While :
   {T,W|Type}{Lt':Rel W W}{is_wf : WF Lt'}
   {u:expression(W)} (* convergence function *)
   {p:Assertion(T)}{b:STATE->bool}{body:prog}
   {While_step:{t:W}
     HD ([Z:T][sigma:STATE]
         (p Z sigma) ∧ (is_true(b sigma)) ∧ (Eq (u sigma) t))
        body ([Z:T][tau:STATE](p Z tau) ∧ (Lt' (u tau) t))}
   HD p (while b body) ([Z:T][tau:STATE](p Z tau) ∧ (is_false(b tau)))]

[Con: {T,U|Type}{p1,q1:Assertion(T)}{p,q:Assertion(U)}{S:prog}
      {ConSC : {Z|U}{sigma:STATE}(p Z sigma) ->
      Ex [Z1:T](p1 Z1 sigma) ∧ ({tau:STATE}(q1 Z1 tau) -> q Z tau)}
      (HD p1 S q1)->HD p S q]
NoReductions;

[T|Type];

Goal Conl :
  {p,p1,q:Assertion(T)}{S:prog}
  {SC:{Z|T}{sigma|STATE}(p Z sigma)->p1 Z sigma}
  (HD p1 S q)->HD p S q;
...


Goal Conr:
  {p,q,q1:Assertion(T)}{S:prog}
  {SC:{Z|T}{tau|STATE}(q1 Z tau)->q Z tau}
      (HD p S q1)->HD p S q;
...

Goal Inv_lemma :
  {p,p':Assertion(T)}
  {SC:{Z|T}{sigma|STATE}(p' Z sigma)->p Z sigma}HD p' (skip) p;
...

Goal Assign_lemma :
  {x:VAR.DecSetoid_carrier}{t:x.sort.expression}
  {p,p':Assertion(T)}
  {SC:{Z|T}{sigma|STATE}
    (p' Z sigma)->p Z (STATE_update x (t sigma) sigma)}
  HD p' (assign x t) p;
...

Discharge T;



Module HAuxsound Import HAuxSem HAux operate_thms;

(** Our new consequence rule preserves soundness **)
$Goal HDConsequencesoundness :
  {T|Type}{p,q:Assertion T}
  {T1|Type}{p1,q1:Assertion T1}
  {S:prog}
  {ConsSC:{z|T}{sigma|STATE}
          (p z sigma) ->
    Ex [z1:T1](p1 z1 sigma ∧ ({tau:STATE}(q1 z1 tau) -> q z tau))}
  {ConsPrem:HAuxSem p1 S q1}HAuxSem p S q;

  Expand HAuxSem;

  (* Given an auxiliary variable Z and an initial state σ
      satisfying p(Z,σ), *)
  intros __________ Z sigma pZsigma;
```

```
  (* we need to find a final state τ such that σ --S--> τ
      and q(Z,τ). Combining p(Z,σ) and ConsSC, *)
  Refine ConsSC pZsigma;

  (* we can extract a witness Z1 such that p1(Z1,σ) holds, and for
     any state τ, we may reduce the task of showing q(Z,τ) to
     q1(Z1,τ) *)

  intros Z1 Z1_details; Refine Z1_details; intros p1Z1sigma q1q;

  (* Appealing to ConsPrem, from p1(Z1,σ) *)
  Refine ConsPrem p1Z1sigma;

  (* we may infer that there is a state τ satisfying both
      σ --S--> τ and q1(Z1,τ). *)
  intros tau tau_details; Refine tau_details; intros _ q1Z1tau;
  Refine ExIn; Refine -0 pair;
  Refine -0 q1q; Immed;
$Save HDConsequencesoundness;

(*  Corresponds to Theorem 2.28 on page 38. *)
Goal Hsoundness :
  {T|Type}
  {P|Assertion T}{S|prog}{Q|Assertion T}
  {der:HD P S Q}HAuxSem P S Q;
...
```

```
Module WF Import TransClosure termination syntax;

[b:expression(bool)][body:prog];

[sofar [tau,sigma:STATE]
  = TransClosure
       ([sigma,tau:STATE](is_true (b sigma)) ∧
                   (prog_operate sigma body tau)) sigma tau];

Goal WF_sofar :
  WF ([tau,sigma:STATE]
       (Termin (while b body) sigma ∧ sofar tau sigma));
...

Discharge b;
```

```
Module HAuxComplete Import operate_thms HAux HAuxSem WF;

(*  Corresponds to Definition 2.29 on page 40. *)
[MGF [S:prog]
  = HD ([Z,sigma:STATE]prog_operate sigma S Z) S (Eq|(STATE))];

(*  Corresponds to Theorem 2.30 on page 40. *)
Goal HL_most_general : {S:prog}MGF S;
...

(*  Corresponds to Corollary 2.31 on page 41. *)
Goal HL_complete :
  {S|prog}{T|Type}{p,q|Assertion T}
  {semant:HAuxSem p S q}HD p S q;

intros -0;
Refine Con; Refine -0 HL_most_general;

Dnf; intros;
Refine semant; Immed;
intros eta eta_details; Refine eta_details; intros;
Refine ExIn;
```

```
Refine -0 pair; Immed;
qnify; Immed;
Save HL_complete;



(* Embedding VDM Correctness Formulae in Hoare Logic
   Proof-Theoretic Argument *)

Module VDMToH Import VDM HAux lib_bool_thms;

[Translate [p:Pred STATE][S:prog][q:Rel STATE STATE] =
    HD ([Z,sigma:STATE](Eq sigma Z) ∧ (p sigma)) S ([Z,tau:STATE]q tau Z)];

(*  Corresponds to Theorem 2.32 on page 43. *)
Goal VDMToH :
  {p|Pred STATE}{S|prog}{q|Rel STATE STATE}
  (VDMder p S q) -> Translate p S q;

Induction 1;

  (** Skip **)
  Refine Inv_lemma; intros; Refine H.fst;

  (** Assign **)
  intros; Refine Assign_lemma; intros;
  Qrepl H.fst; Refine Eq_refl;

  (** Sequential Composition **)
  intros; Refine Seq; Refine -0 Con; Assumption -0; Assumption +1;
  Dnf; intros; andE H; Refine ExIn; Refine +1 pair; Refine +1 pair;
    Refine +1 Eq_refl;

    Assumption;

    Intros; Refine ex_y; Immed;

  (** Conditional **)
  intros; Refine Ifthenelse;
    Refine Conl; Assumption +2; intros; andE H; andE H1;
      Refine pair; Refine -0 pair; Immed;
    Refine Conl; Assumption +2; intros; andE H; andE H1;
      Refine pair; Refine -0 pair; Immed;

  (** Loop **)
  intros;
  [invariant [Z,sigma:STATE] = (reflclose sofar sigma Z) ∧ (P sigma)];
  Refine Con; Refine -0 While; Refine +1 invariant;

    (** Resolving side-condition of consequence rule **)
    Dnf; intros; andE H; Refine ExIn; Refine -0 pair;
    Refine +1 pair; Refine +1 inr;
    Assumption +1; Assumption;
    intros; andE H3; andE H4; Refine pair; Refine pair;
    Immed;

  Refine +3 Id; Assumption +1;
  intros -0; Refine Con; Assumption -0;

    (** Resolving side-condition of consequence rule **)
    Expand Id; Expand andRel VDM_Pred_to_Rel;
     intros -0; andE H; andE H1; Qrepl Eq_sym H2;
    andE H3; Refine ExIn;
    Refine -0 pair; Refine +1 pair; Refine +2 pair;
    Assumption +1; Assumption; Assumption;
    intros; andE H7; Refine pair; Refine pair; Refine -0 pair;

    Refine H5;
      intros is_lt; Refine inl; Refine sofar_trans; Assumption +1;
      Qrepl Eq_sym H2; Immed;
```

```
      intros is_eq; Refine inl; Qrepl Eq_sym is_eq; Qrepl H2; Immed;

    Assumption;
    Qrepl H2;
    Immed;

  (** The Rule of Consequence **)
  intros; Refine Con; Assumption -0; Dnf;
  intros; andE H; Refine ExIn; Refine -0 pair; Refine +1 pair;
  Assumption +1;
  Refine  ConsPre; Assumption;
  intros _; Refine ConsPost; Qrepl Eq_sym H1; Immed;
Save VDMToH;
```

B.2 Recursive Procedures

```
Module syntax Import lib_bool state;

(****************************************)
(*** The syntax of imperative programs ***)
(****************************************)
(*  Corresponds to Definition 3.1 on page 58. *)
Inductive [prog:Type] Theorems
Constructors
  [skip:prog]
  [assign:{x:VAR.DecSetoid_carrier}{t:x.sort.expression}prog]
  [seq:{S1,S2:prog}prog]
  [ifthenelse:{b:bool.expression}{S1,S2:prog}prog]
  [while:{b:bool.expression}{body:prog}prog]
  [call:prog]; (* without parameters *)

Goal ifloop : {T|Type}{S:prog}{then:T}{else:T}T;
...

Goal loop_extract_b : {default:bool.expression}{S:prog}bool.expression;
...

Goal loop_extract_body : {S:prog}prog;
...

[P : {S:prog}Prop];

Goal prog_predicate : {S:prog}Prop;
...

Discharge P;

Goal assign_update :
  {T|Type}
  {S:prog}{x:VAR.DecSetoid_carrier}{t:expression (sort x)}
  {f:{x:VAR.DecSetoid_carrier}{t:expression (sort x)}T}
  T;
...

Goal prog_assign_injective :
   {ix0,iy0|VAR.DecSetoid_carrier}
   {ix1|expression (sort ix0)}{iy1|expression (sort iy0)}
   (Eq  (assign iy0 iy1) (assign ix0 ix1))->
   {P:{ix0:VAR.DecSetoid_carrier}{ix1:expression (sort ix0)}Prop}
   (P ix0 ix1) -> P iy0 iy1;
...



(* A recursive procedure declaration for computing the factorial function *)
```

```
Module fac Import lib_nat_bool_rels lib_nat_Lt lib_nat_times_thms syntax;

(** belongs in library **)
Goal bool_eq_refl : {x:bool}is_true (bool_eq x x);
...

Freeze Eq;

Goal bool_char : Eqchar bool_eq;
...

Goal bool_DecSetoid : DecSetoid;
Refine make_DecSetoid; Refine -0 bool_char;
Save bool_DecSetoid;


(*** (global) variable declaration ***)

Cut [VAR = bool_DecSetoid];
Cut [sort = [_:bool_DecSetoid.DecSetoid_carrier]nat];

$[x = false : VAR.DecSetoid_carrier][y = true : VAR.DecSetoid_carrier];

(*  Corresponds to Example 3.2 on page 58. *)
$Goal FactorialProcedure : prog;

Refine ifthenelse;
  Intros sigma; Refine nat_eq (sigma x) zero;
  Refine assign; Refine y; Intros; Refine one; (* then branch *)

  (** else branch **)
  Refine seq;
    Refine assign; Refine x; Intros sigma; Refine pred (sigma x);
  Refine seq;
    Refine call;
  Refine seq;
    Refine assign; Refine x; Intros sigma; Refine suc (sigma x);
    Refine assign; Refine y; Intros sigma; Refine times (sigma y) (sigma x);

Save FactorialProcedure;



Module operate Import lib_bool_thms lib_nat syntax;

(*  Corresponds to Definition 3.3 on page 59. *)
Inductive [prog_operate : {sigma:STATE}{S:prog}{tau:STATE}Prop]
Relation Inversion NoReductions
Parameters [S0:prog]

Constructors
  [skip_operate
    : {sigma:STATE}prog_operate sigma skip sigma]
  [assign_operate
    : {x:VAR.DecSetoid_carrier}{t:x.sort.expression}{sigma:STATE}
      prog_operate sigma (assign x t) (STATE_update x (t sigma) sigma)]
  [seq_operate
    : {S1,S2:prog}{sigma,tau,eta:STATE}
      (prog_operate sigma S1 tau) -> (prog_operate tau S2 eta)->
      prog_operate sigma (seq S1 S2) eta]
  [ifthen_operate
    : {b:bool.expression}{S1,S2:prog}{sigma,tau:STATE}
      (is_true (b (lookupSTATE sigma))) ->
      (prog_operate sigma S1 tau) ->
      prog_operate sigma (ifthenelse b S1 S2) tau]
  [ifelse_operate
    : {b:bool.expression}{S1,S2:prog}{sigma,tau:STATE}
      (is_false (b (lookupSTATE sigma))) ->
      (prog_operate sigma S2 tau) ->
      prog_operate sigma (ifthenelse b S1 S2) tau]
```

```
  [while_exit_operate
    : {b:bool.expression}{body:prog}{sigma:STATE}
      (is_false (b (lookupSTATE sigma))) ->
      prog_operate sigma (while b body) sigma]
  [while_body_operate
    : {b:bool.expression}{body:prog}{sigma,tau,eta:STATE}
      (is_true (b (lookupSTATE sigma))) ->
      (prog_operate sigma body tau) ->
      (prog_operate tau (while b body) eta) ->
      prog_operate sigma (while b body) eta]
  [call_operate
    : {sigma,tau:STATE}
      (prog_operate sigma S0 tau) ->
      prog_operate sigma call tau];
```

```
Module operate_thms Import lib_bool_funs operate;

Goal oper_Inversion : {S:prog}{sigma,tau:STATE}Prop;
...

Goal oper_inversion : {sigma|STATE}{S|prog}{tau|STATE}
{D:prog_operate sigma S tau}(oper_Inversion S sigma tau);
...

Goal oper_inversion_ifthenelse :
  {b|expression bool}{S1,S2|prog}{sigma,tau|STATE}{c|bool}
  {eq:Eq (b sigma) c}(prog_operate sigma (ifthenelse b S1 S2) tau) ->
  prog_operate sigma (if c S1 S2) tau;
...

Goal oper_inversion_while :
  {c|bool}{b|STATE->bool}{S|prog}{sigma,tau|STATE}
  (Eq (b sigma) c)->(prog_operate sigma (while b S) tau) ->
  if c (Ex [rho:STATE]and (prog_operate sigma S rho)
 (prog_operate rho (while b S) tau)) (Eq sigma tau);
...

Goal oper_determ :
  {sigma|STATE}{S|prog}{tau|STATE}
  {Prem_tau|prog_operate sigma S tau}{C:STATE->Prop}{eta:STATE}
  {Prem_eta:prog_operate sigma S eta}
  {C_eta:C eta}C tau;
...
```

```
(* Contexts for Hoare Logic *)
Module Hcontext Import HAuxAssertion lib_prod lib_rel;

[Spec [T:Type] = prod (Assertion T) (Assertion T)];

(*  Corresponds to Definition 3.4 on page 60. *)
Inductive [HContext : Type] Theorems
Constructors [HNone:HContext][HSome:{T|Type}{spec:Spec T}HContext];

Goal HContext_disjoint : {T|Type}{spec:Spec T}not (Eq HNone (HSome spec));
...

$Goal HContext_predicate : {P:Pred HContext}{O:HContext}Prop;
intros _;
Induction O; Refine trueProp; intros; Refine P (HSome spec);
```

```
$Save HContext_predicate;

$Goal HSome_update :
  {T|Type}{O:HContext}{B|Type}{b:Spec B}{f:{A|Type}{a:Spec A}T}T;
intros _;
Induction O;
  intros; Refine f b;
  intros; Refine f spec;
$Save HSome_update;

Goal HContext_injective :
  {A,B|Type}{a|Spec A}{b|Spec B}(Eq (HSome a) (HSome b)) ->
  {P:{T|Type}{t:Spec T}Prop}(P b) -> P a;
intros;
Claim Eq (HContext_predicate ([O:HContext]HSome_update O b P) (HSome a))
         (HContext_predicate ([O:HContext]HSome_update O b P) (HSome b));
Qrepl ?-0; Immed; Refine Eq_resp; Immed;
Save HContext_injective;



Module H Import Hcontext operate;

[HSomeSpec [T|Type][p,q:Assertion T] = HSome (Pair p q)];

(*  Corresponds to Figure 3.1 on page 63. *)
Inductive
  [HD : {T|Type}
        {Gamma:HContext}
{P:Assertion T}{S:prog}{Q:Assertion T}Prop]
Relation NoReductions

Constructors
  [HDAssumption : {T|Type}{p,q:Assertion T}HD (HSomeSpec p q) p call q]

  [HDSkip : {T|Type}{Gamma:HContext}{p:Assertion T}HD Gamma p skip p]

  [HDAssign : {T|Type}{x:VAR.DecSetoid_carrier}{t:expression (sort x)}
              {Gamma:HContext}{p:Assertion T}
              HD Gamma (UpdateAssertion x t p) (assign x t) p]

  [HDSeq : {T|Type}{Gamma|HContext}{p,r,q|Assertion T}{S1,S2:prog}
           (HD Gamma p S1 r) ->
   (HD Gamma r S2 q) ->
   HD  Gamma p (seq S1 S2) q]

  [HDIfthenelse :
    {T|Type}{Gamma|HContext}{pre,post|Assertion T}
    {b|expression bool}{TH,EL:prog}
    (HD Gamma ([z:T][sigma:STATE]and (pre z sigma)
                     (is_true (b sigma.lookupSTATE)))
        TH post) ->
    (HD Gamma ([z:T][sigma:STATE]and (pre z sigma)
     (is_false (b sigma.lookupSTATE)))
        EL post) ->
    HD Gamma pre (ifthenelse b TH EL) post]

  [HDWhile :
   {Gamma|HContext}{T,W|Type}{Lt':Rel W W}{is_wf : WF Lt'}
   {u:expression(W)} (* convergence function *)
   {p:Assertion(T)}{b:STATE->bool}{body:prog}
   {While_step:{t:W}
     HD Gamma ([Z:T][sigma:STATE]
          (p Z sigma) ∧ (is_true(b sigma)) ∧ (Eq (u sigma) t))
        body ([Z:T][tau:STATE](p Z tau) ∧ (Lt' (u tau) t))}
   HD Gamma p (while b body) ([Z:T][tau:STATE](p Z tau) ∧ (is_false(b tau)))]

  [HDCall :
    {T,W|Type}{Lt':Rel W W}{is_wf : WF Lt'}
    {p|W->Assertion T}
```

```
    {q:Assertion T}
    {Call_step:{t:W}
      HD (HSomeSpec ([Z:T][sigma:STATE]Ex [u:W](p(u)(Z)(sigma)) ∧ (Lt' u t))
      q) (p(t)) S0 q}
           HD HNone ([Z:T][sigma:STATE]Ex [t:W]p(t)(Z)(sigma)) call q]

  [HDConsequence : {Gamma:HContext}
                   {T|Type}{p,q:Assertion T}
                   {T1|Type}{p1,q1:Assertion T1}
                   {S:prog}
                   {ConsSC:KSC p q p1 q1}
                   {ConsPrem:HD Gamma p1 S q1}
                   HD Gamma p S q];
```

```
Module Htheorems Import H;

Goal HoareConsequenceLeft :
  {T|Type}{Gamma|HContext}
  {pre_s : Assertion T}{pre,post:Assertion T}{S:prog}
  {strengthen : {z:T}{sigma:STATE}(pre_s z sigma) -> pre z sigma}
  {prem : HD Gamma pre S post}
  HD Gamma pre_s S post;
...

Goal HDSkipBU :
  {T|Type}{Gamma|HContext}{p,q:Assertion T}
  ({z:T}{sigma:STATE}(p z sigma)->q z sigma) -> HD Gamma p skip q;
...

Goal HDAssignBU :
  {x:VAR.DecSetoid_carrier}{t:expression (sort x)}
  {T|Type}{Gamma|HContext}
  {p,p':Assertion T}
  {p'p:{z:T}{sigma:STATE}
       (p' z sigma) ->  (compose (p z) (STATE_update x (t sigma)) sigma)}
  HD Gamma p' (assign x t) p;
...

(*  Corresponds to Figure 3.1.8 on page 72. *)
Goal HDAdaptation :
  {T,U|Type}{Gamma|HContext}{q|Assertion T}{pre,post|Assertion U}
  {S|prog}
  {prem : HD Gamma pre S post}
  [p [z:T][sigma:STATE] =
    Ex [z':U]and (pre z' sigma)
    ({tau:STATE}(post z' tau) -> q z tau)]
  HD Gamma p S q;
...
```

```
(* Semantics of Hoare logic*)
Module Hsemantics Import operate Hcontext;

[HAtomicSem [T|Type][p:Assertion T][S:prog][q:Assertion T]
  = {z|T}{sigma|STATE}
    (p z sigma) -> Ex [tau:STATE]and (prog_operate sigma S tau) (q z tau)];

(*  Corresponds to Lemma 3.6 on page 64. *)
Goal HSemEx :
  {T,U|Type}{p:U->Assertion T}{S:prog}{q:Assertion T}
  iff (HAtomicSem ([Z:T][sigma:STATE]Ex [t:U]p(t)(Z)(sigma)) S q)
      ({t:U}HAtomicSem (p(t)) S q);
...

(*  Corresponds to Definition 3.5 on page 62. *)
```

```
Goal HSem :
  {T|Type}{Gamma:HContext}{p:Assertion T}{S:prog}{q:Assertion T}Prop;

Induction Gamma;

  Refine HAtomicSem; (* empty context *)

  intros;
  Refine (HAtomicSem spec.Fst call spec.Snd) ->
    HAtomicSem p S q;
Save HSem;



Module Hsound Import Hsemantics H operate_thms;

[Gamma:HContext][T|Type(0)]

[p,r,q:Assertion T]
[b:expression bool]
[S1,S2:prog];

Goal HDAssumptionSound : HSem (HSomeSpec p q) p call q;
...

$Goal HDInvAtomicSound : HAtomicSem p skip p;
...

Goal HDInvSound : HSem Gamma p skip p;
...

$Goal HDAssignAtomicSound :
  {x:VAR.DecSetoid_carrier}
  {t:expression (sort x)}
  HAtomicSem (UpdateAssertion x t p) (assign x t) p;
...

Goal HDAssignSound :
  {x:VAR.DecSetoid_carrier}{t:expression (sort x)}
  HSem Gamma (UpdateAssertion x t p) (assign x t) p;
...

$Goal HDSeqAtomicSound :
  {Seq_prem1:HAtomicSem p S1 r}
  {Seq_prem2:HAtomicSem r S2 q}
  HAtomicSem p (seq S1 S2) q;
...

Goal HDSeqSound :
  {Seq_prem1:HSem Gamma p S1 r}
  {Seq_prem2:HSem Gamma r S2 q}
  HSem Gamma p (seq S1 S2) q;
...

$Goal HDIfthenelseAtomicSound :
  {Ifthenelse_prem1:
      HAtomicSem ([Z:T][sigma:STATE ]
   (p Z sigma) ∧ (is_true (eval sigma b)))
 S1 q}
  {Ifthenelse_prem2:
    HAtomicSem ([Z:T][sigma:STATE ]
      (p Z sigma) ∧ (is_false (eval sigma b)))
                          S2 q}
    HAtomicSem p (ifthenelse b S1 S2) q;
...

Goal HDIfthenelseSound :
  {Ifthenelse_prem1:
      HSem Gamma ([Z:T][sigma:STATE ]
   (p Z sigma) ∧ (is_true (eval sigma b)))
```

```
 S1 q}
  {Ifthenelse_prem2:
    HSem Gamma ([Z:T][sigma:STATE ]
      (p Z sigma) ∧ (is_false (eval sigma b)))
                          S2 q}
    HSem Gamma p (ifthenelse b S1 S2) q;
…

$Goal HDWhileAtomicSound :
  {W|Type}{Lt':Rel W W}{is_wf : WF Lt'}
  {u:(STATE )->W}
  {body:prog}
  {While_step:{t:W}
      HAtomicSem ([Z:T][sigma:STATE ]
   (p Z sigma) ∧ (is_true(eval(sigma)(b))) ∧
   (Eq (u sigma) t))
      body ([Z:T][tau:STATE ](p Z tau) ∧ (Lt' (u tau) t))}
   HAtomicSem p (while b body)
    ([Z:T][tau:STATE ](p Z tau) ∧ (is_false(eval tau b)));
…

Goal HDWhileSound :
{W|Type}{Lt':Rel W W}{is_wf : WF Lt'}
  {u:(STATE )->W}
  {body:prog}
  {While_step:{t:W}
      HSem Gamma ([Z:T][sigma:STATE ]
   (p Z sigma) ∧ (is_true(eval(sigma)(b))) ∧
   (Eq (u sigma) t))
      body ([Z:T][tau:STATE ](p Z tau) ∧ (Lt' (u tau) t))}
   HSem Gamma p (while b body)
    ([Z:T][tau:STATE ](p Z tau) ∧ (is_false(eval tau b)));
…

(** Our new consequence rule preserves soundness **)
$Goal HDConsequenceAtomicsoundness :
  {T|Type}{p,q:Assertion T}
  {T1|Type}{p1,q1:Assertion T1}
  {S:prog}
  {ConsSC:KSC p q p1 q1}
  {ConsPrem:HAtomicSem p1 S q1}HAtomicSem p S q;
…

Goal HDConSound :
  {U|Type}{pre,post:Assertion U}
  {S:prog}
  {Prem : KSC p q pre post}
  {prem : HSem Gamma pre S post}
    HSem Gamma p S q;
…

Goal HDCallSound :
  {W|Type}{Lt':Rel W W}{is_wf : WF Lt'}
    {p|W->Assertion T}
    {Call_step:{t:W}
      HSem (HSomeSpec
          ([Z:T][sigma:STATE]
    Ex [u:W](p(u)(Z)(sigma)) ∧ (Lt' u t))
   q) (p(t)) S0 q}
   HSem HNone
      ([Z:T][sigma:STATE]Ex [t:W]p(t)(Z)(sigma))
      call q;
…


Discharge Gamma;

(*  Corresponds to Theorem 3.7 on page 65. *)
Goal Hsoundness :
  {T|Type}
```

```
  {Gamma|HContext}
  {P|Assertion T}{S|prog}{Q|Assertion T}
  {der:HD Gamma P S Q}
  HSem Gamma P S Q;
...
```

```
Module boperate Import operate;

(*  Corresponds to Definition 3.9 on page 66. *)
Inductive [prog_boperate : {sigma:STATE}{S:prog}{tau:STATE}{n:nat}Prop]
Relation NoReductions Inversion

Constructors
  [skip_boperate
    : {sigma:STATE}{n:nat}prog_boperate sigma skip sigma n]
  [assign_boperate
    : {x:VAR.DecSetoid_carrier}{t:x.sort.expression}
      {sigma:STATE}{n:nat}
      prog_boperate sigma (assign x t) (STATE_update x (t sigma) sigma) n]
  [seq_boperate
    : {S1,S2:prog}{sigma,tau,eta:STATE}{n:nat}
      (prog_boperate sigma S1 tau n) -> (prog_boperate tau S2 eta n)->
      prog_boperate sigma (seq S1 S2) eta n]
  [ifthen_boperate
    : {b:bool.expression}{S1,S2:prog}{sigma,tau:STATE}{n:nat}
      (is_true (b (lookupSTATE sigma))) ->
      (prog_boperate sigma S1 tau n) ->
      prog_boperate sigma (ifthenelse b S1 S2) tau n]
  [ifelse_boperate
    : {b:bool.expression}{S1,S2:prog}{sigma,tau:STATE}{n:nat}
      (is_false (b (lookupSTATE sigma))) ->
      (prog_boperate sigma S2 tau n) ->
      prog_boperate sigma (ifthenelse b S1 S2) tau n]
  [while_exit_boperate
    : {b:bool.expression}{body:prog}{sigma:STATE}{n:nat}
      (is_false (b (lookupSTATE sigma))) ->
      prog_boperate sigma (while b body) sigma n]
  [while_body_boperate
    : {b:bool.expression}{body:prog}{sigma,tau,eta:STATE}{n:nat}
      (is_true (b (lookupSTATE sigma))) ->
      (prog_boperate sigma body tau n) ->
      (prog_boperate tau (while b body) eta n) ->
      prog_boperate sigma (while b body) eta n]
  [call_boperate
    : {sigma,tau:STATE}{n:nat}
      (prog_boperate sigma S0 tau n) ->
      prog_boperate sigma call tau (suc n)];
```

```
Module boperate_thms
  Import lib_bool_funs boperate lib_nat_Prop_rels lib_max_min;

Goal boper_Inversion : {S:prog}{sigma,tau:STATE}{n:nat}Prop;
...

Goal boper_inversion : {sigma|STATE}{S|prog}{tau|STATE}{n|nat}
{D:prog_boperate sigma S tau n}(boper_Inversion S sigma tau n);
...

Goal boper_ifthen : {b|STATE->bool}{S1,S2|prog}{sigma,tau|STATE}{n|nat}
(is_true (b sigma))->
(prog_boperate sigma (ifthenelse b S1 S2) tau n) ->
```

```
prog_boperate sigma S1 tau n;
...

Goal boper_ifelse : {b|STATE->bool}{S1,S2|prog}{sigma,tau|STATE}{n|nat}
(is_false (b sigma)) ->
(prog_boperate sigma (ifthenelse b S1 S2) tau n) ->
prog_boperate sigma S2 tau n;
...

Goal boper_while : {c|bool}{b|STATE->bool}{S|prog}{sigma,tau|STATE}{n|nat}
(Eq (b sigma) c)->(prog_boperate sigma (while b S) tau n) ->
        (and (is_false (b tau))
     (if c (Ex [rho:STATE]
            and (prog_boperate sigma S rho n)
                (prog_boperate rho (while b S) tau n))
                           (Eq sigma tau)));
...

Goal boper_while_false :
  {b|STATE->bool}{S|prog}{sigma,tau|STATE}{n|nat}
  (prog_boperate sigma (while b S) tau n) -> is_false (b tau);
...

Goal boper_determ :
  {sigma|STATE}{S|prog}{tau|STATE}{n|nat}
  {Prem_tau|prog_boperate sigma S tau n}{C:STATE->Prop}{eta:STATE}
  {m|nat}{Prem_eta:prog_boperate sigma S eta m}
  {C_eta:C eta}C tau;
...

(*  Corresponds to Lemma 3.10 on page 66. *)
Goal boperate_not_strict :
  {sigma|STATE}{S|prog}{tau|STATE}{n|nat}
  (prog_boperate sigma S tau n) ->
  {m:nat}(Le n m) -> prog_boperate sigma S tau m;
...

(*  Corresponds to Lemma 3.11 on page 67. *)
Goal operate2boperate :
  {sigma|STATE}{S|prog}{tau|STATE}
  {Prem:prog_operate sigma S tau}
  Ex [n:nat]prog_boperate sigma S tau n;
...

Goal boperate2operate :
  {sigma|STATE}{S|prog}{tau|STATE}{n|nat}
  {Prem:prog_boperate sigma S tau n}
        prog_operate sigma S tau;
...

Freeze Eq;

Goal boperate_zero :
  {sigma,tau:STATE}not (prog_boperate sigma call tau zero);
...



(* Completeness of Hoare logic *)
Module Hcomplete Import boperate_thms operate_thms
     Htheorems Hsemantics WF;

$[Hoarep0 [S:prog][n:nat] =
  [Z,sigma:STATE]prog_boperate sigma S Z n];

$[psi [n:nat] = Hoarep0 call (suc n)];

[q [Z,tau:STATE] = Eq Z tau];

Freeze Eq;
```

```
(*  Corresponds to Lemma 3.12 on page 67. *)
Goal Hoare_completeness_step :
     {n:nat}
     HD (HSomeSpec ([Z,sigma:STATE]
                    Ex ([u:nat](psi u Z sigma ∧ Lt u n)))
   q)
         (psi n) S0 q;
...

(*  Corresponds to Theorem 3.8 on page 65. *)
Goal HL_most_general :
   {S:prog}
    HD HNone  ([z:STATE][sigma:STATE]prog_operate sigma S z)
             S q;
...

(*  Corresponds to Corollary 3.13 on page 69. *)
Goal HL_complete :
  {S|prog}{T|Type}{p|Rel T STATE}{q|Rel T STATE}
  {semant:HSem HNone p S q}HD HNone p S q;
...


(* Proof in Hoare logic that our factorial procedure leaves x invariant*)
Module Hfac Import fac Htheorems;

Cut [S0 = FactorialProcedure];

[HoareFactorialInvariantAssertion [z:nat][sigma:STATE] = Eq (sigma x) z]

[HoareFactorialInvariantCF
  = HD HNone
       HoareFactorialInvariantAssertion call HoareFactorialInvariantAssertion];

(*  Corresponds to Lemma 3.14 on page 69. *)
Goal HoareFactorialInvariantCorrect : HoareFactorialInvariantCF;
...


Module VDMcontext Import lib_prod lib_rel state;

[VDMSpec = prod (Pred STATE) (Rel STATE STATE)];

Inductive [VDMContext : Type] Theorems
Constructors [VDMNone:VDMContext][VDMSome:{spec:VDMSpec}VDMContext];

[VDMSomeSpec [p:Pred STATE][q:Rel STATE STATE]
  = VDMSome (Pair p q)];


Module VDMsemantics Import VDMcontext operate;

[VDMAtomicSem
  [p:Pred STATE][S:prog][q:Rel STATE STATE ]
  = {sigma|STATE}(p sigma) ->
  Ex [tau:STATE]and (prog_operate sigma S tau) (q tau sigma)];

Goal VDMSemEx :
  {U|Type}
  {p:U->Pred STATE}
  {S:prog}
  {q:Rel STATE STATE}
  iff (VDMAtomicSem ([sigma:STATE]
    Ex [t:U]p(t)(sigma)) S q)
      ({t:U}VDMAtomicSem (p(t)) S q);
...

Goal VDMSem :
```

```
  {Gamma:VDMContext}
  {p:Pred STATE}
  {S:prog}
  {q:Rel STATE STATE}
  Prop;
...
```

```
Module VDM Import tms_rel VDMcontext operate;

[VDM_Pred_to_Rel [P:Pred STATE]
  = [tau,_:STATE]P tau : Rel STATE STATE];

(*  Corresponds to Figure 3.4 on page 76. *)
Inductive
  [VDMder :
    {Gamma:VDMContext}{P:Pred STATE}{S:prog}{Q:Rel STATE STATE}Prop]

Relation NoReductions

Constructors
  [VDMAssumption :
    {p:Pred STATE}{q:Rel STATE STATE}
    VDMder (VDMSome (Pair p q)) p call q]

  [VDMSkip : {Gamma:VDMContext}
             VDMder Gamma ([_:STATE]trueProp) skip (Eq|STATE)]

  [VDMAssign :
    {Gamma:VDMContext}
    {x:VAR.DecSetoid_carrier}{t:expression (sort x)}
    VDMder Gamma ([_:STATE]trueProp) (assign x t)
    ([tau,sigma:STATE]
      Eq tau (STATE_update x (eval sigma t) sigma))]

  [VDMSeq :
    {Gamma:VDMContext}
    {p1,p2:STATE.Pred}{r1,r2:Rel STATE STATE}{S1,S2:prog}
    (VDMder Gamma p1 S1 [tau,sigma:STATE]and (p2 tau) (r1 tau sigma)) ->
    (VDMder Gamma p2 S2 r2) ->
    VDMder Gamma p1 (seq S1 S2) (composeRel r2 r1)]

  [VDMIfthenelse :
    {Gamma:VDMContext}
    {pre:STATE.Pred}{post:Rel STATE STATE}
    {b:bool.expression}{TH,EL:prog}
    (VDMder Gamma ([sigma:STATE](pre sigma) ∧ (is_true (b sigma.lookupSTATE)))
            TH post) ->
    (VDMder Gamma ([sigma:STATE](pre sigma) ∧ (is_false (b sigma.lookupSTATE)))
            EL post) ->
    VDMder Gamma pre (ifthenelse b TH EL) post]

  [VDMWhile :
    {Gamma:VDMContext}
    {P:STATE.Pred}
    {sofar:Rel STATE STATE}
    {sofar_trans: trans sofar}
    {sofar_wf: WF (andRel sofar (VDM_Pred_to_Rel P))}
    {b:bool.expression}{body:prog}
    {Prem:VDMder Gamma
                ([sigma:STATE](P sigma) ∧ (is_true (b sigma.lookupSTATE)))
                body
        ([tau,sigma:STATE](P tau) ∧ (sofar tau sigma))}
   VDMder Gamma P (while b body)
          ([tau,sigma:STATE](P tau) ∧ (is_false (b tau.lookupSTATE)) ∧
    (reflclose sofar tau sigma))]

  [VDMCall :
    {W|Type}{Lt':Rel W W}{is_wf : WF Lt'}
```

```
    {p:W->Pred STATE}
    {q:Rel STATE STATE}
    {Call_step:
      {t:W} VDMder (VDMSomeSpec ([sigma:STATE]
  Ex [u:W](p(u)(sigma)) /\ (Lt' u t))
   q)
      (p(t)) S0 q}
   VDMder VDMNone ([sigma:STATE]Ex [t:W]p(t)(sigma)) call q]

  [VDMConsequence :
    {Gamma:VDMContext}
    {p,p1:Pred STATE}{q,q1:Rel STATE STATE}
    {S:prog}
    {ConsSC1:{sigma|STATE}(p sigma) -> (p1 sigma)}
    {ConsSC2:{tau,sigma|STATE}(p sigma) ->
      (q1 tau sigma) -> q tau sigma}
    {ConsPrem:VDMder Gamma p1 S q1}
    VDMder Gamma p S q];
```

```
Module VDMtheorems Import VDM;

Goal VDMPre :
  {Gamma:VDMContext}{pre|STATE.Pred}{S|prog}{post|Rel STATE STATE}
  {Prem:VDMder Gamma pre S post}
  VDMder Gamma pre S [tau,sigma:STATE]and (pre sigma) (post tau sigma);
...

Goal VDMConsequenceLeft :
            {Gamma:VDMContext}
            {pre_s,pre : STATE.Pred}{post:Rel STATE STATE}{S:prog}
            {strengthen : {sigma:STATE}(pre_s sigma) -> pre sigma}
            {prem : VDMder Gamma pre S post}
    VDMder Gamma pre_s S post;
...

Goal VDMConsequenceRight:
  {Gamma:VDMContext}
  {pre : STATE.Pred}{post,post_w:Rel STATE STATE}{S:prog}
  {weaken : {sigma,tau:STATE}(post tau sigma) -> post_w tau sigma}
  {prem : VDMder Gamma pre S post}
  VDMder Gamma pre S post_w;
...

Freeze Eq;

Goal VDMSkipBU :
  {Gamma:VDMContext}
  {p:Rel STATE STATE}{p':STATE.Pred}
  {p'p:{sigma:STATE}(p' sigma) -> p sigma sigma}
  VDMder Gamma p' skip p;
...

Goal VDMAssignBU :
  {Gamma:VDMContext}
  {x:VAR.DecSetoid_carrier}{t:expression (sort x)}
  {p:Rel STATE STATE}{p':STATE.Pred}
  {p'p:
    {tau:STATE}(p' tau) ->
    p (STATE_update x (eval tau t) tau) tau}
  VDMder Gamma p' (assign x t) p;
...

Goal VDMSeqBU :
  {Gamma:VDMContext}
  {p1,p2:STATE.Pred}{r,r1,r2:Rel STATE STATE}{S1,S2:prog}
  {weakening: SubRel (composeRel r2 r1) r}
  (VDMder Gamma p1 S1 [tau,sigma:STATE]and (p2 tau) (r1 tau sigma)) ->
```

```
  (VDMder Gamma p2 S2 r2) ->
  VDMder Gamma p1 (seq S1 S2) r;
…


Goal VDMLoopBU :
   {Gamma:VDMContext}
   {P,I:STATE.Pred}
   {Q,sofar:Rel STATE STATE}
     {sofar_trans: trans sofar}
     {sofar_wf: WF (andRel sofar (VDM_Pred_to_Rel I))}
   {b:expression bool}{body:prog}
   {ScPre:SubPred P I}
   {ScPost:{tau,sigma:STATE}(I sigma) -> (I tau) ->
           (is_false (eval tau b)) ->
   (reflclose sofar tau sigma) -> Q tau sigma}
   {Prem:
     VDMder Gamma ([sigma:STATE](I sigma) ∧
      (is_true (eval sigma b)))
                body
        ([tau,sigma:STATE](I tau) ∧ (sofar tau sigma))}
   VDMder Gamma P (while b body) Q;
…
```

```
Module VDMsound Import VDMsemantics VDM operate_thms;

[p:Pred STATE][q:Rel STATE STATE]
[b:expression bool]
[S1,S2:prog];

Goal VDMAssumptionSound : VDMSem (VDMSomeSpec p q) p call q;
…

$Goal VDMInvAtomicSound :
  VDMAtomicSem  ([_:STATE]trueProp) skip (Eq|STATE);
…

[Gamma:VDMContext];

Goal VDMInvSound :
  VDMSem Gamma ([_:STATE]trueProp) skip (Eq|STATE);
…

$Goal VDMAssignAtomicSound :
  {x:VAR.DecSetoid_carrier}
  {t:expression (sort x)}
  VDMAtomicSem  ([_:STATE]trueProp) (assign x t)
      ([tau,sigma:STATE]
Eq tau (STATE_update x (eval sigma t) sigma));
…

Goal VDMAssignSound :
  {x:VAR.DecSetoid_carrier}
  {t:expression (sort x)}
  VDMSem Gamma ([_:STATE]trueProp) (assign x t)
      ([tau,sigma:STATE]
Eq tau (STATE_update x (eval sigma t) sigma));
…

$Goal VDMSeqAtomicSound :
  {p1,p2:STATE.Pred}
  {r1,r2:Rel STATE STATE}
  {Seq_prem1:VDMAtomicSem  p1 S1 ([tau,sigma:STATE]
  and (p2 tau) (r1 tau sigma))}
  {Seq_prem2:VDMAtomicSem  p2 S2 r2}
  VDMAtomicSem  p1 (seq S1 S2) (composeRel r2 r1);
…
```

```
Goal VDMSeqSound :
  {p1,p2:STATE.Pred}
  {r1,r2:Rel STATE STATE}
  {Seq_prem1:VDMSem Gamma p1 S1 ([tau,sigma:STATE]
  and (p2 tau) (r1 tau sigma))}
  {Seq_prem2:VDMSem Gamma p2 S2 r2}
  VDMSem Gamma p1 (seq S1 S2) (composeRel r2 r1);
…

$Goal VDMIfthenelseAtomicSound :
  {pre:STATE.Pred}
  {post:Rel STATE STATE}
  {TH,EL:prog}
  {Ifthenelse_prem1:
    VDMAtomicSem  ([sigma:STATE]
(pre sigma) ∧ (is_true (eval sigma b)))
    TH post}
  {Ifthenelse_prem2:
    VDMAtomicSem  ([sigma:STATE]
(pre sigma) ∧ (is_false (eval sigma b)))
    EL post}
  VDMAtomicSem  pre (ifthenelse b TH EL) post;
…

Goal VDMIfthenelseSound :
  {pre:STATE.Pred}
  {post:Rel STATE STATE}
  {TH,EL:prog}
  {Ifthenelse_prem1:
      VDMSem Gamma ([sigma:STATE]
(pre sigma) ∧ (is_true (eval sigma b)))
    TH post}
  {Ifthenelse_prem2:
    VDMSem Gamma ([sigma:STATE]
(pre sigma) ∧ (is_false (eval sigma b)))
    EL post}
    VDMSem Gamma pre (ifthenelse b TH EL) post;
…

$Goal VDMWhileAtomicSound :
  {P:STATE.Pred}
    {sofar:Rel STATE STATE}
    {sofar_trans: trans sofar}
    {sofar_wf: WF (andRel sofar (VDM_Pred_to_Rel P))}
    {body:prog}
    {Prem:
      VDMAtomicSem  ([sigma:STATE]
       (P sigma) ∧ (is_true (eval sigma b)))
                body
        ([tau,sigma:STATE]
  (P tau) ∧ (sofar tau sigma))}
   VDMAtomicSem  P (while b body)
          ([tau,sigma:STATE]
    (P tau) ∧ (is_false (eval tau b)) ∧
                           (reflclose sofar tau sigma));
…

Goal VDMWhileSound :
  {P:STATE.Pred}
    {sofar:Rel STATE STATE}
    {sofar_trans: trans sofar}
    {sofar_wf: WF (andRel sofar (VDM_Pred_to_Rel P))}
    {body:prog}
    {Prem:
      VDMSem Gamma ([sigma:STATE]
       (P sigma) ∧ (is_true (eval sigma b)))
                body
        ([tau,sigma:STATE]
  (P tau) ∧ (sofar tau sigma))}
```

```
   VDMSem Gamma P (while b body)
          ([tau,sigma:STATE]
    (P tau) ∧ (is_false (eval tau b)) ∧
    (reflclose sofar tau sigma));
…

Goal VDMCallSound :
  {W|Type}{Lt':Rel W W}{is_wf : WF Lt'}
    {p:W->Pred STATE}
    {q:Rel STATE STATE}
    {Call_step:{t:W}
      VDMSem (VDMSomeSpec
          ([sigma:STATE]
     Ex [u:W](p(u)(sigma)) ∧ (Lt' u t))
    q)
       (p(t)) S0 q}
   VDMSem VDMNone
       ([sigma:STATE]Ex [t:W]p(t)(sigma))
       (call) q;
…

$Goal VDMConAtomicSound :
  {p1:Pred STATE}
  {q1:Rel STATE STATE}
  {S:prog}
  {ConsSC1:{sigma|STATE}(p sigma) -> (p1 sigma)}
  {ConsSC2:{tau,sigma|STATE}(p sigma) ->
    (q1 tau sigma) -> q tau sigma}
  {ConsPrem:VDMAtomicSem  p1 S q1}
  VDMAtomicSem  p S q;
…

Goal VDMConSound :
  {p1:Pred STATE}
  {q1:Rel STATE STATE}
  {S:prog}
  {ConsSC1:{sigma|STATE}(p sigma) -> (p1 sigma)}
  {ConsSC2:{tau,sigma|STATE}(p sigma) ->
    (q1 tau sigma) -> q tau sigma}
  {ConsPrem:VDMSem Gamma p1 S q1}
  VDMSem Gamma p S q;
…

Discharge p;

(*  Corresponds to Theorem 3.17 on page 75. *)
Goal VDMsoundness :
  {Gamma|VDMContext}
  {P|Pred STATE}{S|prog}{Q|Rel STATE STATE}
  {der:VDMder Gamma P S Q}VDMSem Gamma P S Q;
…
```

Module **VDMcomplete**

```
  Import VDMsemantics VDMtheorems termination WF boperate_thms;

[VDMp0 [S:prog][n:nat][sigma:STATE] =
     Ex [tau:STATE]prog_boperate sigma S tau n];


[psi [n:nat] = VDMp0 call (suc n)]

[p [n:nat][sigma:STATE] = Ex ([u:nat](psi u sigma ∧ Lt u n))];

Goal VDMp02Termin :
  {S:prog}{sigma:STATE}{n:nat}(VDMp0 S n sigma) -> Termin S sigma;
…

[b:expression bool][body:prog][n:nat];
```

```
Goal VDMWFsofar :
  WF (andRel (sofar b body)
       (VDM_Pred_to_Rel (Termin (while b body))));
...

Goal VDMWFbsofar :
  WF (andRel (sofar b body)
       (VDM_Pred_to_Rel (VDMp0 (while b body) n)));
...

Discharge b;

Goal VDM_completeness_step :
  {n:nat}{S:prog}
  VDMder (VDMSome (Pair (p n)
                        ([tau,sigma:STATE]prog_operate sigma call tau)))
       (VDMp0 S n) S ([tau,sigma:STATE]prog_operate sigma S tau);
...


Goal VDM_most_general :
   {S:prog}
    VDMder VDMNone ([sigma:STATE]Ex [tau:STATE]prog_operate sigma S tau)
             S
     ([tau,sigma:STATE]prog_operate sigma S tau);
...

(*  Corresponds to Theorem 3.18 on page 75. *)
Goal VDM_complete :
  {S|prog}{p|STATE.Pred}{q|Rel STATE STATE}
  {semant:VDMSem VDMNone p S q}VDMder VDMNone p S q;
...
```

B.3 Local Variables

```
Module state Import DepUpdate;

[VAR|DecSetoid]
[sort|SORT|VAR];

(*  Corresponds to Definition 3.20 on page 79. *)
[STATE [sort:SORT|VAR] = FS sort];
[lookup [sigma:STATE sort] = sigma];
[STATE_update = FS_update]
[STATE_lookup_update = FS_lookup_update]
[STATE_update_dyn = FS_update_dyn]
[STATE_update_dyn' = FS_update_dyn']
[STATE_update_dyn'_lookup = FS_update_dyn'_lookup]
[STATE_lookup_update_dyn = FS_lookup_update_dyn];
[STATE_update_dyn_dyn' = FS_update_dyn_dyn'];

(*  Corresponds to Definition 3.21 on page 79. *)
[expression [sort:SORT|VAR][T:Type] = (STATE sort) -> T];
[eval [T|Type][sigma:STATE sort][t:expression sort T] = t sigma];

Discharge VAR;

[STATE_update_dyn'_update_dyn
  = FS_update_dyn'_update_dyn
  : {VAR|DecSetoid}{sort|SORT|VAR}
    {y:VAR.DecSetoid_carrier}{t:sort(y)}{U|Type(0)}
    {u:U}{sigma:STATE sort}
    Eq (STATE_update_dyn' y t (STATE_update_dyn y u sigma))
       (STATE_update y t sigma)];
```

```
[STATE_update_dyn_lookup
  = FS_update_dyn_lookup
  : {VAR|DecSetoid}{sort|SORT|VAR}
    {y:VAR.DecSetoid_carrier}{U|Type(0)}
    {v:U}{sigma:STATE sort}
    Eq (STATE_update_dyn y v sigma y)
       (coerce (Eq_sym (sort_update_lookup y U sort)) v)];



Module assertion Import state;

[VAR|DecSetoid];

(*  Corresponds to Definition 3.22 on page 79. *)
[Assertion [T:Type][vardecl:VAR.DecSetoid_carrier->Type]
  = T -> (STATE vardecl) -> Prop]

[vardecl|VAR.DecSetoid_carrier->Type][T,U|Type]

[UpdateAssertion
  [x:VAR.DecSetoid_carrier][t:expression vardecl (vardecl x)]
  [p:Assertion T vardecl]
  = [z:T][sigma:STATE vardecl]
    [sigma' = STATE_update x (t sigma) sigma]p z sigma']

[ExAssertion [P:U->Assertion T vardecl]
  = [z:T][sigma:STATE vardecl]Ex ([u:U]P u z sigma)];

[KSC [p,q:Assertion T vardecl][p1,q1:Assertion U vardecl]
  = {z|T}{sigma|STATE vardecl}(p z sigma) ->
     Ex [z1:U](p1 z1 sigma ∧
      ({tau:STATE vardecl}(q1 z1 tau) -> q z tau))];

Discharge VAR;



Module syntax Import lib_bool state;

[VAR|DecSetoid];

(*  Corresponds to Definition 3.23 on page 80. *)
Inductive [prog:{sort:SORT(VAR)}Type]

Constructors

  [skip : {sort:SORT(VAR)}prog sort]

  [assign
    : {sort|SORT(VAR)}
      {x:VAR.DecSetoid_carrier}{t:expression sort (sort x)}prog sort]

  [seq : {sort|SORT(VAR)}{S1,S2:prog(sort)}prog(sort)]

  [ifthenelse
    : {sort|SORT(VAR)}
      {b:expression sort bool}{S1,S2:prog sort}prog sort]

  [while
    : {sort|SORT(VAR)}
      {b:expression sort bool}{body:prog sort}prog sort]

  [new
    : {sort|SORT(VAR)}{T|Type}
       {x:VAR.DecSetoid_carrier}{t:expression sort T}
       {S:prog (sort_update x T sort)}prog sort];

Goal ifloop
```

```
  : {T|Type}{sort|SORT(VAR)}
    {S:prog sort}{then:T}{else:T}T;
...

Goal ifassign
  : {T|Type}{sort|SORT(VAR)}
    {S:prog sort}{then:T}{else:T}T;
...

Goal prog_domain
  : {sort|SORT(VAR)}{S:prog sort}Type;
...

[P : {sort|SORT(VAR)}{S:prog sort}Prop];

Goal prog_predicate
  : {sort|SORT(VAR)}{S:prog sort}Prop;
...

Discharge P;

Goal assign_update :
  {T|Type}{sort|SORT(VAR)}
  {S:prog sort}{y:VAR.DecSetoid_carrier}{sort|SORT(VAR)}{u:expression sort (sort y)}
  {f:{sort|SORT(VAR)}
     {x:VAR.DecSetoid_carrier}{t:expression sort (sort x)}T}
  T;
...

Goal prog_assign_injective' :
   {sort|SORT(VAR)}{ix0,iy0|VAR.DecSetoid_carrier}
   {ix1|expression sort (sort ix0)}{iy1|expression sort (sort iy0)}
   (Eq  (assign iy0 iy1) (assign ix0 ix1))->
   {P:{sort|SORT(VAR)}
      {ix0:VAR.DecSetoid_carrier}{ix1:expression sort (sort ix0)}Prop}
   (P ix0 ix1) -> P iy0 iy1;
...


Goal loop_extract_b
  : {sort|SORT(VAR)}{default:expression sort bool}
    {S:prog sort}expression sort bool;
...

Goal loop_extract_body
  : {sort|SORT(VAR)}{S:prog sort}prog sort;
...

Discharge VAR;




Module operate Import syntax;

[VAR|DecSetoid];

(*  Corresponds to Definition 3.24 on page 80. *)
Inductive
  [prog_operate
    : {vardecl|VAR.DecSetoid_carrier->Type}
      {sigma:STATE vardecl}{S:prog vardecl}{tau:STATE vardecl}Prop]

Relation NoReductions Inversion

Constructors
  [skip_operate
    : {vardecl|VAR.DecSetoid_carrier->Type}{sigma:STATE vardecl}
      prog_operate sigma (skip vardecl) sigma]
  [assign_operate
```

```
    : {vardecl|VAR.DecSetoid_carrier->Type}
      {x:VAR.DecSetoid_carrier}{t:expression vardecl (vardecl x)}
      {sigma:STATE vardecl}
      prog_operate sigma (assign x t) (STATE_update x (t sigma) sigma)]
  [seq_operate
    : {vardecl|VAR.DecSetoid_carrier->Type}{S1,S2:prog vardecl}
      {sigma,tau,eta:STATE vardecl}
      (prog_operate sigma S1 tau) ->
      (prog_operate tau S2 eta)->
      prog_operate sigma (seq S1 S2) eta]
  [ifthen_operate
    : {vardecl|VAR.DecSetoid_carrier->Type}
      {b:expression vardecl bool}{S1,S2:prog vardecl}
      {sigma,tau:STATE vardecl}
      (is_true (b (sigma))) ->
      (prog_operate sigma S1 tau) ->
      prog_operate sigma (ifthenelse b S1 S2) tau]
  [ifelse_operate
    : {vardecl|VAR.DecSetoid_carrier->Type}
      {b:expression vardecl bool}{S1,S2:prog vardecl}
      {sigma,tau:STATE vardecl}
      (is_false (b (sigma))) ->
      (prog_operate sigma S2 tau) ->
      prog_operate sigma (ifthenelse b S1 S2) tau]
  [while_exit_operate
    : {vardecl|VAR.DecSetoid_carrier->Type}
      {b:expression vardecl bool}{body:prog vardecl}
      {sigma:STATE vardecl}
      (is_false (b (sigma))) ->
      prog_operate sigma (while b body) sigma]
  [while_body_operate
    : {vardecl|VAR.DecSetoid_carrier->Type}
      {b:expression vardecl bool}{body:prog vardecl}
      {sigma,tau,eta:STATE vardecl}
      (is_true (b (sigma))) ->
      (prog_operate sigma body tau) ->
      (prog_operate tau (while b body) eta) ->
      prog_operate sigma (while b body) eta]
  [new_operate
    : {vardecl|VAR.DecSetoid_carrier->Type}{T|Type}
      {x:VAR.DecSetoid_carrier}{t:expression vardecl T}
      {S:prog (sort_update x T vardecl)}
      {sigma|STATE vardecl}{tau|STATE (sort_update x T vardecl)}
      (prog_operate (STATE_update_dyn x (t sigma) sigma) S tau) ->
      prog_operate sigma (new x t S) (STATE_update_dyn' x (sigma x) tau)];

Discharge VAR;



Module operate_thms Import operate lib_bool_thms;

[VAR|DecSetoid];

Goal oper_Inversion
  : {vardecl|(DecSetoid_carrier VAR)->Type}
    {S:prog vardecl}{sigma,tau:STATE vardecl}Prop;
...

Goal oper_inversion
  : {sort|VAR.DecSetoid_carrier->Type}
    {sigma|STATE sort}{S|prog sort}{tau|STATE sort}
    {D:prog_operate sigma S tau}(oper_Inversion S sigma tau);
...


Goal oper_inversion_ifthenelse :
  {sort|VAR.DecSetoid_carrier->Type}{c|bool}
  {b|expression sort bool}{S1,S2|prog sort}{sigma,tau|STATE sort}
  (Eq (b sigma) c)->(prog_operate sigma (ifthenelse b S1 S2) tau) ->
```

```
  prog_operate sigma (if c S1 S2) tau;
...

Goal oper_inversion_while :
  {sort|VAR.DecSetoid_carrier->Type}
  {c|bool}{b|expression sort bool}{S|prog sort}{sigma,tau|STATE sort}
  (Eq (b sigma) c)->(prog_operate sigma (while b S) tau) ->
  if c
     (Ex [rho:STATE sort]and (prog_operate sigma S rho)
                         (prog_operate rho (while b S) tau))
     (Eq sigma tau);
...

Goal oper_determ :
  {sort|VAR.DecSetoid_carrier->Type}
  {sigma|STATE sort}{S|prog sort}{tau|STATE sort}
  {Prem_tau|prog_operate sigma S tau}{C:(STATE sort)->Prop}
  {eta:STATE sort}
  {Prem_eta:prog_operate sigma S eta}
  {C_eta:C eta}C tau;
...

Discharge VAR;



Module  termination  Import operate_thms;

[VAR|DecSetoid][sort|SORT|VAR];

[Termin [S:prog sort] =
  [sigma:STATE sort]
  Ex [tau:STATE sort]prog_operate sigma S tau : (STATE sort).Pred];

Goal seq_termin1 :
  {S1,S2|prog sort}{sigma|STATE sort}(Termin (seq S1 S2) sigma) ->
  Termin S1 sigma;
...

Goal seq_termin2 :
  {S1,S2|prog sort}{sigma,tau|STATE sort}
  (Termin (seq S1 S2) sigma) -> (prog_operate sigma S1 tau) ->
  Termin S2 tau;
...

Goal ifthenelse_termin :
  {b|expression sort bool}{S1,S2|prog sort}{sigma|STATE sort}{c|bool}
  {eq:Eq (b sigma) c}
  (Termin (ifthenelse b S1 S2) sigma) ->
  if c (Termin S1 sigma) (Termin S2 sigma);
...


Goal while_termin :
  {b|expression sort bool}{S|prog sort}{sigma,tau|STATE sort}
  (Termin (while b S) sigma) ->
  (is_true (b sigma)) ->
  (prog_operate sigma S tau) ->
  Termin (while b S) tau;
...

Goal while_termin_body :
  {b|expression sort bool}{S|prog sort}{sigma|STATE sort}
  (is_true (b sigma)) -> (Termin (while b S) sigma) -> Termin S sigma;
...

Discharge VAR;



Module  WF  Import TransClosure termination syntax;
```

```
[VAR|DecSetoid][sort|SORT|VAR];

[b:expression(sort)(bool)][body:prog(sort)];

[sofar
  [sort|SORT|VAR][b:expression(sort)(bool)][body:prog(sort)]
  [tau,sigma:STATE sort]
   = TransClosure
        ([sigma,tau:STATE(sort)](is_true (b sigma)) ∧
                    (prog_operate sigma body tau)) sigma tau];

Goal WF_sofar :
  WF ([tau,sigma:STATE(sort)](Termin (while b body) sigma ∧
                                sofar b body tau sigma));
...

Discharge VAR;



Module H_AuxVar Import lib_rel lib_nat assertion operate;

[VAR|DecSetoid];

[Assertion_compress
  [T,U|Type][vardecl|VAR.DecSetoid_carrier->Type][x|VAR.DecSetoid_carrier]
  [P:Assertion T vardecl][v:vardecl x]
   = [z:T][sigma:STATE (sort_update x U vardecl)]
     P z (STATE_update_dyn' x v sigma)];

(*  Corresponds to Figure 3.5 on page 82. *)
Inductive
  [HD : {T|Type}{vardecl|VAR.DecSetoid_carrier->Type}
{P:Assertion T vardecl}{S:prog vardecl}{Q:Assertion T vardecl}Prop]
Relation NoReductions

Constructors
  [HDSkip :
    {T|Type}{vardecl:VAR.DecSetoid_carrier->Type}{p:Assertion T vardecl}
    HD p (skip vardecl) p]

  [HDAssign
    : {T|Type}{vardecl:VAR.DecSetoid_carrier->Type}
      {x:VAR.DecSetoid_carrier}{t:expression vardecl (vardecl x)}
      {p:Assertion T vardecl}
      HD (UpdateAssertion x t p) (assign x t) p]

  [HDSeq
    : {T|Type}{vardecl:VAR.DecSetoid_carrier->Type}
      {p,r,q|Assertion T vardecl}{S1,S2:prog vardecl}
      (HD p S1 r) -> (HD r S2 q) -> HD p (seq S1 S2) q]

  [HDIfthenelse :
    {T|Type}{vardecl:VAR.DecSetoid_carrier->Type}{pre,post|Assertion T vardecl}
    {b|expression vardecl bool}{TH,EL:prog vardecl}
    (HD ([z:T][sigma:STATE vardecl]and (pre z sigma)
                    (is_true (b (sigma))))
        TH post) ->
    (HD ([z:T][sigma:STATE vardecl]
         and (pre z sigma)
     (is_false (b (sigma))))
        EL post) ->
    HD pre (ifthenelse b TH EL) post]

  [HDWhile :
   {T,W|Type}{vardecl:VAR.DecSetoid_carrier->Type}
   {Lt':Rel W W}{is_wf : WF Lt'}
   {u:expression(vardecl)(W)} (* convergence function *)
   {p:Assertion(T)(vardecl)}{b:expression(vardecl)(bool)}{body:prog vardecl}
   {While_step:{t:W}
```

```
     HD ([Z:T][sigma:STATE vardecl]
         (p Z sigma) ∧ (is_true(b sigma)) ∧ (Eq (u sigma) t))
        body ([Z:T][tau:STATE vardecl](p Z tau) ∧ (Lt' (u tau) t))}
   HD p (while b body) ([Z:T][tau:STATE vardecl](p Z tau) ∧ (is_false(b tau)))]

  [HDNew :
    {T|Type}{vardecl:VAR.DecSetoid_carrier->Type}{p,q|Assertion T vardecl}
    {x:VAR.DecSetoid_carrier}{U|Type}{t|expression vardecl U}
    {S|prog (sort_update x U vardecl)}
    {HDNewPremiss:
      {v:vardecl x}
      HD ([z:T][sigma:STATE (sort_update x U vardecl)]
        (Assertion_compress p v z sigma) ∧
        (Eq (coerce (sort_update_lookup x U vardecl) (sigma x))
            (t (STATE_update_dyn' x v sigma))))
        S (Assertion_compress q v)}
     HD p (new x t S) q]

  [HDConsequence
    : {T|Type}{vardecl:VAR.DecSetoid_carrier->Type}{p,q:Assertion T vardecl}
      {T1|Type}{p1,q1:Assertion T1 vardecl}
      {S:prog vardecl}
      {ConsSC: KSC p q p1 q1}
      {ConsPrem:HD p1 S q1} HD p S q];


Discharge VAR;



Module Htheorems Import H_AuxVar;

[VAR|DecSetoid][sort|SORT|VAR];

Goal HoareConsequenceLeft :
  {T|Type}
  {pre_s : Assertion T sort}{pre,post:Assertion T sort}{S:prog(sort)}
  {strengthen : {z:T}{sigma:STATE(sort)}(pre_s z sigma) -> pre z sigma}
  {prem : HD pre S post}
  HD pre_s S post;
...

Goal HDSkipBU :
  {T|Type}{p,q:Assertion T sort}
  ({z:T}{sigma:STATE(sort)}(p z sigma)->q z sigma) -> HD p (skip ?) q;
...

Goal HDAssignBU :
  {x:VAR.DecSetoid_carrier}{t:expression(sort) (sort x)}{T|Type}
  {p,p':Assertion T sort}
  {p'p:{z:T}{sigma:STATE(sort)}
       (p' z sigma) ->  (compose (p z) (STATE_update x (t sigma)) sigma)}
  HD p' (assign x t) p;
...

Discharge VAR;



(* Semantics of Hoare logic*)

Module Hsemantics Import assertion operate;

[VAR|DecSetoid][T|Type][vardecl|VAR.DecSetoid_carrier->Type]

[HSem [p:Assertion T vardecl][S:prog vardecl][q:Assertion T vardecl]
  = {z|T}{sigma|STATE vardecl}
    (p z sigma) ->
    Ex [tau:STATE vardecl]and (prog_operate sigma S tau) (q z tau)];

Discharge VAR;
```

```
(* Soundness of Hoare logic *)
Module Hsound Import Hsemantics H_AuxVar operate_thms;

[VAR|DecSetoid];

(** Our new consequence rule preserves soundness **)
$Goal HDConsequencesoundness :
  {T|Type}{vardecl|(DecSetoid_carrier VAR)->Type}{p,q:Assertion T vardecl}
  {T1|Type}{p1,q1:Assertion T1 vardecl}
  {S:prog vardecl}
  {ConsSC:KSC p q p1 q1}
  {ConsPrem:HSem p1 S q1}HSem p S q;
...

(*  Corresponds to Theorem 3.25 on page 81. *)
Goal Hsoundness :
  {T|Type}{vardecl|VAR.DecSetoid_carrier->Type}
  {P|Assertion T vardecl}{S|prog vardecl}{Q|Assertion T vardecl}
  {der:HD P S Q}HSem P S Q;
...

Discharge VAR;



Module HAuxComplete Import operate_thms Hsemantics WF Htheorems;

[VAR|DecSetoid];

Freeze Eq;

(*  Corresponds to Theorem 3.26 on page 83. *)
Goal HL_most_general :
   {sort|SORT|VAR}{S:prog(sort)}
    HD ([Z,sigma:STATE(sort)]prog_operate sigma S Z)
       S (Eq|(STATE(sort)));
...

(*  Corresponds to Corollary 3.27 on page 83. *)
Goal HL_complete :
  {sort|SORT|VAR}{S|prog(sort)}{T|Type}{p,q|Assertion T sort}
  {semant:HSem p S q}HD p S q;
...

Discharge VAR;



Module Hlc Import Hsound HAuxComplete;

[VAR:DecSetoid];

(*  Corresponds to Corollary 3.28 on page 84. *)
Goal HDBlocksLC :
  {T|Type}{vardecl:VAR.DecSetoid_carrier->Type}
  {x:VAR.DecSetoid_carrier}{U|Type}
  [vardecl' = sort_update x U vardecl]
  {p|Assertion T vardecl'}{q|Assertion T vardecl}
  {t|expression vardecl U}
  {S|prog vardecl'}
  {Prem:{v:vardecl x}
    HD p S ([Z:T][tau:STATE vardecl']q Z (STATE_update_dyn' x v tau))}
  HD ([Z:T][sigma:STATE vardecl]
       p Z (STATE_update_dyn x (eval sigma t) sigma))
     (new x t S) q;

intros; Refine HL_complete; Expand HSem;
intros; Refine Hsoundness (Prem (sigma x)); Assumption +2;
intros tau tau_details; andE tau_details;
Refine ExIn; Refine -0 pair; Refine +1 new_operate;
Immed;
```

```
Save HDBlocksLC;

Discharge VAR;


Module VDM Import lib_rel syntax;

[VAR|DecSetoid][vardecl|VAR.DecSetoid_carrier->Type];

[VDM_Pred_to_Rel
  [P:Pred (STATE vardecl)] =
  [sigma,_:STATE vardecl]P sigma : Rel (STATE vardecl) (STATE vardecl)];

[reflclose [R:Rel (STATE vardecl) (STATE vardecl)] =
  [tau,sigma:STATE vardecl](R tau sigma) ∨ (Eq tau sigma)];

Discharge vardecl;

(*  Corresponds to Figure 3.7 on page 86. *)
Inductive
  [VDMder : {vardecl|VAR.DecSetoid_carrier->Type}
            {P:Pred (STATE vardecl)}{S:prog vardecl}
    {Q:Rel (STATE vardecl) (STATE vardecl)}Prop]

Relation NoReductions

Constructors
  [VDMSkip :
    {vardecl|VAR.DecSetoid_carrier->Type}
    VDMder ([_:(STATE vardecl)]trueProp) (skip vardecl)
           ([sigma,tau:(STATE vardecl)]Eq tau sigma)]

  [VDMAssign :
    {vardecl|VAR.DecSetoid_carrier->Type}
    {x:VAR.DecSetoid_carrier}{t:expression vardecl (vardecl x)}
    VDMder ([_:(STATE vardecl)]trueProp)
           (assign x t)
   ([tau,sigma:(STATE vardecl)]
     Eq tau (STATE_update x (t(sigma)) sigma))]

  [VDMSeq :
    {vardecl|VAR.DecSetoid_carrier->Type}
    {p1,p2:(STATE vardecl).Pred}
    {r1,r2:Rel (STATE vardecl) (STATE vardecl)}{S1,S2:prog vardecl}
    (VDMder p1 S1 [tau,sigma:(STATE vardecl)]and (p2 tau) (r1 tau sigma)) ->
    (VDMder p2 S2 r2) ->
    VDMder p1 (seq S1 S2) (composeRel r2 r1)]

  [VDMIfthenelse :
    {vardecl|VAR.DecSetoid_carrier->Type}
    {pre:(STATE vardecl).Pred}{post:Rel (STATE vardecl) (STATE vardecl)}
    {b:expression vardecl bool}{TH,EL:prog vardecl}
    (VDMder ([sigma:(STATE vardecl)]
      (pre sigma) ∧ (is_true (eval sigma b)))
           TH post) ->
    (VDMder ([sigma:(STATE vardecl)]
      (pre sigma) ∧ (is_false (eval sigma b)))
           EL post) ->
    VDMder pre (ifthenelse b TH EL) post]

  [VDMWhile :
   {vardecl|VAR.DecSetoid_carrier->Type}
   {P:(STATE vardecl).Pred}
   {sofar:Rel (STATE vardecl) (STATE vardecl)}
     {sofar_trans: trans sofar}
     {sofar_wf: WF (andRel sofar (VDM_Pred_to_Rel P))}
   {b:expression vardecl bool}{body:prog vardecl}
   {Prem:VDMder ([sigma:(STATE vardecl)]
  (P sigma) ∧ (is_true (eval sigma b)))
                body
```

```
        ([tau,sigma:(STATE vardecl)](P tau) ∧ (sofar tau sigma))}
    VDMder P (while b body)
           ([tau,sigma:(STATE vardecl)]
     (P tau) ∧ (is_false (eval tau b)) ∧
     (reflclose sofar tau sigma))]

  [VDMNew :
    {vardecl:VAR.DecSetoid_carrier->Type}
    {p|Pred (STATE vardecl)}{q|Rel (STATE vardecl) (STATE vardecl)}
    {x:VAR.DecSetoid_carrier}{U|Type}{t|expression vardecl U}
    {S|prog (sort_update x U vardecl)}
    {HDNewPremiss:
      {v:vardecl x}
      VDMder ([sigma:STATE (sort_update x U vardecl)]
       p (STATE_update_dyn' x v sigma) ∧
        (Eq (coerce (sort_update_lookup x U vardecl) (sigma x))
            (t (STATE_update_dyn' x v sigma))))
       S ([tau,sigma:STATE (sort_update x U vardecl)]
    q (STATE_update_dyn' x v tau)
      (STATE_update_dyn' x v sigma))}
     VDMder p (new x t S) q]

  [VDMConsequence :
    {vardecl|VAR.DecSetoid_carrier->Type}
    {p,p1:Pred (STATE vardecl)}{q,q1:Rel (STATE vardecl) (STATE vardecl)}
    {S:prog vardecl}
    {ConsPre:{sigma|(STATE vardecl)}(p sigma) -> p1 sigma}
    {ConsPost:{tau,sigma|(STATE vardecl)}(p sigma) ->
              (q1 tau sigma) -> q tau sigma}
    {ConsPrem:VDMder p1 S q1}
    VDMder p S q];

Discharge VAR;



Module VDMtheorems Import VDM;

[VAR|DecSetoid][vardecl|VAR.DecSetoid_carrier->Type];

Goal VDMPre :
  {pre|Pred (STATE vardecl)}{S|prog vardecl}
  {post|Rel (STATE vardecl) (STATE vardecl)}
  {Prem:VDMder pre S post}
  VDMder pre S
    [tau,sigma:(STATE vardecl)]and (pre sigma) (post tau sigma);
...

Goal VDMConsequenceLeft :
  {pre_s,pre : Pred (STATE vardecl)}
  {post:Rel (STATE vardecl) (STATE vardecl)}{S:prog vardecl}
  {strengthen : {sigma:(STATE vardecl)}(pre_s sigma) -> pre sigma}
  {prem : VDMder pre S post}
  VDMder pre_s S post;
...

Goal VDMConsequenceRight:
  {pre : Pred (STATE vardecl)}
  {post,post_w:Rel (STATE vardecl) (STATE vardecl)}{S:prog vardecl}
  {weaken :
    {tau,sigma:(STATE vardecl)}(post tau sigma) -> post_w tau sigma}
  {prem : VDMder pre S post}
  VDMder pre S post_w;
...

Goal VDMSkipBU :
  {p:Rel (STATE vardecl) (STATE vardecl)}{p':Pred (STATE vardecl)}
  {p'p:{sigma:(STATE vardecl)}(p' sigma) -> p sigma sigma}
  VDMder p' (skip ?) p;
...
```

```
Goal VDMAssignBU :
  {x:VAR.DecSetoid_carrier}{t:expression vardecl (vardecl x)}
  {p:Rel (STATE vardecl) (STATE vardecl)}{p':Pred (STATE vardecl)}
  {p'p:
    {sigma:(STATE vardecl)}(p' sigma) ->
    p (STATE_update x (t sigma) sigma) sigma}
  VDMder p' (assign x t) p;
...

Goal VDMSeqBU :
  {p1,p2:Pred (STATE vardecl)}
  {r,r1,r2:Rel (STATE vardecl) (STATE vardecl)}{S1,S2:prog vardecl}
  {weakening: SubRel (composeRel r2 r1) r}
  (VDMder p1 S1 [tau,sigma:(STATE vardecl)]and (p2 tau) (r1 tau sigma)) ->
  (VDMder p2 S2 r2) ->
  VDMder p1 (seq S1 S2) r;
...

Goal VDMLoopBU :
   {P,I:Pred (STATE vardecl)}
   {Q,sofar:Rel (STATE vardecl) (STATE vardecl)}
     {sofar_trans: trans sofar}
     {sofar_wf: WF (andRel sofar (VDM_Pred_to_Rel I))}
   {b:expression vardecl bool}{body:prog vardecl}
   {ScPre:SubPred P I}
   {ScPost:{tau,sigma:(STATE vardecl)}(I sigma) -> (I tau) ->
           (is_false (eval tau b)) ->
   (reflclose sofar tau sigma) -> Q tau sigma}
   {Prem:
     VDMder ([sigma:(STATE vardecl)]
      (I sigma) ∧ (is_true (eval sigma b)))
               body
     ([tau,sigma:(STATE vardecl)](I tau) ∧ (sofar tau sigma))}
   VDMder P (while b body) Q;
...

Discharge VAR;



Module VDMsemantics Import lib_rel operate;

[VAR|DecSetoid][vardecl|SORT VAR];

[VDMSem
  [p:Pred (STATE vardecl)]
  [S:prog vardecl]
  [r:Rel (STATE vardecl) (STATE vardecl)]
  = {sigma:STATE vardecl}
    (p sigma) ->
  Ex [tau:STATE vardecl]and (prog_operate sigma S tau) (r tau sigma)];

Discharge VAR;


(* Soundness of VDM's decomposition rules *)
Module VDMsound Import VDMsemantics VDM operate_thms;

[VAR|DecSetoid];

(*  Corresponds to Theorem 3.29 on page 85. *)
Goal VDMsoundness :
  {vardecl|VAR.DecSetoid_carrier->Type}
  {P|Pred (STATE vardecl)}
  {S|prog vardecl}
  {Q|Rel (STATE vardecl) (STATE vardecl)}
  {der:VDMder P S Q}VDMSem P S Q;
...

Discharge VAR;
```

```
Module VDMcomplete Import VDMsemantics VDMtheorems termination WF;

[VAR|DecSetoid];

[vardecl|SORT VAR][b:expression vardecl bool][body:prog vardecl];

[VDM_Most_General [S:prog vardecl] =
  VDMder (Termin S) S ([tau,sigma:STATE vardecl]prog_operate sigma S tau)];

$Goal VDMWFsofar :
  WF (andRel (sofar b body) (VDM_Pred_to_Rel (Termin (while b body))));
...

Freeze Eq;

Goal VDM_most_general_loop :
  {ih:VDM_Most_General body}VDM_Most_General (while b body);
...

Discharge vardecl;

Goal VDM_most_general :
  {vardecl|VAR.DecSetoid_carrier->Type}{S:prog vardecl}
  VDM_Most_General S;
...

(*  Corresponds to Theorem 3.30 on page 85. *)
Goal VDM_complete :
  {vardecl|SORT VAR}
  {S|prog vardecl}
  {p|(STATE vardecl).Pred}{q|Rel (STATE vardecl) (STATE vardecl)}
  {semant:VDMSem p S q}VDMder p S q;
...

Discharge VAR;
```

B.4 Recursive Procedures and Static Binding

```
Module state Import lib_bool_funs DepUpdate;


[VAR|DecSetoid];
[gsort:SORT|VAR];

Record [STATE_internal:Type]
Parameters [lsort:SORT|VAR] Theorems
Fields [global_internal : FS gsort][local_internal : FS lsort];

[Env_internal = VAR.DecSetoid_carrier->bool]
[E:Env_internal];
[Env_internal_empty = [x:VAR.DecSetoid_carrier]false : Env_internal];

[Env_add [f:Env_internal][y:VAR.DecSetoid_carrier]
  = [x:VAR.DecSetoid_carrier]if (VAR.DecSetoid_eq x y) true (f x)
    : Env_internal];

Goal Env_add_lookup :
  {x:VAR.DecSetoid_carrier}
  Eq (Env_add E x x) true;
...


(*  Corresponds to Definition 3.32 on page 88. *)
[STATE = STATE_internal];

[x:VAR.DecSetoid_carrier]

[if_local [T|Type][t,e:T] = if (E x) t e];
```

```
[AccSort = if_local (lsort x) (gsort x)];

Goal STATE_update_internal : {t:AccSort}{sigma:STATE}STATE;
Expand AccSort; Expand if_local;
Induction (E x);

  (** updating local component **)
  intros; Refine make_STATE_internal sigma.global_internal;
    Refine FS_update x t sigma.local_internal;

  (** updating global component **)
  intros; Refine make_STATE_internal ? sigma.local_internal;
    Refine FS_update x t sigma.global_internal;
Save STATE_update_internal;

Discharge x;

(* Corresponds to Definition 3.33 on page 89.*)
[expression [sort:SORT|VAR][T:Type] = (FS sort) -> T];

Goal lookup_internal : {sigma:STATE}{x:VAR.DecSetoid_carrier}(AccSort x);
intros; Expand AccSort; Expand if_local;
Induction (E x);
  Refine sigma.local_internal x;
  Refine sigma.global_internal x;
Save lookup_internal;

[T|Type][sigma:STATE][t:expression AccSort T]

[eval_internal  = t (lookup_internal sigma)];

[STATE_update_local_internal [x:VAR.DecSetoid_carrier][v:lsort x]
  = make_STATE_internal sigma.global_internal
                        (FS_update x v sigma.local_internal)
  : STATE];

Discharge E;

Goal Env_empty_global :
  {x:VAR.DecSetoid_carrier}Eq (gsort x) (AccSort Env_internal_empty x);
...


Discharge lsort;

[lsortA|SORT VAR];

Goal STATE_coerce_internal :
  {lsortB|SORT VAR}
  {coerce:EXTEQ lsortA lsortB}{sigma:STATE lsortA}STATE lsortB;
...

Goal STATE_coerce_redundant_internal :
  {sigma:STATE lsortA}
  Eq (STATE_coerce_internal (EXTEQrefl ?) sigma) sigma;
...

Discharge lsortA;

Goal FS_update_commutative :
  {x,y|VAR.DecSetoid_carrier}{T|Type(0)}{v:T}{lsort1|SORT VAR}
  [lsort'1 = sort_update x T lsort1]
  {prog_eq'y : Eq (lsort1 y) (lsort'1 y)}
  {E:Env_internal}{xglobal:is_false (E x)}
  {prog_eq : EXTEQ (AccSort lsort1 E) (AccSort lsort'1 E)}
  {t : expression (AccSort lsort1 E) (lsort1 y)}
  {eq:is_false (VAR.DecSetoid_eq y x)}
  {sigma1 : STATE lsort1}
  Eq (FS_update y
```

```
               (eval_internal ? E
                 (make_STATE_internal ? (global_internal ? sigma1)
                   (FS_update_dyn x v
                     (local_internal ? sigma1)))
                 ([sigma'2:FS (AccSort lsort'1 E)]
                  Eq_subst prog_eq'y ([z'3:Type(0)]z'3)
                   (t ([var:DecSetoid_carrier VAR]
                        Eq_subst (Eq_sym (prog_eq var)) ([z'4:Type(0)]z'4)
                         (sigma'2 var)))))
               (FS_update_dyn x v
                 (local_internal ? sigma1)))
           (FS_update_dyn x v
             (FS_update y (eval_internal ? E sigma1 t) (local_internal ? sigma1)));
...

Goal AccSort_update :
  {T:Type(0)}{E:Env_internal}{lsort:SORT|VAR}
  {x,y:VAR.DecSetoid_carrier}
  Eq (sort_update x T (AccSort lsort E) y)
     (AccSort (sort_update x T lsort) (Env_add E x) y);
...


Discharge gsort;

[gsort,lsort|SORT|VAR]

[lookup             = lookup_internal gsort lsort]
[STATE_update       = STATE_update_internal gsort lsort];
[STATE_update_local = STATE_update_local_internal gsort lsort]
[eval               = eval_internal gsort lsort];
[global             = global_internal gsort lsort]
[local              = local_internal gsort lsort]

[make_STATE         = make_STATE_internal gsort lsort
                    : (FS gsort) -> (FS lsort) -> STATE gsort lsort];

[STATE_coerce       = STATE_coerce_internal gsort];

[STATE_coerce_redundant
  = STATE_coerce_redundant_internal gsort|lsort
  : {sigma:STATE gsort lsort}
    Eq (STATE_coerce (EXTEQrefl ?) sigma) sigma];

DischargeKeep lsort;

[E:Env_internal][sigma:STATE gsort lsort][x:VAR.DecSetoid_carrier][T|Type]
[t:expression (AccSort gsort lsort E) T];

[STATE_add_local' [v:T]
  = make_STATE sigma.global (FS_update_dyn x v sigma.local)];

[STATE_add_local = STATE_add_local'(eval E sigma t)];

[v:lsort x][tau:STATE gsort (sort_update x T lsort)]

[STATE_remove_local
  =  make_STATE tau.global (FS_update_dyn' x v tau.local)
  : STATE gsort lsort];

[STATEuL = STATE_remove_local]

[STATE_elim = STATE_internal_elim];



Discharge v;

Goal STATE_remove_add' :
  {v:T}Eq (STATE_remove_local (local sigma x) (STATE_add_local' v))
```

```
      sigma;
...

Goal STATE_remove_add :
 Eq (STATE_remove_local (local sigma x) STATE_add_local)
    sigma;
...

Discharge sigma;

Goal STATE_add_remove :
  {T|Type(0)}{x|VAR.DecSetoid_carrier}
  {sigma:STATE gsort (sort_update x T lsort)}{v:lsort x}
  Eq sigma
    (STATE_add_local' (STATE_remove_local x v sigma) x
      (coerce (sort_update_lookup x T lsort) (local sigma x)));
...

DischargeKeep lsort;

Goal STATE_coerce_add_remove :
  {T|Type(0)}{y:VAR.DecSetoid_carrier}
  {v:lsort y}[lsort' = sort_update y T lsort]{sigma:STATE gsort lsort'}
  Eq (STATE_coerce (sort_dupdate y ? T) (STATE_add_local' sigma y v))
  (STATE_remove_local y v sigma);
...

Discharge VAR;

[VAR:DecSetoid]
[Env  = Env_internal|VAR]
[Env_empty = Env_internal_empty|VAR : Env];

Discharge VAR;
```

```
Module assertion Import state;

[VAR|DecSetoid]

(*  Corresponds to Definition 3.34 on page 90. *)
[Assertion [gsort,lsort:SORT VAR][T:Type] =
  T -> (STATE gsort lsort) -> Prop];

[gsort,lsort|VAR.DecSetoid_carrier->Type][T|Type]

[Assertion_and [p,q:Assertion gsort lsort T]
  = [Z:T][sigma:STATE gsort lsort](p Z sigma) ∧ (q Z sigma)
  : Assertion gsort lsort T]

[E:VAR.DecSetoid_carrier->bool][x:VAR.DecSetoid_carrier]

$[AS = AccSort gsort lsort E]

[Assertion_update
[t:expression AS (AS x)]
  [p:Assertion gsort lsort T]
  = [Z:T][sigma:STATE gsort lsort]
     p Z (STATE_update E x (eval E sigma t) sigma)
  : Assertion gsort lsort T];

[Assertion_update_const [v:AS x]
  = Assertion_update [_:FS|?|AS]v];

[U,V|Type(0)][p',q':Assertion gsort lsort V]
$[lsort' = sort_update x U lsort];
```

```
[Fr [T|Type(0)][p:Assertion gsort lsort T][v:lsort x]
  = [Z:T][sigma:STATE gsort lsort']p Z (STATE_remove_local x v sigma)];

[Assertion_preserve [T|Type(0)][p:Assertion gsort lsort T][v:lsort x]
  = [Z:T#U][sigma:STATE gsort lsort']
  (Fr p v Z.1 sigma) ∧
  (Eq (coerce (sort_update_lookup x U lsort) (sigma.local x)) Z.2)];

[KSC [x:(Assertion gsort lsort T)#(Assertion gsort lsort T)]
     [y:(Assertion gsort lsort U)#(Assertion gsort lsort U)]
  = {Z:T}{sigma,tau:STATE gsort lsort}
    (x.1 Z sigma) ->
    Ex [Z':U]and (y.1 Z' sigma) ((y.2 Z' tau) -> x.2 Z tau)];

DischargeKeep U;

Goal KSC_refl : refl (KSC|T);
...

Goal KSC_trans : trans (KSC|T);
...

Discharge VAR;



Module syntax Import state;

(* Corresponds to Definition 3.36 on page 91.*)
Inductive [prog:{sort:SORT|VAR}Type(2)] ElimOver Type

Parameters [VAR|DecSetoid]

Constructors
  [skip : {sort:SORT|VAR}prog sort]

  [assign
    : {sort|SORT|VAR}
      {x:VAR.DecSetoid_carrier}{t:expression sort (sort x)}prog sort]

  [seq
    : {sort|SORT|VAR}{S1,S2:prog sort}prog sort]

  [ifthenelse
    : {sort|SORT|VAR}{b:expression sort bool}
      {S1,S2:prog sort}prog sort]

  [while
    : {sort|SORT|VAR}{b:expression sort bool}
      {body:prog sort}prog sort]

  [call : {sort:SORT|VAR}prog sort]

  [new
    : {sort|SORT|VAR}{T|Type(0)}
      {x:VAR.DecSetoid_carrier}{t:expression sort T}
      {S:prog (sort_update x T sort)} prog sort];

Goal ifcall :
  {sort|SORT|VAR}{S:prog sort}
  {then:Prop}{else:Prop}Prop;
...

Goal ifloop
  : {sort|SORT|VAR}{S:prog sort}{T|Type}
    {then:T}{else:T}T;
...

Goal ifblock
  : {sort|SORT|VAR}{S:prog sort}{T|Type}
```

```
    {then:T}{else:T}T;
...

Goal loop_extract_b :
  {sort|SORT|VAR}{default:expression sort bool}
  {S:prog sort} expression sort bool;
...

Goal loop_extract_body : {sort|SORT|VAR}{S:prog sort}prog sort;
...

Goal coerce_expression :
  {T|Type(0)}{VAR:DecSetoid}
  {sortA,sortB:SORT VAR}
  {coerce:{x:VAR.DecSetoid_carrier}Eq (sortA x) (sortB x)}
  {t:expression sortA T}expression sortB T;
...

Goal coerce_expression' :
  {VAR:DecSetoid}{x:VAR.DecSetoid_carrier}
  {sortA,sortB:SORT VAR}
  {coerce:{x:VAR.DecSetoid_carrier}Eq (sortA x) (sortB x)}
  {t:expression sortA (sortA x)}expression sortB (sortB x);
...

[sortA,sortB|SORT VAR];

[eq:EXTEQ sortA sortB];

Goal coerce_sort_update
  : {T:Type(0)}{x:DecSetoid_carrier VAR}
    EXTEQ (sort_update x T sortA) (sort_update x T sortB);
...

Discharge sortA;

Goal coerce_prog :
  {sortA,sortB|SORT VAR}
  {coerce:EXTEQ sortA sortB}
  {S:prog sortA}prog sortB;
...

Discharge VAR;



Module syntax_thms Import syntax lib_nat;

[VAR|DecSetoid];

[sortB|SORT VAR][n:nat][sortf:nat->SORT VAR]
[coerce_fam : {n:nat}EXTEQ (sortf n) (sortf (suc n))];

Goal expression_iter_coerce :
  {T|Type(0)}{sortA|SORT VAR}{t:expression sortA T}
  {coerce_init: EXTEQ sortA (sortf zero)}
  {coerce_fin : EXTEQ (sortf n) sortB}
  expression sortB T;
...

Goal prog_iter_coerce :
  {sortA|SORT VAR}{S:prog sortA}
  {coerce_init: EXTEQ sortA (sortf zero)}
  {coerce_fin : EXTEQ (sortf n) sortB}
  prog sortB;
...
```

```
Goal Eq_trans'
  : {sortA,sortB,sortC|SORT VAR}
    {ab : EXTEQ sortA sortB}{bc : EXTEQ sortB sortC}
    EXTEQ sortA sortC;
...


Goal coerce_expression_redun :
  {T|Type(0)}{sort|SORT VAR}{t:expression sort T}
  Eq (coerce_expression VAR sort sort (EXTEQrefl sort) t) t;
...

Goal coerce_prog_redun :
  {sort|SORT VAR}
  {S:prog sort}
  Eq (coerce_prog (EXTEQrefl sort) S) S;
...

[sortC:SORT VAR][c2:EXTEQ sortB sortC];

Goal coerce_expression_denest' :
  {sortA:SORT VAR}{c1:EXTEQ sortA sortB}
  {x:DecSetoid_carrier VAR}{t:expression sortA (sortA x)}
  Eq (coerce_expression' VAR x sortB sortC c2
        (coerce_expression' VAR x sortA sortB c1 t))
     (coerce_expression' VAR x sortA sortC (Eq_trans' c1 c2) t);
...

Goal coerce_expression_denest :
  {T|Type(0)}{sortA:SORT VAR}{c1:EXTEQ sortA sortB}
  {t:expression sortA T}
  Eq (coerce_expression VAR sortB sortC c2
        (coerce_expression VAR sortA sortB c1 t))
     (coerce_expression VAR sortA sortC (Eq_trans' c1 c2) t);
...

Discharge sortB;

Goal coerce_prog_denest :
  {sortA|SORT VAR}{S:prog sortA}{sortB,sortC|SORT VAR}
  {c1:EXTEQ sortA sortB}{c2:EXTEQ sortB sortC}
  Eq (coerce_prog c2 (coerce_prog c1 S))
     (coerce_prog (Eq_trans' c1 c2) S);
...

Freeze Eq;

Goal prog_elim' :
  {C_prog:{gsort,lsort|SORT|VAR}
  {E|Env|VAR}(prog (AccSort gsort lsort E))->Type}
  ({gsort,lsort:SORT|VAR}{E:Env|VAR}C_prog (skip (AccSort gsort lsort E)))->
  ({gsort,lsort|SORT|VAR}{E|Env|VAR}
   {x:DecSetoid_carrier VAR}
   {t:expression (AccSort gsort lsort E) (AccSort gsort lsort E x)}
   C_prog (assign x t))->
  ({gsort,lsort|SORT|VAR}{E|Env|VAR}
   {S1,S2:prog (AccSort gsort lsort E)}{S1_ih:C_prog S1}{S2_ih:C_prog S2}
   C_prog (seq S1 S2))->
  ({gsort,lsort|SORT|VAR}{E|Env|VAR}
   {b:expression (AccSort gsort lsort E) bool}
   {S1,S2:prog (AccSort gsort lsort E)}{S1_ih:C_prog S1}
   {S2_ih:C_prog S2}C_prog (ifthenelse b S1 S2))->
  ({gsort,lsort|SORT|VAR}{E|Env|VAR}
   {b:expression (AccSort gsort lsort E) bool}
   {body:prog (AccSort gsort lsort E)}{body_ih:C_prog body}
   C_prog (while b body))->
   ({gsort,lsort:SORT|VAR}{E:Env|VAR}C_prog (call (AccSort gsort lsort E)))->
  ({gsort,lsort|SORT|VAR}{E|Env|VAR}{T|Type(0)}
   {x:DecSetoid_carrier VAR}{t:expression (AccSort gsort lsort E) T}
```

```
   {S:prog (sort_update x T (AccSort gsort lsort E))}
   {S_ih:C_prog (coerce_prog (AccSort_update gsort T E lsort x) S)}
   C_prog (new x t S))->
  {gsort,lsort|SORT|VAR}{E|Env|VAR}{z:prog (AccSort gsort lsort E)}C_prog z;
...

Discharge VAR;

Unfreeze Eq;



Module operate Import lib_bool_thms syntax;

(* Corresponds to Definition 3.37 on page 91.*)
Inductive
  [prog_operate
    : {lsort|SORT|VAR}{E:Env|VAR}
      {sigma:STATE gsort lsort}
      {S:prog (AccSort gsort lsort E)}
      {tau:STATE gsort lsort}Prop]

Relation
(* Lego can not generate an Inversion theorem due to type dependency
   We prove it manually in the module operate_thms                *)
NoReductions

Parameters
  [VAR|DecSetoid][gsort|SORT|VAR][S0:prog gsort]

Constructors
  [skip_operate :
    {lsort|SORT|VAR}{E:Env|VAR}{sigma:STATE gsort lsort}
     prog_operate E sigma (skip (AccSort gsort lsort E)) sigma]

  [assign_operate
    : {lsort|SORT|VAR}{E:Env|VAR}{sigma:STATE gsort lsort}
      {x|VAR.DecSetoid_carrier}
      {t:expression (AccSort gsort lsort E) (AccSort gsort lsort E x)}
      prog_operate E sigma (assign x t)
        (STATE_update E x (eval E sigma t) sigma)]

  [seq_operate
    : {lsort|SORT|VAR}{E:Env|VAR}{S1,S2:prog (AccSort gsort lsort E)}
      {sigma,tau,eta:STATE gsort lsort}
      (prog_operate E sigma S1 tau) ->
      (prog_operate E tau S2 eta) ->
      prog_operate E sigma (seq S1 S2) eta]

  [ifthen_operate
     : {lsort|SORT|VAR}{E:Env|VAR}{b:expression (AccSort gsort lsort E) bool}
       {S1,S2:prog  (AccSort gsort lsort E)}
       {sigma,tau:STATE gsort lsort}
       (is_true (eval E sigma b)) ->
       (prog_operate E sigma S1 tau) ->
       prog_operate E sigma (ifthenelse b S1 S2) tau]

  [ifelse_operate
    : {lsort|SORT|VAR}{E:Env|VAR}{b:expression (AccSort gsort lsort E) bool}
      {S1,S2:prog  (AccSort gsort lsort E)}
      {sigma,tau:STATE gsort lsort}
      (is_false (eval E sigma b)) -> (prog_operate E sigma S2 tau) ->
      prog_operate E sigma (ifthenelse b S1 S2) tau]

  [while_exit_operate
    : {lsort|SORT|VAR}{E:Env|VAR}{b:expression (AccSort gsort lsort E) bool}
      {body:prog  (AccSort gsort lsort E)}
      {sigma:STATE gsort lsort}
      (is_false (eval E sigma b)) ->
      prog_operate E sigma (while b body) sigma]
```

```
  [while_body_operate
    : {lsort|SORT|VAR}{E:Env|VAR}{b:expression (AccSort gsort lsort E) bool}
      {body:prog  (AccSort gsort lsort E)}
      {sigma,tau,eta:STATE gsort lsort}
      (is_true (eval E sigma b)) ->
      (prog_operate E sigma body tau) ->
      (prog_operate E tau (while b body) eta) ->
      prog_operate E sigma (while b body) eta]


  [call_operate
    : {lsort|SORT|VAR}{E:Env|VAR}{sigma,tau:STATE gsort lsort}
      (prog_operate (Env_empty|VAR) sigma
         (coerce_prog (Env_empty_global gsort lsort) S0) tau)->
      (prog_operate E sigma (call (AccSort gsort lsort E)) tau)]


  [new_operate
    : {lsort|SORT|VAR}{E:Env|VAR}{T:Type(0)}
      {x:VAR.DecSetoid_carrier}{t:expression (AccSort gsort lsort E) T}
      {S:prog (sort_update x T (AccSort gsort lsort E))}
      {sigma:STATE gsort lsort}
      {tau:STATE gsort (sort_update x T lsort)}
      (prog_operate (Env_add E x)
         (STATE_add_local E sigma x t)
   (coerce_prog (AccSort_update gsort T E lsort x) S) tau) ->
      prog_operate E sigma (new x t S)
        (STATE_remove_local x (sigma.local x) tau)];

Discharge VAR;



Module operate_thms Import lib_bool_funs operate;

[VAR|DecSetoid];

Goal oper_inversion_assign :
  {sort|SORT|VAR}{x:DecSetoid_carrier VAR}{t:expression sort (sort x)}
  {gsort,lsort|SORT|VAR}(prog gsort)->{E:Env|VAR}
  {sort_eq:Eq sort (AccSort gsort lsort E)}{sigma:STATE gsort lsort}
  {tau:STATE gsort lsort}Prop;
...


Goal oper_Inversion :
  {gsort,lsort|SORT|VAR}
  {S0:prog gsort}{E:Env|VAR}
  {S:prog (AccSort gsort lsort E)}
  {sigma,tau:STATE gsort lsort}Prop;
...

Goal oper_inversion :
  {gsort,lsort|SORT|VAR}
  {S0|prog gsort}{E|Env|VAR}
  {sigma|STATE gsort lsort}
  {S|prog (AccSort gsort lsort E)}
  {tau|STATE gsort lsort}
  {D:prog_operate S0 E sigma S tau}
  (oper_Inversion S0 E S sigma tau);
...

[gsort,lsort|SORT|VAR][S0|prog gsort][E|Env|VAR]
[b|expression (AccSort gsort lsort E) bool];

Goal oper_inversion_ifthenelse :
  {S1,S2|prog (AccSort gsort lsort E)}{sigma,tau|STATE gsort lsort}
  {c|bool}{eq:Eq (eval E sigma b) c}
  (prog_operate S0 E sigma (ifthenelse b S1 S2) tau) ->
  prog_operate S0 E sigma (if c S1 S2) tau;
...
```

```
Goal oper_inversion_while :
  {c|bool}{S|prog (AccSort gsort lsort E)}{sigma,tau|STATE gsort lsort}
  (Eq (eval E sigma b) c)->
  (prog_operate S0 E sigma (while b S) tau) ->
  if c (Ex [rho:STATE gsort lsort]and (prog_operate S0 E sigma S rho)
                                (prog_operate S0 E rho (while b S) tau))
       (Eq sigma tau);
...

Discharge lsort;

Goal oper_determ :
  {lsort|SORT|VAR}
  {S0|prog gsort}{E|Env|VAR}
  {sigma|STATE gsort lsort}
  {S|prog (AccSort gsort lsort E)}
  {tau|STATE gsort lsort}
  {Prem_tau|prog_operate S0 E sigma S tau}{C:(STATE gsort lsort)->Prop}
  {eta:STATE gsort lsort}
  {Prem_eta:prog_operate S0 E sigma S eta}
  {C_eta:C eta}C tau;
...

Goal call_env_indep :
  {E,E':Env|VAR}{lsort|SORT|VAR}
  {S0|prog gsort}
  {sigma|STATE gsort lsort}{tau|STATE gsort lsort}
  [sort = AccSort gsort lsort]
  {Der:prog_operate S0 E  sigma (call E.sort) tau}
  prog_operate S0 E' sigma (call E'.sort) tau;
...

[lsort|SORT|VAR][x:VAR.DecSetoid_carrier][U|Type(0)]
[S0:prog gsort][E|Env|VAR][v:lsort x]
$[lsort' = sort_update x U lsort]
[sigma|STATE gsort lsort'][tau|STATE gsort lsort];

$[call_reduce_env_internal :
  [S=S0]
  (prog_operate S0 (Env_empty|?) (STATE_remove_local x v sigma)
     (coerce_prog (Env_empty_global gsort lsort) S) tau) ->
  prog_operate S0 (Env_empty|?) sigma
    (coerce_prog (Env_empty_global gsort lsort') S)
    (STATE_add_local' tau x (coerce (sort_update_lookup x U lsort)
    (local sigma x)))];

Goal call_reduce_env :
  {Prem:prog_operate S0 E (STATE_remove_local x v sigma)
           (call ?) tau}
  prog_operate S0 (Env_add E x) sigma (call ?)
    (STATE_add_local' tau x
       (coerce (sort_update_lookup x U lsort) (local sigma x)));
...

Discharge VAR;



Module termination Import operate_thms lib_rel;

[VAR|DecSetoid][gsort,lsort|SORT|VAR][E:Env|VAR][S0:prog gsort];

$[myAccSort = AccSort gsort lsort E]
$[myState   = STATE gsort lsort]

[Termin [S:prog myAccSort]
  = [sigma:myState]Ex [tau:myState]prog_operate S0 E sigma S tau
  : myState.Pred];
```

```
Goal seq_termin1 :
  {S1,S2|prog myAccSort}{sigma|myState}
  (Termin (seq S1 S2) sigma) -> Termin S1 sigma;
...

Goal seq_termin2 :
  {S1,S2|prog myAccSort}{sigma,tau|myState}
  (Termin (seq S1 S2) sigma) -> (prog_operate S0 E sigma S1 tau) ->
  Termin S2 tau;
...

Goal ifthenelse_termin :
  {b|expression myAccSort bool}{S1,S2|prog myAccSort}{sigma|myState}{c|bool}
  {eq:Eq (eval E sigma b) c}
  (Termin (ifthenelse b S1 S2) sigma) ->
  if c (Termin S1 sigma) (Termin S2 sigma);
...


Goal while_termin :
  {b|expression myAccSort bool}{S|prog myAccSort}{sigma,tau|myState}
  (Termin (while b S) sigma) ->
  (is_true (eval E sigma b)) ->
  (prog_operate S0 E sigma S tau) ->
  Termin (while b S) tau;
...

Goal while_termin_body :
  {b|expression myAccSort bool}{S|prog myAccSort}{sigma|myState}
  (is_true (eval E sigma b)) -> (Termin (while b S) sigma) -> Termin S sigma;
...

Discharge VAR;



Module WF Import TransClosure termination syntax;

[VAR|DecSetoid][gsort|SORT|VAR][S0:prog gsort]
  [E:Env|VAR][lsort|SORT|VAR];

$[myAccSort = AccSort gsort lsort E]
$[myState   = STATE gsort lsort]

[b:expression myAccSort bool][body:prog myAccSort];

[sofar [tau,sigma:myState] =
  TransClosure
        ([sigma,tau:myState](is_true (eval E sigma b)) ∧
                    (prog_operate S0 E sigma body tau)) sigma tau];

DischargeKeep E;

Goal WF_sofar :
  WF ([tau,sigma:myState]
      (Termin E S0 (while b body) sigma ∧ sofar E b body tau sigma));
...

Discharge VAR;



Module boperate Import lib_nat_Prop_rels lib_max_min operate;

(*  Corresponds to Figure 3.9 on page 94. *)
Inductive
  [prog_boperate
    : {lsort|SORT|VAR}{E:Env|VAR}
      {sigma:STATE gsort lsort}
      {S:prog (AccSort gsort lsort E)}{tau:STATE gsort lsort}
      {n:nat}Prop]
```

```
Relation NoReductions

Parameters
  [VAR|DecSetoid][gsort|SORT|VAR][S0:prog gsort]

Constructors
  [skip_boperate
    : {lsort|SORT|VAR}{E:Env|VAR}{sigma:STATE gsort lsort}{n:nat}
      prog_boperate E sigma (skip  (AccSort gsort lsort E)) sigma n]
  [assign_boperate
    : {lsort|SORT|VAR}{E:Env|VAR}{sigma:STATE gsort lsort}
      {x:VAR.DecSetoid_carrier}
      {t:expression  (AccSort gsort lsort E) (AccSort gsort lsort E x)}
      {n:nat}
      prog_boperate E sigma (assign x t)
        (STATE_update E x (eval E sigma t) sigma) n]
  [seq_boperate
    : {lsort|SORT|VAR}{E:Env|VAR}
      {S1,S2:prog (AccSort gsort lsort E)}
      {sigma,tau,eta:STATE gsort lsort}{n:nat}
      (prog_boperate E sigma S1 tau n) ->
      (prog_boperate E tau S2 eta n)->
      prog_boperate E sigma (seq S1 S2) eta n]
  [ifthen_boperate
    : {lsort|SORT|VAR}{E:Env|VAR}
      {b:expression  (AccSort gsort lsort E) bool}
      {S1,S2:prog (AccSort gsort lsort E)}{sigma,tau:STATE gsort lsort}{n:nat}
      (is_true (eval E sigma b)) ->
      (prog_boperate E sigma S1 tau n) ->
      prog_boperate E sigma (ifthenelse b S1 S2) tau n]
  [ifelse_boperate
    : {lsort|SORT|VAR}{E:Env|VAR}
      {b:expression  (AccSort gsort lsort E) bool}
      {S1,S2:prog (AccSort gsort lsort E)}{sigma,tau:STATE gsort lsort}{n:nat}
      (is_false (eval E sigma b)) ->
      (prog_boperate E sigma S2 tau n) ->
      prog_boperate E sigma (ifthenelse b S1 S2) tau n]
  [while_exit_boperate
    : {lsort|SORT|VAR}{E:Env|VAR}
       {b:expression  (AccSort gsort lsort E) bool}
       {body:prog (AccSort gsort lsort E)}{sigma:STATE gsort lsort}{n:nat}
      (is_false (eval E sigma b)) ->
      prog_boperate E sigma (while  b body) sigma n]
  [while_body_boperate
    : {lsort|SORT|VAR}{E:Env|VAR}
      {b:expression  (AccSort gsort lsort E) bool}
      {body:prog (AccSort gsort lsort E)}{sigma,tau,eta:STATE gsort lsort}
      {n:nat}
      (is_true (eval E sigma b)) ->
      (prog_boperate E sigma body tau n) ->
      (prog_boperate E tau (while b body) eta n) ->
      prog_boperate E sigma (while b body) eta n]
  [call_boperate
    : {lsort|SORT|VAR}{E:Env|VAR}{sigma,tau:STATE gsort lsort}{n:nat}
      (prog_boperate (Env_empty|VAR) sigma
         (coerce_prog (Env_empty_global gsort lsort) S0) tau n)->
      (prog_boperate E sigma (call  (AccSort gsort lsort E)) tau (suc n))]

  [new_boperate
    : {lsort|SORT|VAR}{E:Env|VAR}{T:Type(0)}
      {x:VAR.DecSetoid_carrier}
      {t:expression (AccSort gsort lsort E) T}
      {S:prog (sort_update x T (AccSort gsort lsort E))}
      {sigma:STATE gsort lsort}{n|nat}
      {tau:STATE gsort (sort_update x T lsort)}
      (prog_boperate (Env_add E x) (STATE_add_local E sigma x t)
        (coerce_prog (AccSort_update gsort T E lsort x) S) tau n) ->
      prog_boperate E sigma (new x t S)
                    (STATE_remove_local x (sigma.local x) tau) n];
```

```
Goal boperate_not_strict :
  {lsort|SORT|VAR}{E:Env|VAR}
  {sigma|STATE gsort lsort}
  {S|prog (AccSort gsort lsort E)}{tau|STATE gsort lsort}{n|nat}
  (prog_boperate E sigma S tau n) ->{m:nat}(Le n m) ->
  prog_boperate E sigma S tau m;
...

(*  Corresponds to Lemma 3.38 on page 93. *)
Goal operate2boperate :
  {lsort|SORT|VAR}{E:Env|VAR}
  {sigma|STATE gsort lsort}
  {S|prog (AccSort gsort lsort E)}
  {tau|STATE gsort lsort}
  {Prem:prog_operate S0 E sigma S tau}
  Ex [n:nat]prog_boperate E sigma S tau n;
...

Goal boperate2operate :
  {lsort|SORT|VAR}{E:Env|VAR}
  {sigma|STATE gsort lsort}
  {S|prog (AccSort gsort lsort E)}
  {tau|STATE gsort lsort}{n|nat}
  {Prem:prog_boperate E sigma S tau n}
        prog_operate S0 E sigma S tau;
...

Freeze Eq;

Goal boperate_zero :
  {lsort|SORT|VAR}{E:Env|VAR}{sigma,tau:STATE gsort lsort}
  not (prog_boperate E sigma (call  (AccSort gsort lsort E)) tau zero);
...

Discharge VAR;

Unfreeze Eq;



Module boperate_thms Import
  lib_bool_funs lib_nat_suc_thms boperate operate_thms syntax_thms;

[VAR|DecSetoid][gsort|SORT|VAR][S0|prog gsort];

Goal boper_Inversion :
  {lsort|SORT|VAR}{E:Env|VAR}
  {S:prog (AccSort gsort lsort E)}{sigma,tau:STATE gsort lsort}{n:nat}Prop;
...

Goal boper_inversion :
  {lsort|SORT|VAR}{E|Env|VAR}
  {sigma|STATE gsort lsort}{S|prog (AccSort gsort lsort E)}
  {tau|STATE gsort lsort}{n|nat}
  {D:prog_boperate S0 E sigma S tau n}
  (boper_Inversion E S sigma tau n);
...

Goal boper_ifthen :
  {lsort|SORT|VAR}{E|Env|VAR}
  {b|expression (AccSort gsort lsort E) bool}
  {S1,S2|prog (AccSort gsort lsort E)}{sigma,tau|STATE gsort lsort}{n|nat}
  (is_true (eval E sigma b))->
  (prog_boperate S0 E sigma (ifthenelse b S1 S2) tau n) ->
  prog_boperate S0 E sigma S1 tau n;
...

Goal boper_ifelse :
  {lsort|SORT|VAR}{E|Env|VAR}
```

```
  {b|expression (AccSort gsort lsort E) bool}
  {S1,S2|prog (AccSort gsort lsort E)}{sigma,tau|STATE gsort lsort}{n|nat}
(is_false (eval E sigma b)) ->
(prog_boperate S0 E sigma (ifthenelse b S1 S2) tau n) ->
prog_boperate S0 E sigma S2 tau n;
...

Goal boper_while :
  {lsort|SORT|VAR}{E|Env|VAR}
  {c|bool}{b|expression (AccSort gsort lsort E) bool}
  {S|prog (AccSort gsort lsort E)}
  {sigma,tau|STATE gsort lsort}{n|nat}
  (Eq (eval E sigma b) c)->
  (prog_boperate S0 E sigma (while b S) tau n) ->
  (and (is_false (eval E tau b))
       (if c (Ex [rho:STATE gsort lsort]
         and (prog_boperate S0 E sigma S rho n)
             (prog_boperate S0 E rho (while b S) tau n))
             (Eq sigma tau)));
...

Goal boper_while_false :
  {lsort|SORT|VAR}{E|Env|VAR}
  {b|expression (AccSort gsort lsort E) bool}
  {S|prog (AccSort gsort lsort E)}{sigma,tau|STATE gsort lsort}{n|nat}
  (prog_boperate S0 E sigma (while b S) tau n) ->
  is_false (eval E tau b);
...

(*  Corresponds to Lemma 3.39 on page 95. *)
Goal boper_determ :
  {lsort|SORT|VAR}{E|Env|VAR}
  {sigma|STATE gsort lsort}{S|prog (AccSort gsort lsort E)}
  {tau|STATE gsort lsort}{n|nat}
  {Prem_tau|prog_boperate S0 E sigma S tau n}
  {C:(STATE gsort lsort)->Prop}{eta:STATE gsort lsort}
  {m|nat}{Prem_eta:prog_boperate S0 E sigma S eta m}
  {C_eta:C eta}C tau;
...

DischargeKeep VAR;


Goal eval_under_coerc :
  {T | Type(0)}{x : DecSetoid_carrier VAR}{lsort1 | SORT|VAR}{E1 | Env|VAR}
  {x_global : is_false (E1 x)}{b : expression (AccSort gsort lsort1 E1) bool}
  {sigma1 : STATE gsort lsort1}[lsort'1 = sort_update x T lsort1]
  {v:T}
  {prog_eq' : EXTEQ (AccSort gsort lsort1 E1) (AccSort gsort lsort'1 E1)}
  Eq (eval E1 (STATE_add_local' sigma1 x v)
              (coerce_expression VAR (AccSort gsort lsort1 E1)
               (AccSort gsort lsort'1 E1) prog_eq' b))
     (eval E1 sigma1 b);
...

Goal eval_under_coerc' :
  {lsort,lsort' | SORT|VAR}{E | Env|VAR}
  {b : expression (AccSort gsort lsort E) bool}
  {sigma : STATE gsort lsort}
  {eq : EXTEQ lsort lsort'}{prog_eq : EXTEQ ??}
  Eq (eval E (STATE_coerce eq sigma)
              (coerce_expression VAR (AccSort gsort lsort E)
               (AccSort gsort lsort' E) prog_eq b))
     (eval E sigma b);
...

Goal AccSort_coerce :
  {lsort,lsort'|SORT VAR}{E|Env|VAR}
  {eq:EXTEQ lsort lsort'}
  EXTEQ (AccSort gsort lsort E) (AccSort gsort lsort' E);
```

```
...

Unfreeze Eq;

Goal boperate_coerce :
  {lsort|SORT VAR}{E|Env|VAR}
  {sigma|STATE gsort lsort}{S|prog (AccSort gsort lsort E)}
  {tau|STATE gsort lsort}{n|nat}
  {Der:prog_boperate S0 E sigma S tau n}
  {lsort':SORT VAR}{eq:EXTEQ lsort lsort'}
  prog_boperate S0 E (STATE_coerce eq sigma)
    (coerce_prog (AccSort_coerce eq) S) (STATE_coerce eq tau)
    n;
...




[U|Type(0)][y:VAR.DecSetoid_carrier][v:U];
[lsort|SORT|VAR][E|Env|VAR][yGlobal:is_false (E y)];

Goal prog_coerce_preserve_local :
  EXTEQ (AccSort gsort lsort E)
        (AccSort gsort (sort_update y U lsort) E);
...

Discharge lsort;

(*  Corresponds to Theorem 3.41 on page 95. *)
$Goal boper_preserve_local :
  {lsort|SORT VAR}{E:Env|VAR}
  {sigma|STATE gsort lsort}
  {S|prog (AccSort gsort lsort E)}
  {tau|STATE gsort lsort}{n|nat}
  {Prem:prog_boperate S0 E sigma S tau n}
  {yGlobal:is_false (E y)}
  prog_boperate S0 E (STATE_add_local' sigma y v)
    (coerce_prog (prog_coerce_preserve_local yGlobal) S)
    (STATE_add_local' tau y v) n;
...

(*  Corresponds to Lemma 3.40 on page 95. *)
Goal bcall_env_indep :
  {lsort|SORT|VAR}{E,E':Env|VAR}
  {sigma|STATE gsort lsort}{tau|STATE gsort lsort}{n|nat}
  {Der:prog_boperate S0 E  sigma (call ?) tau n}
  prog_boperate S0 E' sigma (call ?) tau n;
...

[TR : {E:Env|VAR}{lsort|SORT VAR}
         {sigma:STATE gsort lsort}
         {S:prog (AccSort gsort lsort E)}
         {tau:STATE gsort lsort}
       Prop]

[OPERLOCAL'
    = {T|Type(0)}{E|Env|VAR}{lsort|SORT VAR}{x:VAR.DecSetoid_carrier}
      [lsort' = sort_update x T lsort]
      {sigma,tau|STATE gsort lsort'}
      {v:lsort x}
      (TR (Env_add E x) sigma (call ?) tau) ->
      TR E (STATE_remove_local x v sigma) (call ?)
           (STATE_remove_local x v tau)];

[OPERLOCAL
    = {T|Type(0)}{E|Env|VAR}{lsort|SORT VAR}{x:VAR.DecSetoid_carrier}
      [lsort' = sort_update x T lsort]
      {sigma|STATE gsort lsort'}{tau|STATE gsort lsort}
      {v:lsort x}
```

```
      (TR (Env_add E x) (STATE_remove_local x v sigma) (call ?) tau) ->
      TR E sigma (call ?)
        (STATE_add_local' tau x (coerce (sort_update_lookup x T lsort)
  (local sigma x)))];

Discharge VAR;



Module operate_thms2 Import boperate_thms;

[VAR|DecSetoid][gsort|SORT|VAR][S0|prog gsort];

[y:VAR.DecSetoid_carrier][T|Type(0)][v:T];

Goal oper_preserve_local :
  {lsort|SORT VAR}{E:Env|VAR}
  {sigma|STATE gsort lsort}
  {S|prog (AccSort gsort lsort E)}
  {tau|STATE gsort lsort}
  {Prem:prog_operate S0 E sigma S tau}
  {yGlobal:is_false (E y)}
  prog_operate S0 E (STATE_add_local' sigma y v)
    (coerce_prog (prog_coerce_preserve_local y yGlobal) S)
    (STATE_add_local' tau y v);
...

Goal operate_coerce :
  {lsort|SORT VAR}{E|Env|VAR}
  {sigma|STATE gsort lsort}{S|prog (AccSort gsort lsort E)}
  {tau|STATE gsort lsort}
  {Der:prog_operate S0 E sigma S tau}
  {lsort':SORT VAR}{eq:EXTEQ lsort lsort'}
  prog_operate S0 E (STATE_coerce eq sigma)
    (coerce_prog (AccSort_coerce eq) S) (STATE_coerce eq tau);
...

Discharge T;

Unfreeze Eq;

(*  Corresponds to Corollary 3.42 on page 98. *)
Goal operate_resp_context :
  {lsort|SORT|VAR}
  {E|Env|VAR}
  {sigma|STATE gsort lsort}
  {S|prog (AccSort gsort lsort E)}
  {tau|STATE gsort lsort}
  {Der:prog_operate S0 E sigma S tau}
  {hidden:is_false (E y)}
  Eq (sigma.local y) (tau.local y);
...

Goal boper_local' :
  {n|nat}
  OPERLOCAL' ([E:Env|VAR][lsort|SORT VAR]
     [sigma:STATE gsort lsort]
     [S:prog (AccSort gsort lsort E)]
     [tau:STATE gsort lsort]prog_boperate S0 E sigma S tau n);
...

Goal boper_local :
  {n|nat}
  OPERLOCAL ([E:Env|VAR][lsort|SORT VAR]
     [sigma:STATE gsort lsort]
     [S:prog (AccSort gsort lsort E)]
     [tau:STATE gsort lsort]prog_boperate S0 E sigma S tau n);
...
```

```
Goal oper_local :
 OPERLOCAL ([E:Env|VAR][lsort|SORT VAR]
    [sigma:STATE gsort lsort]
    [S:prog (AccSort gsort lsort E)]
    [tau:STATE gsort lsort]prog_operate S0 E sigma S tau);
…

Goal oper_local' :
 OPERLOCAL' ([E:Env|VAR][lsort|SORT VAR]
    [sigma:STATE gsort lsort]
    [S:prog (AccSort gsort lsort E)]
    [tau:STATE gsort lsort]prog_operate S0 E sigma S tau);
…

Discharge VAR;


(* Contexts for Hoare Logic *)
Module Hcontext Import lib_prod lib_rel assertion;

[VAR|DecSetoid][gsort,lsort:SORT VAR]

[Spec [T:Type] =
  prod (Assertion gsort lsort T) (Assertion gsort lsort T)];

Inductive [HContext : Type] Theorems
Constructors
  [HNone:HContext]
  [HSome_internal:{T|Type}{spec:Spec T}HContext];

Goal HContext_disjoint : {T|Type}{spec:Spec T}not (Eq HNone (HSome_internal spec));
…

$Goal HContext_predicate : {P:Pred HContext}{O:HContext}Prop;
…

$Goal HSome_update :
  {T|Type}{O:HContext}{B|Type}{b:Spec B}{f:{A|Type}{a:Spec A}T}T;
…

Goal HContext_injective :
  {A,B|Type}{a|Spec A}{b|Spec B}(Eq (HSome_internal a) (HSome_internal b)) ->
  {P:{T|Type}{t:Spec T}Prop}(P b) -> P a;
…

Discharge gsort;

[gsort,lsort|SORT VAR];

[HSome = HSome_internal gsort lsort]
[HSomeSpec [T|Type][p,q:Assertion gsort lsort T] = HSome (Pair p q)];

DischargeKeep lsort;

Goal Hcontext_preserve :
  {U|Type(0)}
  {Gamma:HContext gsort lsort}{x:VAR.DecSetoid_carrier}{v:lsort x}
  HContext gsort (sort_update x U lsort);

Induction Gamma;

  intros; Refine HNone; (* empty context *)

  intros; Refine HSomeSpec;
    Refine +1 Assertion_preserve x spec.Fst v;
       Refine Assertion_preserve x spec.Snd v;
Save Hcontext_preserve;

Discharge VAR;
```

```
(* Semantics of Hoare logic *)
Module Hsemantics Import Hcontext operate;

[VAR|DecSetoid][gsort,lsort|SORT VAR]
[S0:prog gsort][E:Env|VAR][T|Type];

[HAtomicSem
  [p:Assertion gsort lsort T]
  [S:prog (AccSort gsort lsort E)]
  [q:Assertion gsort lsort T]
  = {z|T}{sigma|STATE gsort lsort}
    (p z sigma) ->
  Ex [tau:STATE gsort lsort]
  and (prog_operate S0 E sigma S tau) (q z tau)];

Goal HSemEx :
  {U|Type}
  {p:U->Assertion gsort lsort T}
  {S:prog (AccSort gsort lsort E)}
  {q:Assertion gsort lsort T}
  iff (HAtomicSem ([Z:T][sigma:STATE gsort lsort]
    Ex [t:U]p(t)(Z)(sigma)) S q)
      ({t:U}HAtomicSem (p(t)) S q);
...

DischargeKeep T;

(*  Corresponds to Definition 3.43 on page 98. *)
Goal HSem :
  {Gamma:HContext gsort lsort}
  {p:Assertion gsort lsort T}
  {S:prog (AccSort gsort lsort E)}
  {q:Assertion gsort lsort T}
  Prop;

Induction Gamma;

  Refine HAtomicSem; (* empty context *)

  intros;
  Refine (HAtomicSem spec.Fst (call ?) spec.Snd) ->
    HAtomicSem p S q;
Save HSem;

Discharge VAR;



Module H Import Hcontext syntax;

[VAR|DecSetoid][gsort|SORT VAR][S0:prog gsort];

(*  Corresponds to Figure 3.10 on page 101. *)
Inductive
  [HD :
    {E:Env|VAR}{lsort|SORT VAR}{T|Type(0)}
    {Gamma:HContext gsort lsort}
    {p:Assertion gsort lsort T}
    {S:prog (AccSort gsort lsort E)}
    {q:Assertion gsort lsort T}
    Prop]

  Relation NoReductions

  Constructors

  [HDAssumption :
    {E:Env|VAR}{lsort|SORT|VAR}{T|Type}{p,q:Assertion gsort lsort T}
    HD E (HSomeSpec p q) p (call (AccSort gsort lsort E)) q]

  [HDInv :
```

```
  {E:Env|VAR}{lsort|SORT|VAR}{T|Type}
  {Gamma:HContext gsort lsort}{p:Assertion gsort lsort T}
  HD E Gamma p (skip (AccSort gsort lsort E)) p]

[HDAssign :
  {E:Env|VAR}{lsort|SORT|VAR}{T|Type}
  {Gamma:HContext gsort lsort}{p:Assertion gsort lsort T}
  {x:VAR.DecSetoid_carrier}
  {t:expression (AccSort gsort lsort E) (AccSort gsort lsort E x)}
  HD E Gamma (Assertion_update E x t p) (assign x t) p]

[HDSeq :
  {E|Env|VAR}{lsort|SORT VAR}{T|Type}
  {Gamma:HContext gsort lsort}
  {p,q,r:Assertion gsort lsort T}
  {S1,S2|prog (AccSort gsort lsort E)}
  {Seq_prem1:HD E Gamma p S1 r}{Seq_prem2:HD E Gamma r S2 q}
  HD E Gamma p (seq S1 S2) q]

[HDIfthenelse :
  {E|Env|VAR}{lsort|SORT VAR}{T|Type}
  {Gamma:HContext gsort lsort}
  {p,q:Assertion gsort lsort T}
  {S1,S2|prog (AccSort gsort lsort E)}
  {b:expression (AccSort gsort lsort E) bool}
  {Ifthenelse_prem1:
    HD E Gamma ([Z:T][sigma:STATE gsort lsort]
 (p Z sigma) ∧ (is_true (eval E sigma b)))
S1 q}
  {Ifthenelse_prem2:
    HD E Gamma ([Z:T][sigma:STATE gsort lsort]
 (p Z sigma) ∧ (is_false (eval E sigma b)))
                      S2 q}
  HD E Gamma p (ifthenelse b S1 S2) q]

[HDWhile :
  {E|Env|VAR}{lsort|SORT VAR}{T|Type}
  {Gamma:HContext gsort lsort}
  {p:Assertion gsort lsort T}
  {b:expression (AccSort gsort lsort E) bool}
  {W|Type}{Lt':Rel W W}{is_wf : WF Lt'}
  {u:(STATE gsort lsort)->W} (* convergence function *)
  {body:prog (AccSort gsort lsort E)}
  {While_step:{t:W}
    HD E Gamma ([Z:T][sigma:STATE gsort lsort]
 (p Z sigma) ∧ (is_true(eval(E)(sigma)(b))) ∧
 (Eq (u sigma) t))
    body ([Z:T][tau:STATE gsort lsort](p Z tau) ∧ (Lt' (u tau) t))}
 HD E Gamma p (while b body)
  ([Z:T][tau:STATE gsort lsort](p Z tau) ∧ (is_false(eval E tau b)))]

[HDNew :
  {E|Env|VAR}{lsort|SORT|VAR}{T,U|Type}
  {Gamma:HContext gsort lsort}{p,q:Assertion gsort lsort T}
  {x:VAR.DecSetoid_carrier}{t:expression (AccSort gsort lsort E) U}
  {S:prog (sort_update x U (AccSort gsort lsort E))}
  {NewPrem :
    {v:lsort x}
    HD (Env_add E x) (Hcontext_preserve Gamma x v)
       ([Z:T][sigma:STATE gsort (sort_update x U lsort)]
 (Fr x p v Z sigma) ∧
 (Eq (coerce (sort_update_lookup x U lsort) (sigma.local x))
   (eval E (STATE_remove_local x v sigma) t)))
    (coerce_prog (AccSort_update gsort U E lsort x) S)
    (Fr x q v)}
  HD E Gamma p (new x t S) q]

[HDCall :
  {E|Env|VAR}{lsort|SORT|VAR}{T|Type}
  {W|Type}{Lt':Rel W W}{is_wf : WF Lt'}
```

```
    {p|W->Assertion gsort lsort T}
    {q:Assertion gsort lsort T}
    {Call_step:{t:W}
      HD (Env_empty|VAR) (HSomeSpec
          ([Z:T][sigma:STATE gsort lsort]
    Ex [u:W](p(u)(Z)(sigma)) ∧ (Lt' u t))
   q)
      (p(t))
         (coerce_prog (Env_empty_global gsort lsort) S0) q}
   HD E (HNone gsort lsort)
      ([Z:T][sigma:STATE gsort lsort]Ex [t:W]p(t)(Z)(sigma))
      (call (AccSort gsort lsort E)) q]

  [HDCon :
    {E|Env|VAR}{lsort|SORT VAR}{T,U|Type}
    {Gamma:HContext gsort lsort}
    {p,q:Assertion gsort lsort T}{pre,post:Assertion gsort lsort U}
    {S:prog (AccSort gsort lsort E)}
    {Prem : KSC (p,q) (pre,post)}
    {prem : HD E Gamma pre S post}
    HD E Gamma p S q];

Discharge VAR;
```

```
Module Htheorems Import H;

[VAR|DecSetoid][gsort|SORT VAR][S0:prog gsort]
[E:Env|VAR][lsort|SORT VAR][T|Type]
[Gamma:HContext gsort lsort];

[p,q:Assertion gsort lsort T];

Goal HDConl :
  {pre|Assertion gsort lsort T}{S|prog (AccSort gsort lsort E)}
  {Prem: SubRel p pre}
  {prem : HD S0 E Gamma pre S q}HD S0 E Gamma p S q;
...

Goal HDConr :
  {post|Assertion gsort lsort T}{S|prog (AccSort gsort lsort E)}
  {Prem: SubRel post q}
  {prem: HD S0 E Gamma p S post}HD S0 E Gamma p S q;
...

DischargeKeep p;

Goal HDInv_lemma :  (SubRel p q) -> HD S0 E Gamma p (skip ?)  q;
...

$[sort = AccSort gsort lsort E];

Goal HDAssign_lemma :
  {x:VAR.DecSetoid_carrier}{t:expression (AccSort gsort lsort E) (sort x)}
  (SubRel q
          ([Z:T][sigma:STATE gsort lsort]
           p Z (STATE_update E x (eval E sigma t) sigma))) ->
  HD S0 E Gamma q (assign x t) p;
...

Discharge VAR;
```

```
Module Hsound Import Htheorems Hsemantics operate_thms2;

[VAR|DecSetoid][gsort|SORT|VAR][S0:prog gsort]
```

```
[E:Env|VAR][lsort|SORT|VAR][T|Type][Gamma:HContext gsort lsort]

[p,r,q:Assertion gsort lsort T]
[b:expression (AccSort gsort lsort E) bool]
[S1,S2:prog (AccSort gsort lsort E)];

Goal HDAssumptionSound :
  HSem S0 E (HSomeSpec p q) p (call (AccSort gsort lsort E)) q;
...

$Goal HDInvAtomicSound :
  HAtomicSem S0 E p (skip (AccSort gsort lsort E)) p;
...

Goal HDInvSound :
  HSem S0 E Gamma  p (skip (AccSort gsort lsort E)) p;
...

$Goal HDAssignAtomicSound :
  {x:VAR.DecSetoid_carrier}
  {t:expression (AccSort gsort lsort E) (AccSort gsort lsort E x)}
  HAtomicSem S0 E (Assertion_update E x t p) (assign x t) p;
...

Goal HDAssignSound :
  {x:VAR.DecSetoid_carrier}
  {t:expression (AccSort gsort lsort E) (AccSort gsort lsort E x)}
  HSem S0 E Gamma (Assertion_update E x t p) (assign x t) p;
...

$Goal HDSeqAtomicSound :
  {Seq_prem1:HAtomicSem S0 E p S1 r}
  {Seq_prem2:HAtomicSem S0 E r S2 q}
  HAtomicSem S0 E p (seq S1 S2) q;
...

Goal HDSeqSound :
  {Seq_prem1:HSem S0 E Gamma p S1 r}
  {Seq_prem2:HSem S0 E Gamma r S2 q}
  HSem S0 E Gamma p (seq S1 S2) q;
...

$Goal HDIfthenelseAtomicSound :
  {Ifthenelse_prem1:
      HAtomicSem S0 E ([Z:T][sigma:STATE gsort lsort]
   (p Z sigma) ∧ (is_true (eval E sigma b)))
 S1 q}
  {Ifthenelse_prem2:
    HAtomicSem S0 E ([Z:T][sigma:STATE gsort lsort]
      (p Z sigma) ∧ (is_false (eval E sigma b)))
                        S2 q}
    HAtomicSem S0 E p (ifthenelse b S1 S2) q;
...

Goal HDIfthenelseSound :
  {Ifthenelse_prem1:
      HSem S0 E Gamma ([Z:T][sigma:STATE gsort lsort]
   (p Z sigma) ∧ (is_true (eval E sigma b)))
 S1 q}
  {Ifthenelse_prem2:
    HSem S0 E Gamma ([Z:T][sigma:STATE gsort lsort]
      (p Z sigma) ∧ (is_false (eval E sigma b)))
                        S2 q}
    HSem S0 E Gamma p (ifthenelse b S1 S2) q;
...

$Goal HDWhileAtomicSound :
  {W|Type}{Lt':Rel W W}{is_wf : WF Lt'}
  {u:(STATE gsort lsort)->W}
```

```
  {body:prog (AccSort gsort lsort E)}
  {While_step:{t:W}
      HAtomicSem S0 E ([Z:T][sigma:STATE gsort lsort]
   (p Z sigma) ∧ (is_true(eval(E)(sigma)(b))) ∧
   (Eq (u sigma) t))
      body ([Z:T][tau:STATE gsort lsort](p Z tau) ∧ (Lt' (u tau) t))}
   HAtomicSem S0 E p (while b body)
    ([Z:T][tau:STATE gsort lsort](p Z tau) ∧ (is_false(eval E tau b)));
…

Goal HDWhileSound :
{W|Type}{Lt':Rel W W}{is_wf : WF Lt'}
  {u:(STATE gsort lsort)->W}
  {body:prog (AccSort gsort lsort E)}
  {While_step:{t:W}
      HSem S0 E Gamma ([Z:T][sigma:STATE gsort lsort]
   (p Z sigma) ∧ (is_true(eval(E)(sigma)(b))) ∧
   (Eq (u sigma) t))
      body ([Z:T][tau:STATE gsort lsort](p Z tau) ∧ (Lt' (u tau) t))}
   HSem S0 E Gamma p (while b body)
    ([Z:T][tau:STATE gsort lsort](p Z tau) ∧ (is_false(eval E tau b)));
…

Goal HDNewSound :
  {U|Type}
  {x:VAR.DecSetoid_carrier}{t:expression (AccSort gsort lsort E) U}
  {S:prog (sort_update x U (AccSort gsort lsort E))}
  {NewPrem :
    {v:lsort x}
    HSem S0 (Env_add E x) (Hcontext_preserve Gamma x v)
    ([Z:T][sigma:STATE gsort (sort_update x U lsort)]
      (Fr x p v Z sigma) ∧
      (Eq (coerce (sort_update_lookup x U lsort) (sigma.local x))
(eval E (STATE_remove_local x v sigma) t)))
    (coerce_prog (AccSort_update gsort U E lsort x) S)
    (Fr x q v)}
  HSem S0 E Gamma p (new x t S) q;
…

Goal HDCallSound :
  {W|Type}{Lt':Rel W W}{is_wf : WF Lt'}
    {p|W->Assertion gsort lsort T}
    {Call_step:{t:W}
      HSem S0 (Env_empty|VAR) (HSomeSpec
          ([Z:T][sigma:STATE gsort lsort]
    Ex [u:W](p(u)(Z)(sigma)) ∧ (Lt' u t))
   q)
      (p(t))
         (coerce_prog (Env_empty_global gsort lsort) S0) q}
   HSem S0 E (HNone gsort lsort)
      ([Z:T][sigma:STATE gsort lsort]Ex [t:W]p(t)(Z)(sigma))
      (call (AccSort gsort lsort E)) q;
…

$Goal HDConAtomicSound :
  {U|Type}{pre,post:Assertion gsort lsort U}
  {S:prog (AccSort gsort lsort E)}
  {Prem : KSC (p,q) (pre,post)}
  {prem : HAtomicSem S0 E pre S post}
    HAtomicSem S0 E p S q;
…

Goal HDConSound :
  {U|Type}{pre,post:Assertion gsort lsort U}
  {S:prog (AccSort gsort lsort E)}
  {Prem : KSC (p,q) (pre,post)}
  {prem : HSem S0 E Gamma pre S post}
    HSem S0 E Gamma p S q;
…
```

```
Discharge E;

(*  Corresponds to Theorem 3.44 on page 102. *)
Goal HSound :
  {E|Env|VAR}{lsort|SORT VAR}{T|Type(0)}
  {Gamma|HContext gsort lsort}
  {p|Assertion gsort lsort T}
  {S|prog (AccSort gsort lsort E)}
  {q|Assertion gsort lsort T}
  {der:HD S0 E Gamma p S q}HSem S0 E Gamma p S q;
...


Discharge VAR;




Module Hcomplete Import Hsemantics Htheorems WF operate_thms2;

[VAR|DecSetoid][gsort|SORT VAR][S0:prog gsort];

[Hoarep0
  [lsort|SORT|VAR][E:Env|VAR][S:prog (AccSort gsort lsort E)][n:nat] =
  [Z,sigma:STATE gsort lsort]prog_boperate S0 E sigma S Z n];

[psi [lsort:SORT|VAR][E:Env|VAR][n:nat]
  = Hoarep0 E (call (AccSort gsort lsort E)) (suc n)]

[p [lsort:SORT|VAR][E:Env|VAR][n:nat][Z,sigma:STATE gsort lsort]
  = Ex ([u:nat](psi lsort E u Z sigma ∧ Lt u n))];

Discharge gsort;

Freeze Eq;

(* Corresponds to Theorem 3.45 on page 104.*)
Goal completeness_step :
  {gsort,lsort|SORT|VAR}
  {S0:prog gsort}{n:nat}
  HD S0 (Env_empty|VAR)
  (HSomeSpec (p S0 lsort (Env_empty|VAR) n) (Eq|(STATE ??)))
  (psi S0 lsort (Env_empty|VAR) n)
  (coerce_prog (Env_empty_global gsort lsort) S0)
  (Eq|(STATE gsort lsort));
...

(*  Corresponds to Theorem 3.46 on page 107. *)
Goal HL_most_general :
   {gsort,lsort|SORT|VAR} {E:Env|VAR}{S:prog (AccSort gsort lsort E)}
   {S0:prog gsort}
    HD S0 E (HNone ??) ([Z,sigma:STATE gsort lsort]
        prog_operate S0 E sigma S Z)
       S (Eq|(STATE gsort lsort));
...

(*  Corresponds to Corollary 3.47 on page 107. *)
Goal HL_complete :
  {gsort,lsort|SORT|VAR}{E:Env|VAR}{S0:prog gsort}
  {S|prog (AccSort gsort lsort E)}{T|Type}
  {p,q|Assertion gsort lsort T}
  {semant:HSem S0 E (HNone ??) p S q}HD S0 E (HNone ??) p S q;
...

Discharge VAR;

Unfreeze Eq;
```

```
Module Hadmiss Import Hsound Hcomplete;

[VAR|DecSetoid][gsort|SORT|VAR][S0:prog gsort]

[T|Type][lsort|SORT|VAR]
[E:Env|VAR][q|Assertion gsort lsort T];

(*  Corresponds to Corollary 3.48 on page 107. *)
Goal HDNRCall :
  {p|Assertion gsort lsort T}{Prem:HD S0 (Env_empty|VAR) (HNone ??) p
    (coerce_prog (Env_empty_global gsort lsort) S0) q}
  HD S0 E (HNone gsort lsort) p (call ?) q;
...

[Gamma|HContext gsort lsort][U|Type(0)];

Goal Adaptation :
  {pre,post:Assertion gsort lsort U}{S:prog ?}
  {prem : HD S0 E Gamma pre S post}
  [p [Z:T][sigma:STATE gsort lsort] =
    {tau:STATE gsort lsort}Ex [Z':U]and (pre Z' sigma)
                                ((post Z' tau) -> q Z tau)]
  HD S0 E Gamma p S q;
...

Goal HDNew_LC :
  {x:VAR.DecSetoid_carrier}{t:expression ? U}
  [sort = sort_update x U lsort]
  {p:Assertion gsort sort T}{S:prog ?}
  {NewPrem : {v:lsort x}
    [E = Env_add E x]
    HD S0 E (Hcontext_preserve Gamma x v) p
         (coerce_prog (AccSort_update gsort ?? lsort x) S)
         (Fr x q v)}
  HD S0 E Gamma ([Z:T][sigma:STATE ??]p Z (STATE_add_local E sigma x t))
   (new x t S) q;
...

Discharge VAR;




Module static Import Hadmiss;

Inductive [myVAR:Type] Double
Constructors [X,Z:myVAR];

Goal myVAR_eq : myVAR -> myVAR -> bool;
...

Goal myVAR_eq_refl : {x:myVAR}is_true (myVAR_eq x x);
...

Goal myVAR_eq_char : Eqchar myVAR_eq;
...

Goal myVAR_DecSetoid : DecSetoid;
...

(*  Corresponds to Example 3.31 on page 87. *)
[mysort = [_:myVAR]nat : myVAR_DecSetoid.DecSetoid_carrier -> Type]

[S0 = assign|myVAR_DecSetoid X ([sigma:FS mysort]sigma(Z))
    : prog|myVAR_DecSetoid mysort];

Goal myprog : prog|myVAR_DecSetoid mysort;
Refine seq; Refine assign; Refine Z; Intros; Refine one;
Refine new; Refine +1 Z; Intros +1; Refine zero; Refine call;
```

```
Save myprog;

[E:Env|myVAR_DecSetoid]
[E' [x:myVAR_DecSetoid.DecSetoid_carrier] =
  (andalso (inv (myVAR_DecSetoid.DecSetoid_eq x X))
    (andalso (inv (myVAR_DecSetoid.DecSetoid_eq x Z)) (E x)))];

Goal myprog_coerce : EXTEQ mysort (AccSort mysort mysort E');
...

Goal Xcoerce :
  Eq (AccSort%%myVAR_DecSetoid mysort mysort E' X) (mysort X);
...

[post [Z,sigma:STATE mysort mysort] =
  Eq (coerce Xcoerce (lookup(E')(sigma)(X))) one];

(*  Corresponds to Lemma 3.49 on page 109. *)
Goal myprog_correct :
  HD S0 E' (HNone ??) ([_,_:STATE ??]trueProp)
  (coerce_prog myprog_coerce myprog) post;
...



(* Contexts for VDM *)
Module VDMcontext Import lib_prod lib_rel state;

[VAR|DecSetoid][gsort,lsort:SORT VAR]

[VDMSpec =
  prod (Pred (STATE gsort lsort))
       (Rel (STATE gsort lsort) (STATE gsort lsort))];

Inductive [VDMContext : Type] Theorems
Constructors
  [VDMNone:VDMContext]
  [VDMSome_internal:{spec:VDMSpec}VDMContext];

Discharge gsort;

[gsort,lsort|SORT VAR];

[VDMSome = VDMSome_internal gsort lsort]
[VDMSomeSpec
  [p:Pred (STATE gsort lsort)]
  [q:Rel (STATE gsort lsort) (STATE gsort lsort)] =
  VDMSome (Pair p q)];

DischargeKeep lsort;

[U|Type(0)]
[x:VAR.DecSetoid_carrier]
[v:lsort x]
$[lsort' = sort_update x U lsort]
$[STATE' = STATE gsort lsort'];

[VDMPr [q:Rel (STATE gsort lsort) (STATE gsort lsort)] =
  [tau,sigma:STATE'](q (STATEuL x v tau) (STATEuL x v sigma)) ∧
                      (Eq (tau.local x) (sigma.local x))];

Goal VDMcontextPr :
  {Gamma:VDMContext gsort lsort}
  VDMContext gsort (sort_update x U lsort);

Induction Gamma;

  intros; Refine VDMNone; (* empty context *)

  intros; Refine VDMSomeSpec;
```

```
    Intros sigma; Refine spec.Fst (STATEuL x v sigma);
    Refine  VDMPr spec.Snd;
Save VDMcontextPr;

Discharge VAR;



Module  VDMsemantics Import VDMcontext operate;

[VAR|DecSetoid][gsort,lsort|SORT VAR]
[S0:prog gsort][E:Env|VAR];

[VDMAtomicSem
  [p:Pred (STATE gsort lsort)]
  [S:prog (AccSort gsort lsort E)]
  [q:Rel (STATE gsort lsort) (STATE gsort lsort) ]
  = {sigma|STATE gsort lsort}
    (p sigma) ->
  Ex [tau:STATE gsort lsort]
  and (prog_operate S0 E sigma S tau) (q tau sigma)];

Goal VDMSemEx :
  {U|Type}
  {p:U->Pred (STATE gsort lsort)}
  {S:prog (AccSort gsort lsort E)}
  {q:Rel (STATE gsort lsort) (STATE gsort lsort)}
  iff (VDMAtomicSem ([sigma:STATE gsort lsort]
    Ex [t:U]p(t)(sigma)) S q)
      ({t:U}VDMAtomicSem (p(t)) S q);
...

Goal VDMSem :
  {Gamma:VDMContext gsort lsort}
  {p:Pred (STATE gsort lsort)}
  {S:prog (AccSort gsort lsort E)}
  {q:Rel (STATE gsort lsort) (STATE gsort lsort)}
  Prop;

Induction Gamma;

  Refine VDMAtomicSem; (* empty context *)

  intros;
  Refine (VDMAtomicSem spec.Fst (call ?) spec.Snd) ->
    VDMAtomicSem p S q;
Save VDMSem;

Discharge VAR;



Module  VDM Import tms_rel VDMcontext operate;

[VAR|DecSetoid][gsort|SORT VAR][S0:prog gsort];

[lsort|SORT VAR];

[VDM_Pred_to_Rel [P:Pred (STATE gsort lsort)]
  = [sigma,_:STATE gsort lsort]P sigma
  : Rel (STATE gsort lsort) (STATE gsort lsort)];

Discharge lsort;

(*  Corresponds to Figure 3.12 on page 111. *)
Inductive
  [VDMder :
    {E:Env|VAR}{lsort|SORT VAR}
    {Gamma:VDMContext gsort lsort}
    {P:Pred (STATE gsort lsort)}
    {S:prog (AccSort gsort lsort E)}
```

```
    {Q:Rel (STATE gsort lsort) (STATE gsort lsort)}
    Prop]

Relation NoReductions

Constructors
  [VDMAssumption :
    {E:Env|VAR}{lsort|SORT VAR}
    {p:Pred (STATE gsort lsort)}
    {q:Rel (STATE gsort lsort) (STATE gsort lsort)}
    VDMder E (VDMSomeSpec p q) p (call (AccSort gsort lsort E)) q]

  [VDMSkip :
    {E:Env|VAR}{lsort|SORT VAR}
    {Gamma:VDMContext gsort lsort}
    VDMder E Gamma ([_:(STATE gsort lsort)]trueProp)
    (skip (AccSort gsort lsort E))
      (Eq|(STATE gsort lsort))]

  [VDMAssign :
    {E:Env|VAR}{lsort|SORT VAR}
    {Gamma:VDMContext gsort lsort}
    {x:VAR.DecSetoid_carrier}
    {t:expression (AccSort gsort lsort E) (AccSort gsort lsort E x)}
    VDMder E Gamma ([_:(STATE gsort lsort)]trueProp) (assign x t)
      ([tau,sigma:STATE gsort lsort]
Eq tau (STATE_update E x (eval E sigma t) sigma))]

  [VDMSeq :
    {E:Env|VAR}{lsort|SORT VAR}
    {Gamma:VDMContext gsort lsort}
    {p1,p2:(STATE gsort lsort).Pred}
    {r1,r2:Rel (STATE gsort lsort) (STATE gsort lsort)}
    {S1,S2:prog (AccSort gsort lsort E)}
    (VDMder E Gamma p1 S1 ([tau,sigma:(STATE gsort lsort)]
  and (p2 tau) (r1 tau sigma))) ->
    (VDMder E Gamma p2 S2 r2) ->
    VDMder E Gamma p1 (seq S1 S2) (composeRel r2 r1)]

  [VDMIfthenelse :
    {E:Env|VAR}{lsort|SORT VAR}
    {Gamma:VDMContext gsort lsort}
    {pre:(STATE gsort lsort).Pred}
    {post:Rel (STATE gsort lsort) (STATE gsort lsort)}
    {b:expression (AccSort gsort lsort E) bool}
    {TH,EL:prog (AccSort gsort lsort E)}
    (VDMder E Gamma ([sigma:(STATE gsort lsort)]
    (pre sigma) ∧ (is_true (eval E sigma b)))
            TH post) ->
    (VDMder E Gamma ([sigma:(STATE gsort lsort)]
    (pre sigma) ∧ (is_false (eval E sigma b)))
            EL post) ->
    VDMder E Gamma pre (ifthenelse b TH EL) post]

  [VDMWhile :
    {E:Env|VAR}{lsort|SORT VAR}
    {Gamma:VDMContext gsort lsort}
    {P:(STATE gsort lsort).Pred}
    {sofar:Rel (STATE gsort lsort) (STATE gsort lsort)}
    {sofar_trans: trans sofar}
    {sofar_wf: WF (andRel sofar (VDM_Pred_to_Rel P))}
    {b:expression (AccSort gsort lsort E) bool}
    {body:prog (AccSort gsort lsort E)}
    {Prem:
      VDMder E Gamma ([sigma:STATE gsort lsort]
       (P sigma) ∧ (is_true (eval E sigma b)))
              body
        ([tau,sigma:STATE gsort lsort]
  (P tau) ∧ (sofar tau sigma))}
   VDMder E Gamma P (while b body)
```

```
          ([tau,sigma:STATE gsort lsort]
    (P tau) ∧ (is_false (eval E tau b)) ∧
                          (reflclose sofar tau sigma))]

  [VDMNew :
    {E|Env|VAR}{lsort|SORT VAR}{U|Type}
    {Gamma:VDMContext gsort lsort}
    {p:Pred (STATE gsort lsort)}
    {q:Rel (STATE gsort lsort) (STATE gsort lsort)}
    {x:VAR.DecSetoid_carrier}{t:expression (AccSort gsort lsort E) U}
    {S:prog (sort_update x U (AccSort gsort lsort E))}
    {NewPrem :
      {v:lsort x}
      VDMder (Env_add E x) (VDMcontextPr x v Gamma)
         ([sigma:STATE gsort (sort_update x U lsort)]
   (p (STATEuL x v sigma)) ∧
   (Eq (coerce (sort_update_lookup x U lsort) (sigma.local x))
     (eval E (STATE_remove_local x v sigma) t)))
      (coerce_prog (AccSort_update gsort U E lsort x) S)
      ([tau,sigma:STATE gsort (sort_update x U lsort)]
q (STATEuL x v tau) (STATEuL x v sigma))}
    VDMder E Gamma p (new x t S) q]

  [VDMCall :
    {E:Env|VAR}{lsort|SORT VAR}
    {W|Type}{Lt':Rel W W}{is_wf : WF Lt'}
    {p:W->Pred (STATE gsort lsort)}
    {q:Rel (STATE gsort lsort) (STATE gsort lsort)}
    {Call_step:{t:W}
      VDMder (Env_empty|VAR) (VDMSomeSpec
          ([sigma:STATE gsort lsort]
    Ex [u:W](p(u)(sigma)) ∧ (Lt' u t))
   q)
      (p(t))
         (coerce_prog (Env_empty_global gsort lsort) S0) q}
   VDMder E (VDMNone gsort lsort)
      ([sigma:STATE gsort lsort]Ex [t:W]p(t)(sigma))
      (call (AccSort gsort lsort E)) q]

  [VDMConsequence :
    {E:Env|VAR}{lsort|SORT VAR}
    {Gamma:VDMContext gsort lsort}
    {p,p1:Pred (STATE gsort lsort)}
    {q,q1:Rel (STATE gsort lsort) (STATE gsort lsort)}
    {S:prog (AccSort gsort lsort E)}
    {ConsSC1:{sigma|STATE gsort lsort}(p sigma) -> (p1 sigma)}
    {ConsSC2:{tau,sigma|STATE gsort lsort}(p sigma) ->
    (q1 tau sigma) -> q tau sigma}
    {ConsPrem:VDMder E Gamma p1 S q1}
    VDMder E Gamma p S q];

Discharge VAR;



Module  VDMtheorems Import VDM ;

[VAR|DecSetoid][gsort|SORT VAR][S0:prog gsort]
[E:Env|VAR][lsort|SORT VAR]
[Gamma:VDMContext gsort lsort];

Goal VDMPre :
  {Gamma:VDMContext gsort lsort}
  {pre|(STATE gsort lsort).Pred}
  {S|prog (AccSort gsort lsort E)}
  {post|Rel (STATE gsort lsort) (STATE gsort lsort)}
  {Prem:VDMder S0 E Gamma pre S post}
  VDMder S0 E Gamma pre S
    [tau,sigma:(STATE gsort lsort)]and (pre sigma) (post tau sigma);
...
```

```
Goal VDMConsequenceLeft :
  {Gamma:VDMContext gsort lsort}
  {pre_s,pre : (STATE gsort lsort).Pred}
  {post:Rel (STATE gsort lsort) (STATE gsort lsort)}
  {S:prog (AccSort gsort lsort E)}
  {strengthen : {sigma:(STATE gsort lsort)}(pre_s sigma) -> pre sigma}
  {prem : VDMder S0 E Gamma pre S post}
  VDMder S0 E Gamma pre_s S post;
…

Goal VDMConsequenceRight:
  {Gamma:VDMContext gsort lsort}
  {pre : (STATE gsort lsort).Pred}
  {post,post_w:Rel (STATE gsort lsort) (STATE gsort lsort)}
  {S:prog (AccSort gsort lsort E)}
  {weaken :
    {sigma,tau:(STATE gsort lsort)}(post tau sigma) -> post_w tau sigma}
  {prem : VDMder S0 E Gamma pre S post}
  VDMder S0 E Gamma pre S post_w;
…

Goal VDMSkipBU :
  {p:Rel (STATE gsort lsort) (STATE gsort lsort)}{p':(STATE gsort lsort).Pred}
  {p'p:{sigma:(STATE gsort lsort)}(p' sigma) -> p sigma sigma}
  VDMder S0 E Gamma p' (skip ?) p;
…

Goal VDMAssignBU :
  {x:VAR.DecSetoid_carrier}
  {t:expression (AccSort gsort lsort E) (AccSort gsort lsort E x)}
  {p:Rel (STATE gsort lsort) (STATE gsort lsort)}
  {p':(STATE gsort lsort).Pred}
  {p'p:
    {sigma,tau:(STATE gsort lsort)}
    (p' sigma) ->
   p (STATE_update E x (eval E sigma t) tau) sigma}
  VDMder S0 E Gamma p' (assign x t) p;
…

Goal VDMSeqBU :
  {Gamma:VDMContext gsort lsort}
  {p1,p2:(STATE gsort lsort).Pred}
  {r,r1,r2:Rel (STATE gsort lsort) (STATE gsort lsort)}
  {S1,S2:prog (AccSort gsort lsort E)}
  {weakening: SubRel (composeRel r2 r1) r}
  (VDMder S0 E Gamma p1 S1
    [tau,sigma:(STATE gsort lsort)]and (p2 tau) (r1 tau sigma)) ->
  (VDMder S0 E Gamma p2 S2 r2) ->
  VDMder S0 E Gamma p1 (seq S1 S2) r;
…

Goal VDMLoopBU :
   {P,I:(STATE gsort lsort).Pred}
   {Q,sofar:Rel (STATE gsort lsort) (STATE gsort lsort)}
     {sofar_trans: trans sofar}
     {sofar_wf: WF (andRel sofar (VDM_Pred_to_Rel I))}
   {b:expression ? bool}{body:prog ?}
   {ScPre:SubPred P I}
   {ScPost:{tau,sigma:(STATE gsort lsort)}(I sigma) -> (I tau) ->
           (is_false (eval E tau b)) ->
   (reflclose sofar tau sigma) -> Q tau sigma}
   {Prem:
     VDMder S0 E Gamma ([sigma:(STATE gsort lsort)](I sigma) ∧
      (is_true (eval E sigma b)))
                body
        ([tau,sigma:(STATE gsort lsort)](I tau) ∧ (sofar tau sigma))}
   VDMder S0 E Gamma P (while b body) Q;
…

Discharge VAR;
```

```
Module VDMsound Import VDMsemantics VDM operate_thms2;

[VAR|DecSetoid][gsort|SORT|VAR][S0:prog gsort]

[E:Env|VAR][lsort|SORT|VAR][Gamma:VDMContext gsort lsort]

[p:Pred (STATE gsort lsort)][q:Rel (STATE gsort lsort) (STATE gsort lsort)]
[b:expression (AccSort gsort lsort E) bool]
[S1,S2:prog (AccSort gsort lsort E)];

Goal VDMAssumptionSound :
  VDMSem S0 E (VDMSomeSpec p q) p (call (AccSort gsort lsort E)) q;
...

$Goal VDMInvAtomicSound :
  VDMAtomicSem S0 E ([_:(STATE gsort lsort)]trueProp)
    (skip (AccSort gsort lsort E))
      (Eq|(STATE gsort lsort));
...

Goal VDMInvSound :
  VDMSem S0 E Gamma ([_:(STATE gsort lsort)]trueProp)
    (skip (AccSort gsort lsort E))
      (Eq|(STATE gsort lsort));
...

$Goal VDMAssignAtomicSound :
  {x:VAR.DecSetoid_carrier}
  {t:expression (AccSort gsort lsort E) (AccSort gsort lsort E x)}
  VDMAtomicSem S0 E ([_:(STATE gsort lsort)]trueProp) (assign x t)
      ([tau,sigma:STATE gsort lsort]
Eq tau (STATE_update E x (eval E sigma t) sigma));
...

Goal VDMAssignSound :
  {x:VAR.DecSetoid_carrier}
  {t:expression (AccSort gsort lsort E) (AccSort gsort lsort E x)}
  VDMSem S0 E Gamma ([_:(STATE gsort lsort)]trueProp) (assign x t)
      ([tau,sigma:STATE gsort lsort]
Eq tau (STATE_update E x (eval E sigma t) sigma));
...

$Goal VDMSeqAtomicSound :
  {p1,p2:(STATE gsort lsort).Pred}
  {r1,r2:Rel (STATE gsort lsort) (STATE gsort lsort)}
  {Seq_prem1:VDMAtomicSem S0 E p1 S1 ([tau,sigma:(STATE gsort lsort)]
  and (p2 tau) (r1 tau sigma))}
  {Seq_prem2:VDMAtomicSem S0 E p2 S2 r2}
  VDMAtomicSem S0 E p1 (seq S1 S2) (composeRel r2 r1);
...

Goal VDMSeqSound :
  {p1,p2:(STATE gsort lsort).Pred}
  {r1,r2:Rel (STATE gsort lsort) (STATE gsort lsort)}
  {Seq_prem1:VDMSem S0 E Gamma p1 S1 ([tau,sigma:(STATE gsort lsort)]
  and (p2 tau) (r1 tau sigma))}
  {Seq_prem2:VDMSem S0 E Gamma p2 S2 r2}
  VDMSem S0 E Gamma p1 (seq S1 S2) (composeRel r2 r1);
...

$Goal VDMIfthenelseAtomicSound :
  {pre:(STATE gsort lsort).Pred}
  {post:Rel (STATE gsort lsort) (STATE gsort lsort)}
  {TH,EL:prog (AccSort gsort lsort E)}
  {Ifthenelse_prem1:
    VDMAtomicSem S0 E ([sigma:(STATE gsort lsort)]
(pre sigma) ∧ (is_true (eval E sigma b)))
    TH post}
  {Ifthenelse_prem2:
    VDMAtomicSem S0 E ([sigma:(STATE gsort lsort)]
```

```
(pre sigma) ∧ (is_false (eval E sigma b)))
    EL post}
  VDMAtomicSem S0 E pre (ifthenelse b TH EL) post;
…

Goal VDMIfthenelseSound :
  {pre:(STATE gsort lsort).Pred}
  {post:Rel (STATE gsort lsort) (STATE gsort lsort)}
  {TH,EL:prog (AccSort gsort lsort E)}
  {Ifthenelse_prem1:
      VDMSem S0 E Gamma ([sigma:(STATE gsort lsort)]
(pre sigma) ∧ (is_true (eval E sigma b)))
    TH post}
  {Ifthenelse_prem2:
    VDMSem S0 E Gamma ([sigma:(STATE gsort lsort)]
(pre sigma) ∧ (is_false (eval E sigma b)))
    EL post}
    VDMSem S0 E Gamma pre (ifthenelse b TH EL) post;
…

$Goal VDMWhileAtomicSound :
  {P:(STATE gsort lsort).Pred}
    {sofar:Rel (STATE gsort lsort) (STATE gsort lsort)}
    {sofar_trans: trans sofar}
    {sofar_wf: WF (andRel sofar (VDM_Pred_to_Rel P))}
    {body:prog (AccSort gsort lsort E)}
    {Prem:
      VDMAtomicSem S0 E ([sigma:STATE gsort lsort]
       (P sigma) ∧ (is_true (eval E sigma b)))
                body
        ([tau,sigma:STATE gsort lsort]
  (P tau) ∧ (sofar tau sigma))}
   VDMAtomicSem S0 E P (while b body)
          ([tau,sigma:STATE gsort lsort]
    (P tau) ∧ (is_false (eval E tau b)) ∧
                           (reflclose sofar tau sigma));
…

Goal VDMWhileSound :
{P:(STATE gsort lsort).Pred}
    {sofar:Rel (STATE gsort lsort) (STATE gsort lsort)}
    {sofar_trans: trans sofar}
    {sofar_wf: WF (andRel sofar (VDM_Pred_to_Rel P))}
    {body:prog (AccSort gsort lsort E)}
    {Prem:
      VDMSem S0 E Gamma ([sigma:STATE gsort lsort]
       (P sigma) ∧ (is_true (eval E sigma b)))
                body
        ([tau,sigma:STATE gsort lsort]
  (P tau) ∧ (sofar tau sigma))}
   VDMSem S0 E Gamma P (while b body)
          ([tau,sigma:STATE gsort lsort]
    (P tau) ∧ (is_false (eval E tau b)) ∧
                           (reflclose sofar tau sigma));
…

Goal VDMNewSound :
  {U|Type(0)}
  {x:VAR.DecSetoid_carrier}{t:expression (AccSort gsort lsort E) U}
  {S:prog (sort_update x U (AccSort gsort lsort E))}
  {NewPrem :
    {v:lsort x}
    VDMSem S0 (Env_add E x) (VDMcontextPr x v Gamma)
         ([sigma:STATE gsort (sort_update x U lsort)]
   (p (STATEuL x v sigma)) ∧
   (Eq (coerce (sort_update_lookup x U lsort) (sigma.local x))
     (eval E (STATE_remove_local x v sigma) t)))
    (coerce_prog (AccSort_update gsort U E lsort x) S)
      ([tau,sigma:STATE gsort (sort_update x U lsort)]
q (STATEuL x v tau) (STATEuL x v sigma))}
```

```
  VDMSem S0 E Gamma p (new x t S) q;
...

Goal VDMCallSound :
  {W|Type}{Lt':Rel W W}{is_wf : WF Lt'}
    {p:W->Pred (STATE gsort lsort)}
    {q:Rel (STATE gsort lsort) (STATE gsort lsort)}
    {Call_step:{t:W}
      VDMSem S0 (Env_empty|VAR) (VDMSomeSpec
           ([sigma:STATE gsort lsort]
    Ex [u:W](p(u)(sigma)) ∧ (Lt' u t))
   q)
      (p(t))
         (coerce_prog (Env_empty_global gsort lsort) S0) q}
   VDMSem S0 E (VDMNone gsort lsort)
      ([sigma:STATE gsort lsort]Ex [t:W]p(t)(sigma))
      (call (AccSort gsort lsort E)) q;
...

$Goal VDMConAtomicSound :
  {p1:Pred (STATE gsort lsort)}
  {q1:Rel (STATE gsort lsort) (STATE gsort lsort)}
  {S:prog (AccSort gsort lsort E)}
  {ConsSC1:{sigma|STATE gsort lsort}(p sigma) -> (p1 sigma)}
  {ConsSC2:{tau,sigma|STATE gsort lsort}(p sigma) ->
    (q1 tau sigma) -> q tau sigma}
  {ConsPrem:VDMAtomicSem S0 E p1 S q1}
  VDMAtomicSem S0 E p S q;
...

Goal VDMConSound :
  {p1:Pred (STATE gsort lsort)}
  {q1:Rel (STATE gsort lsort) (STATE gsort lsort)}
  {S:prog (AccSort gsort lsort E)}
  {ConsSC1:{sigma|STATE gsort lsort}(p sigma) -> (p1 sigma)}
  {ConsSC2:{tau,sigma|STATE gsort lsort}(p sigma) ->
    (q1 tau sigma) -> q tau sigma}
  {ConsPrem:VDMSem S0 E Gamma p1 S q1}
  VDMSem S0 E Gamma p S q;
...

Discharge E;

(*  Corresponds to Theorem 3.50 on page 110. *)
$Goal VDMsoundness :
  {E|Env|VAR}{lsort|SORT VAR}{Gamma|VDMContext gsort lsort}
  {P|Pred (STATE ??)}{S|prog ?}{Q|Rel (STATE ??) (STATE ??)}
  {der:VDMder S0 E Gamma P S Q}
  VDMSem S0 E Gamma P S Q;
...

Discharge VAR;



Module VDMcomplete Import operate_thms2 WF VDMsemantics VDMtheorems;

[VAR|DecSetoid][gsort|SORT VAR][S0:prog gsort];



[VDMp0
  [lsort|SORT VAR][E|Env VAR][S:prog (AccSort gsort lsort E)]
  [n:nat][sigma:(STATE gsort lsort)] =
     Ex [tau:(STATE gsort lsort)]prog_boperate S0 E sigma S tau n];

[psi [lsort:SORT|VAR][E:Env|VAR][n:nat]
  = VDMp0 (call (AccSort gsort lsort E)) (suc n)]

[p [lsort:SORT|VAR][E:Env|VAR][n:nat][sigma:STATE gsort lsort]
  = Ex ([u:nat](psi lsort E u sigma ∧ Lt u n))];
```

```
[lsort|SORT VAR][E|Env VAR];

Goal VDMp02Termin :
  {S:prog (AccSort gsort lsort E)}{sigma:(STATE gsort lsort)}{n:nat}
  (VDMp0 S n sigma) -> Termin E S0 S sigma;
...


[VDMSC
  [x,y:(Pred (STATE gsort lsort))
    #(Rel (STATE gsort lsort) (STATE gsort lsort))] =
  {tau,sigma|STATE gsort lsort}(x.1 sigma) ->
  ((y.1 sigma) ^ ((y.2 tau sigma) -> x.2 tau sigma))];

Goal VDMSC_refl : refl VDMSC;
...

[b:expression (AccSort gsort lsort E) bool]
[body:prog (AccSort gsort lsort E)];


[n:nat];

Goal VDMWFbsofar :
  WF (andRel (sofar S0 ? b body)
       (VDM_Pred_to_Rel (VDMp0 (while b body) n)));
...

Discharge gsort;

Goal VDM_completeness_step :
  {gsort,lsort|SORT|VAR}
  {S0:prog gsort}{n:nat}
  [CSO = coerce_prog (Env_empty_global gsort lsort) S0]
  VDMder S0 (Env_empty VAR)
  (VDMSomeSpec (p S0 lsort (Env_empty|VAR) n)
    ([tau,sigma:(STATE gsort lsort)]
      prog_operate S0 (Env_empty VAR) sigma (call ?) tau))
   (VDMp0 S0 CSO n)
  CSO
  ([tau,sigma:(STATE gsort lsort)]
    prog_operate S0 (Env_empty VAR) sigma CSO tau);
...


Goal VDM_most_general :
  {gsort,lsort|SORT|VAR}{E|Env VAR}
  {S:prog (AccSort gsort lsort E)}{S0:prog gsort}
  VDMder S0 E (VDMNone ??)
  ([sigma:(STATE gsort lsort)]
    Ex [tau:(STATE gsort lsort)]prog_operate S0 E sigma S tau)
  S
  ([tau,sigma:(STATE gsort lsort)]prog_operate S0 E sigma S tau);
...

(*  Corresponds to Theorem 3.51 on page 110. *)
Goal VDM_complete :
  {gsort,lsort|SORT VAR}{E|Env VAR}{S0|prog gsort}
  {S|prog (AccSort gsort lsort E)}
  {p|(STATE gsort lsort).Pred}
  {q|Rel (STATE gsort lsort) (STATE gsort lsort)}
  {semant:VDMSem S0 E (VDMNone ??) p S q}VDMder S0 E (VDMNone ??) p S q;
...

Discharge VAR;
```

B.5 Quicksort

```
Module quicksort_theory Import lib_nat_Prop_rels lib_list_sorted lib_bool_funs
                               lib_nat_bool_rels lib_nat_rels;

Goal minus_zero_Le : {a,b|nat}(Eq (b .minus a) zero) -> Le b a;
...

Goal Le_zero_Eq_zero : {a|nat}(Le a zero) -> Eq a zero;
...

Goal Lt_imp_Le_pred : {a,b|nat}(Lt a b) -> Le a (pred b);
...

Goal Le_suc_imp_Lt_pred_suc : {a,b|nat}(Le a (suc b)) -> Lt (pred a) (suc b);
...


Goal Le_Lt_contradiction : {a,b|nat}{ab:Le a b}{ba:Lt b a}absurd;
...


Goal minus_suc_left :
  {a,b|nat}(Le a b) -> Eq (minus (suc b) a) (suc (minus b a));
...

Freeze Eq;

Goal Lt_minus_stable_left :
  {a,b,c|nat}(Lt a c) -> (Le c b) -> Lt (minus b c) (minus b a);
...

Goal Lt_minus_stable_right :
  {a,b,c|nat}(Lt a c) -> (Le c b) -> Lt (minus (pred c) a) (minus b a);
...

[s|Type];

Goal singletonSorted : {R:Rel s s}{x:s}Sorted R (singleton x);
...

Goal nat_array_to_list : {A:nat->s}{a,b:nat}list s;
intros A' a' b';
Refine nat_double_elim [a,b:nat]{A:nat->s}list s;
  Refine -0 A'; Refine -0 b'; Refine -0 a';

  (** nat_array_to_list zero zero **)
  intros; Refine singleton (A zero);

  (** nat_array_to_list zero (suc n) **)
  intros _ nat_array_to_list_zero_n A;
  Refine cons (A zero) (nat_array_to_list_zero_n (compose A suc));

  (** nat_array_to_list (suc m) zero **)
  intros; Refine nil;

  (** nat_array_to_list (suc m) (suc n) **)
  intros; Refine H n (compose A suc);
Save nat_array_to_list;

Goal nat_array_to_list_nil :
  {a,b|nat}{A:nat->s}
  (b .Lt a)
   -> Eq (nat_array_to_list A a b) (nil s);
...

Goal nat_array_to_list_cons :
 {a,b|nat}{A:nat->s}(a .Le b) ->
   Eq (nat_array_to_list A a b)
      (cons (A a) (nat_array_to_list A (suc a) b));
```

```
…

Goal nat_array_to_list_cons' :
 {b:nat}{X:nat->s}
 Eq (nat_array_to_list X zero b)
    (cons (X zero) (nat_array_to_list X one b));
…

Goal nat_array_to_list_singleton :
  {b:nat}{X:nat->s}
  Eq (nat_array_to_list X b b) (singleton (X b));
…


Goal nat_array_to_list_append :
  {a,c,b|nat}{X:nat->s}
  (a .Lt b)
    -> (b .Le c)
      ->   Eq (nat_array_to_list X a c)
               (append (nat_array_to_list X a (pred b))
                       (nat_array_to_list X b c));
…

Goal nat_array_to_list_character :
  {a,b:nat}{X:nat->s}
  ((Eq (nat_array_to_list X a b) (nil ?)) ∧ (Lt b a)) ∨
  ((Eq (nat_array_to_list X a b)
       (cons (X a) (nat_array_to_list X (suc a) b))) ∧ (Le a b));
…

Discharge s;

[stable_outside =
  [X:nat->nat][X0:nat->nat][a,b:nat]
  {c:nat}(or (Lt c a) (Lt b c)) -> Eq (X c) (X0 c)];

Goal nat_array_to_list_stable :
   {a,b:nat}{c,d|nat}{X,X'|nat->nat}
   (stable_outside X X' c d)
   -> ((Lt b c) ∨ (Lt d a))
     -> Eq (nat_array_to_list X a b) (nat_array_to_list X' a b);
…

[X,X'|nat->nat][a,b,c|nat];

[s|Type];

[update [A:nat->s][y:nat][t:s] = [x:nat]if (nat_eq y x) t (A x)];

$[Nat_array_to_list_update [a,b:nat] =
  {c|nat}{X:nat->s}{t:s}
  (not (Eq c zero)) ->
  Eq (nat_array_to_list (update X c t) a b) (nat_array_to_list X a b)];

$Goal nat_array_to_list_update_zero_zero : Nat_array_to_list_update zero zero;
…

Goal compose_suc :
     {c:nat}{X:nat->s}{t:s}  (not (Eq c zero)) ->
     Eq (compose (update X c t) suc)
                (update (compose X suc) (pred c) t);
…

Goal compose_suc_zero :
     {X:nat->s}{t:s}Eq (compose (update X zero t) suc) (compose X suc);
…

Goal nat_array_to_list_tack :
  {a,b|nat}{X:nat->s}(Lt (pred a) b) ->
     Eq (nat_array_to_list X a b)
```

```
        (tack (X b) (nat_array_to_list X a (pred b)));
…

(*  Corresponds to Lemma 4.2 on page 115. *)
Goal nat_array_to_list_update :
  {t:s}{a,b:nat}{c|nat}{X:nat->s}
  ((Lt c a) \/ (Lt b c)) ->
  Eq (nat_array_to_list (update X c t) a b) (nat_array_to_list X a b);
…

Discharge X;

[s,t|Type][R:Rel s t];


Goal redun_update : {b:nat}{X:nat->s}Eq X (update X b (X b));
…

(*  Corresponds to Lemma 4.3 on page 115. *)
Goal nat_array_to_list_update_factorise :
  {a,c,b|nat}{X:nat->s}{t:s}
  {ab:Le a b}{bc:Le b c}
  Eq (nat_array_to_list (update X b t) a c)
     (insert t (if (nat_eq b zero) (nil ?) (nat_array_to_list X a (pred b)))
              (nat_array_to_list X (suc b) c));
…

Goal appendPerm :
  {k,k'|list s}
  (Perm k k')->{l,l'|list s}(Perm l l') ->
  Perm (append k l) (append k' l');
…

(*  Corresponds to Lemma 4.5 on page 116. *)
Goal updatePerm :
  {a,b,c,d|nat}{X:nat->s}
  {ac:Le a c}{ad:Le a d}{cb:Le c b}{db:Le d b}
  Perm (nat_array_to_list (update (update X d (X c)) c (X d)) a b)
       (nat_array_to_list X a b);
…

[list_pointwise_ext_right = Rlist];

[list_pointwise_ext = op (Rlist (op (Rlist R)))];

Goal list_pointwise_ext_nil : {k:list s}list_pointwise_ext k (nil t);
…

Goal list_pointwise_ext_cons :
   {k:list s}{x:t}{xs:list t}
   iff (list_pointwise_ext k (cons x xs))
       (and (list_pointwise_ext k (singleton x))
            (list_pointwise_ext k xs));
…

(** morally, this lemma subsumes appclRlist in lib_list_sorted **)
Goal list_pointwise_ext_append_left :
  {k,k':list s}{l:list t}
  iff (and (list_pointwise_ext k l)
           (list_pointwise_ext k' l))
       (list_pointwise_ext (append k k') l);
…

Goal list_pointwise_ext_tack_left :
  {a,b|nat}{X|nat->s}{l|list t}
  (list_pointwise_ext (nat_array_to_list X a b) l) ->
  (list_pointwise_ext_right R (X (suc b)) l) ->
  list_pointwise_ext (nat_array_to_list X a (suc b)) l;
…
```

```
Goal list_pointwise_ext_append_right :
  {k:list s}{l,l':list t}
  iff (and (list_pointwise_ext k l)
           (list_pointwise_ext k l'))
       (list_pointwise_ext k (append l l'));
...

Goal list_pointwise_ext_tack_right :
  {a,b|nat}{X:nat->t}{l|list s}
  (list_pointwise_ext l (nat_array_to_list X a b)) ->
  (list_pointwise_ext l (singleton (X (suc b)))) ->
  list_pointwise_ext l (nat_array_to_list X a (suc b));
...

Goal list_pointwise_ext_resp_perm_left :
  {k,k'|list s}{l|list t}
  (Perm k k')
      -> (list_pointwise_ext k l)
        -> list_pointwise_ext k' l;
...

Goal list_pointwise_ext_resp_perm_right :
  {k|list s}{l,l'|list t}
  (Perm l l')
      -> (list_pointwise_ext k l)
        -> list_pointwise_ext k l';
...



[a|s][k|list s][b|t][l|list t];

Goal list_pointwise_ext_boundary :
   (list_pointwise_ext (tack a k) (cons b l))
     -> R a b;
...


Discharge a;

Goal list_pointwise_nat_array_to_list_suc :
  {r|s}{a,b|nat}{X|nat->t}
  (R r (X a)) ->
  (list_pointwise_ext_right R r (nat_array_to_list X (suc a) b)) ->
  (list_pointwise_ext_right R r (nat_array_to_list X a b));
...

DischargeKeep R;

[S:Rel s t];

Goal list_pointwise_resp_SubRel :
   (SubRel R S) -> (SubRel (list_pointwise_ext R) (list_pointwise_ext S));
...

Discharge t;

[R:Rel s s];

Goal list_pointwise_trans :
  (trans R) ->
  {k,l|list s}{m|s}
  {km:list_pointwise_ext R k (singleton m)}
  {ml:list_pointwise_ext R (singleton m) l}
  list_pointwise_ext R k l;
...

Discharge s;
```

```
Goal array_sorted_append :
  {a,b,c|nat}{X|nat->nat}
  [Xleft = nat_array_to_list X a (pred b)]
  [Xright= nat_array_to_list X b c]
  (list_pointwise_ext Le Xleft Xright)
    -> (Sorted Le Xleft)
      -> (Sorted Le Xright)
        -> Sorted Le (append Xleft Xright);
...

(*  Corresponds to Definition 4.6 on page 116. *)
[Perm' [A,B:nat->nat][a,b:nat] =
  (Perm (nat_array_to_list A a b) (nat_array_to_list B a b)) ∧
  (stable_outside A B a b)];

(*  Corresponds to Definition 4.7 on page 116. *)
[Spec_Quicksort_post [a,b:nat][x,X:nat->nat] =
     [Xab  = nat_array_to_list x a b]
     (Sorted Le Xab) ∧
     (Perm' x X a b)];

[a,b,c|nat][X0,X1,X2,X3|nat->nat]
[ac : Lt a c][cb :Le c b]

[Quicksort_Perm: Perm' X1 X0 a b]

[Quicksort_Partition:
    list_pointwise_ext Le
    (nat_array_to_list X1 a (pred c)) (nat_array_to_list X1 c b)]

[Quicksort_first_segment:  Spec_Quicksort_post a (pred c) X2 X1]
[Quicksort_second_segment: Spec_Quicksort_post c b X3 X2];

Freeze Perm nat_array_to_list list_pointwise_ext;

(*  Corresponds to Theorem 4.8 on page 116. *)
Goal Sorted_Quicksort : Spec_Quicksort_post a b X3 X0;
...

Discharge a;

Unfreeze Eq nat_array_to_list list_pointwise_ext;


Module  quicksort_prog Import syntax quicksort_theory;

Inductive [QuicksortVAR:Type] Theorems
Constructors [X_,a_,b_,Pa_,Pb_,c_,d_,exch_,r_:QuicksortVAR];

Goal QuicksortVAR_eq : {x,y:QuicksortVAR}bool;
...

Goal QuicksortVAR_eq_refl : {x:QuicksortVAR}is_true (QuicksortVAR_eq x x);
...

Freeze Eq;

Goal QuicksortVAR_eq_char : Eqchar QuicksortVAR_eq;
...

Unfreeze Eq;

Goal QuicksortVAR_DecSetoid : DecSetoid;
...

[X    = X_    : QuicksortVAR_DecSetoid.DecSetoid_carrier]
[a    = a_    : QuicksortVAR_DecSetoid.DecSetoid_carrier]
[b    = b_    : QuicksortVAR_DecSetoid.DecSetoid_carrier]
[Pa   = Pa_   : QuicksortVAR_DecSetoid.DecSetoid_carrier]
[Pb   = Pb_   : QuicksortVAR_DecSetoid.DecSetoid_carrier]
```

```
[c    = c_    : QuicksortVAR_DecSetoid.DecSetoid_carrier]
[d    = d_    : QuicksortVAR_DecSetoid.DecSetoid_carrier]
[exch = exch_ : QuicksortVAR_DecSetoid.DecSetoid_carrier]
[r    = r_    : QuicksortVAR_DecSetoid.DecSetoid_carrier];

Goal QuicksortSORT : {x:QuicksortVAR_DecSetoid.DecSetoid_carrier}Type;
Expand DecSetoid_carrier;
Induction x;
Refine nat ->nat;
Refine nat; Refine nat; Refine nat; Refine nat; Refine nat;
Refine nat; Refine nat; Refine nat;
Save QuicksortSORT;

(*  Corresponds to Definition 4.13 on page 120. *)
Goal Prepare:
   prog QuicksortSORT;

Refine seq;
Refine ifthenelse;
  Intros sigma; Refine lt (sigma(X)(sigma b)) (sigma(X)(sigma a));

  (** Then branch **)
  Refine new; Refine +1 exch; Intros +1 sigma; Refine sigma(X)(sigma a);
    Refine seq;
      Refine assign; Refine X;
        Intros sigma;
          Refine update (sigma X) (sigma a) (sigma X (sigma b));
      Refine assign; Refine X;
        Intros sigma;
          Refine update (sigma X) (sigma b) (sigma exch);

  (** Else branch **)
  Refine skip;

  Refine new; Refine +1 r;
    Intros +1 sigma;
      Refine (sigma(X)(sigma a));

    (** partition (a+1,b-1,r,c,X) **)
    Refine seq; Refine assign; Refine c; Intros sigma; Refine suc(sigma a);
     Refine new; Refine +1 d; Intros +1 sigma; Refine pred(sigma(b));
       Refine while; Intros sigma; Refine le (sigma(c)) (sigma(d));
         Refine ifthenelse; Intros sigma;
           Refine le (sigma(X)(sigma(c))) (sigma(r));
           Refine assign; Refine c; Intros sigma; Refine suc (sigma(c));
           Refine ifthenelse; Intros sigma;
             Refine lt (sigma(r)) (sigma(X)(sigma(d)));
             Refine assign; Refine d; Intros sigma; Refine pred (sigma(d));
             Refine seq;
               Refine new; Refine +1 exch; Intros +1 sigma;
       Refine sigma(X)(sigma(c));
                Refine seq;
                  Refine assign; Refine X;
                    Intros sigma;
                    Refine update (sigma X) (sigma c) (sigma(X)(sigma(d)));
                  Refine assign; Refine X;
                    Intros sigma;
                    Refine update (sigma X) (sigma d)  (sigma(exch));
                Refine seq;
                  Refine assign; Refine c; Intros sigma; Refine suc (sigma(c));
                  Refine assign; Refine d; Intros sigma; Refine pred (sigma(d));
Save Prepare;

Goal coerce_Prepare :
  EXTEQ QuicksortSORT
  (sort_update c nat (sort_update b nat (sort_update a nat QuicksortSORT)));
Expand EXTEQ; Expand DecSetoid_carrier;
Induction x;
  Refine Eq_refl; Refine Eq_refl; Refine Eq_refl; Refine Eq_refl;
  Refine Eq_refl; Refine Eq_refl; Refine Eq_refl; Refine Eq_refl;
```

```
  Refine Eq_refl;
SaveFrozen coerce_Prepare;

(*  Corresponds to Definition 4.12 on page 120. *)
Goal S0 : prog|QuicksortVAR_DecSetoid QuicksortSORT;
Refine new; Refine +1 a; Intros +1 sigma; Refine sigma(Pa);
Refine new; Refine +1 b; Intros +1 sigma; Refine sigma(Pb);
Refine ifthenelse;

  (** boolean expression **)
  Intros sigma;
  Refine lt (sigma a) (sigma b);

  (** THEN branch **)
  Refine  new; Refine +1 c; Intros +1; Refine zero; (* any value will do *)
  For 6 Refine seq; Refine coerce_prog coerce_Prepare Prepare;

  (** a' := a **)
  Refine assign; Refine Pa;
    Intros sigma; Refine sigma (a);

  (** b' := c-1 **)
  Refine assign; Refine Pb;
    Intros sigma; Refine pred (sigma c);

  Refine call;

  (** a' := c **)
  Refine assign; Refine Pa;
    Intros sigma; Refine sigma c;

  (** b' := b **)
  Refine assign; Refine Pb;
    Intros sigma; Refine sigma b;

  Refine call;

  (** ELSE branch **)
  Refine skip;
Save S0;
```

```
Module  quicksort_spec
  Import lib_prod quicksort_prog quicksort_theory assertion;

[qsSTATE = STATE QuicksortSORT QuicksortSORT]
[qsSTATE' = ((qsSTATE#nat)#nat)#nat]

[PrepareLSORT =
  sort_update c nat (sort_update b nat (sort_update a nat QuicksortSORT))]

[PrepareSTATE = STATE QuicksortSORT PrepareLSORT]

[E:Env QuicksortVAR_DecSetoid];

[Spec_Quicksort_pre [Z,sigma:qsSTATE]
   = (Eq (lookup E sigma Pa) (lookup E Z Pa)) ∧
     (Eq (lookup E sigma Pb) (lookup E Z Pb)) ∧
     (Eq (lookup E sigma X) (lookup E Z X))];

Discharge E;

[sortdr = sort_update d nat (sort_update r nat QuicksortSORT)];
[E = Env_add (Env_add (Env_add (Env_empty QuicksortVAR_DecSetoid) a) b) c]
[Erd = Env_add (Env_add E r) d];

[Spec_Prepare_pre
   = [z:qsSTATE'][sigma:PrepareSTATE]
```

```
      [z=z.1.1.1]
      [A=    lookup E z a][B=    lookup E z b]
      [a=lookup E sigma a]
      [b=lookup E sigma b]
      [x=lookup E sigma X]
      [X=    lookup E z X]
      (Lt a b) ∧ (Eq a A) ∧ (Eq b B) ∧ (Eq x X)]

(*  Corresponds to Definition 4.16 on page 121. *)
[Assertion_Prepare_variant
  [z:qsSTATE][sigma:PrepareSTATE] =
  [x = (lookup E sigma X)]
  [X = (lookup E z X)]
  [A = (lookup E z a)]
  [B = (lookup E z b)]
  [c = (lookup E sigma c)]
  [Xleft = nat_array_to_list x A (pred c)]
  [Xright = nat_array_to_list x c B]
  (Lt A c) ∧ (Le c B) ∧
  (list_pointwise_ext Le Xleft Xright) ∧
  (Perm' x X A B)];

(*  Corresponds to Definition 4.17 on page 121. *)
[Spec_Prepare_inv = [D:qsSTATE -> nat]
  [z,sigma:qsSTATE]
  [x = (lookup Erd sigma X)]
  [X = (lookup Erd z X)]
  [A = (lookup Erd z a)]
  [a = (lookup Erd sigma a)]
  [B = (lookup Erd z b)]
  [b = (lookup Erd sigma b)]
  [c = (lookup Erd sigma c)]
  [r = (lookup Erd sigma r)]
  (Lt A c) ∧ (Le c (suc (D sigma))) ∧ (Lt (D sigma) B) ∧
   (list_pointwise_ext Le (nat_array_to_list x A (pred c))
      (singleton r)) ∧
   (list_pointwise_ext Lt (singleton r)
      (nat_array_to_list x (suc (D sigma)) (pred B))) ∧
   (Le r (x B)) ∧
   (Perm' x X A B) ∧
    (Eq a A) ∧ (Eq b B)];

[Spec_Prepare_post
  = [z:qsSTATE']
    [sigma:PrepareSTATE]
    [z = z.1.1.1]
    (Assertion_Prepare_variant z sigma) ∧
    (Eq (lookup E sigma a) (lookup E z a)) ∧
    (Eq (lookup E sigma b) (lookup E z b))];

Configure Infix - left 5;
[op- = minus];

Goal EXTEQdrPrepareLQuicksort :
  EXTEQ (sort_update d nat (sort_update r nat PrepareLSORT))
        QuicksortSORT;
...

Goal drPrepareLSORT2qsSTATE :
  {sigma:STATE QuicksortSORT
    (sort_update d nat (sort_update r nat PrepareLSORT))}
  qsSTATE;
...

Goal Prepare_coerce' :
  EXTEQ QuicksortSORT (AccSort QuicksortSORT PrepareLSORT E);
...


Module Prepare_vc1 Import quicksort_spec;
```

```
[t:nat];

(* corresponds to the case x[c]<=r *)

Goal Pvc1 :
  SubRel ([Z:qsSTATE']
   [sigma:STATE QuicksortSORT
     (sort_update d nat (sort_update r nat PrepareLSORT))]
   (Spec_Prepare_inv ([sigma'3:qsSTATE]local sigma'3 d) Z.1.1.1
     (drPrepareLSORT2qsSTATE sigma) ∧
       is_true (eval Erd sigma
 (coerce_expression QuicksortVAR_DecSetoid
   (sort_update d nat
     (AccSort QuicksortSORT
       (sort_update r nat PrepareLSORT)
       (Env_add E r)))
   (AccSort QuicksortSORT
     (sort_update d nat
       (sort_update r nat PrepareLSORT))
     (Env_add (Env_add E r) d))
   (AccSort_update QuicksortSORT nat (Env_add E r)
     (sort_update r nat PrepareLSORT) d)
   (coerce_expression QuicksortVAR_DecSetoid
     (sort_update d nat
       (sort_update r nat
 (AccSort QuicksortSORT PrepareLSORT E)))
     (sort_update d nat
       (AccSort QuicksortSORT
 (sort_update r nat PrepareLSORT)
 (Env_add E r)))
     (coerce_sort_update (AccSort_update QuicksortSORT
   nat E PrepareLSORT r)
       nat d)
     (coerce_expression QuicksortVAR_DecSetoid
       (sort_update d nat
 (sort_update r nat QuicksortSORT))
       (sort_update d nat
 (sort_update r nat
   (AccSort QuicksortSORT PrepareLSORT E)))
       (coerce_sort_update (coerce_sort_update Prepare_coerce'
     nat r) nat d)
       ([sigma'3:FS (sort_update d nat
      (sort_update r nat
QuicksortSORT))]
 le (sigma'3 c) (sigma'3 d))))))) ∧
     Eq (suc (lookup Erd sigma d) - lookup Erd sigma c) t ∧
     is_true (eval Erd sigma
       (coerce_expression QuicksortVAR_DecSetoid
 (sort_update d nat
   (AccSort QuicksortSORT
       (sort_update r nat PrepareLSORT)
     (Env_add E r)))
 (AccSort QuicksortSORT
   (sort_update d nat
       (sort_update r nat PrepareLSORT))
     (Env_add (Env_add E r) d))
   (AccSort_update QuicksortSORT nat (Env_add E r)
     (sort_update r nat PrepareLSORT) d)
   (coerce_expression QuicksortVAR_DecSetoid
     (sort_update d nat
       (sort_update r nat
 (AccSort QuicksortSORT PrepareLSORT E)))
     (sort_update d nat
       (AccSort QuicksortSORT
 (sort_update r nat PrepareLSORT)
 (Env_add E r)))
     (coerce_sort_update (AccSort_update QuicksortSORT
   nat E PrepareLSORT r)
       nat d)
```

```
     (coerce_expression QuicksortVAR_DecSetoid
       (sort_update d nat
 (sort_update r nat QuicksortSORT))
       (sort_update d nat
 (sort_update r nat
   (AccSort QuicksortSORT PrepareLSORT E)))
       (coerce_sort_update (coerce_sort_update Prepare_coerce'
     nat r) nat d)
       ([sigma'3:FS (sort_update d nat
      (sort_update r nat
QuicksortSORT))]
 le (sigma'3 X (sigma'3 c)) (sigma'3 r))))))
       ))
    ([Z:qsSTATE']
      [sigma:STATE QuicksortSORT
(sort_update d nat (sort_update r nat PrepareLSORT))]
      (Spec_Prepare_inv ([sigma'3:qsSTATE]local sigma'3 d) Z.1.1.1
(drPrepareLSORT2qsSTATE (STATE_update Erd c
  (eval Erd sigma
    (coerce_expression' QuicksortVAR_DecSetoid
      c
      (sort_update d nat
(AccSort QuicksortSORT
  (sort_update r nat PrepareLSORT
    ) (Env_add E r)))
      (AccSort QuicksortSORT
(sort_update d nat
  (sort_update r nat PrepareLSORT
    )) (Env_add (Env_add E r) d))
      (AccSort_update QuicksortSORT nat
(Env_add E r)
(sort_update r nat PrepareLSORT)
d)
      (coerce_expression' QuicksortVAR_DecSetoid
c
(sort_update d nat
  (sort_update r nat
    (AccSort QuicksortSORT
      PrepareLSORT E)))
(sort_update d nat
  (AccSort QuicksortSORT
    (sort_update r nat
      PrepareLSORT) (Env_add E r)
    ))
(coerce_sort_update (AccSort_update QuicksortSORT
      nat E
      PrepareLSORT
      r) nat d)
(coerce_expression' QuicksortVAR_DecSetoid
  c
  (sort_update d nat
    (sort_update r nat
      QuicksortSORT))
  (sort_update d nat
    (sort_update r nat
      (AccSort QuicksortSORT
PrepareLSORT E)))
  (coerce_sort_update (coerce_sort_update Prepare_coerce'
nat r)
    nat d)
  ([sigma'3:FS (sort_update d nat
 (sort_update r
   nat
   QuicksortSORT)
 )]
    suc (sigma'3 c)))))) sigma))
∧
Lt (suc (lookup Erd
  (STATE_update Erd c
    (eval Erd sigma
```

```
      (coerce_expression' QuicksortVAR_DecSetoid c
(sort_update d nat
  (AccSort QuicksortSORT
    (sort_update r nat PrepareLSORT) (Env_add E r)
    ))
(AccSort QuicksortSORT
  (sort_update d nat
    (sort_update r nat PrepareLSORT))
  (Env_add (Env_add E r) d))
(AccSort_update QuicksortSORT nat (Env_add E r)
  (sort_update r nat PrepareLSORT) d)
(coerce_expression' QuicksortVAR_DecSetoid c
  (sort_update d nat
    (sort_update r nat
      (AccSort QuicksortSORT PrepareLSORT E)))
  (sort_update d nat
    (AccSort QuicksortSORT
      (sort_update r nat PrepareLSORT)
      (Env_add E r)))
  (coerce_sort_update (AccSort_update QuicksortSORT
nat E PrepareLSORT r) nat
    d)
  (coerce_expression' QuicksortVAR_DecSetoid c
    (sort_update d nat
      (sort_update r nat QuicksortSORT))
    (sort_update d nat
      (sort_update r nat
(AccSort QuicksortSORT PrepareLSORT E)))
    (coerce_sort_update (coerce_sort_update Prepare_coerce'
  nat r) nat d)
    ([sigma'3:FS (sort_update d nat
   (sort_update r nat
    QuicksortSORT))]
     suc (sigma'3 c)))))) sigma) d) -
     lookup Erd
     (STATE_update Erd c
       (eval Erd sigma
 (coerce_expression' QuicksortVAR_DecSetoid c
   (sort_update d nat
     (AccSort QuicksortSORT
       (sort_update r nat PrepareLSORT) (Env_add E r)))
   (AccSort QuicksortSORT
     (sort_update d nat (sort_update r nat PrepareLSORT))
     (Env_add (Env_add E r) d))
   (AccSort_update QuicksortSORT nat (Env_add E r)
     (sort_update r nat PrepareLSORT) d)
   (coerce_expression' QuicksortVAR_DecSetoid c
     (sort_update d nat
       (sort_update r nat
 (AccSort QuicksortSORT PrepareLSORT E)))
     (sort_update d nat
       (AccSort QuicksortSORT
 (sort_update r nat PrepareLSORT) (Env_add E r)))
     (coerce_sort_update (AccSort_update QuicksortSORT nat
   E PrepareLSORT r) nat d)
     (coerce_expression' QuicksortVAR_DecSetoid c
       (sort_update d nat
 (sort_update r nat QuicksortSORT))
       (sort_update d nat
 (sort_update r nat
   (AccSort QuicksortSORT PrepareLSORT E)))
       (coerce_sort_update (coerce_sort_update Prepare_coerce'
     nat r) nat d)
       ([sigma'3:FS (sort_update d nat
     (sort_update r nat QuicksortSORT))]
 suc (sigma'3 c)))))) sigma) c) t));
...

Discharge t;
```

```
Module Prepare_vc2 Import quicksort_spec;

[t:nat];

(* corresponds to the case x[c] > d and x[d] > r *)
Goal Pvc2 :
 SubRel ([Z:qsSTATE']
  [sigma:STATE QuicksortSORT
    (sort_update d nat (sort_update r nat PrepareLSORT))]
  (Spec_Prepare_inv ([sigma'3:qsSTATE]local sigma'3 d) Z.1.1.1
    (drPrepareLSORT2qsSTATE sigma) ∧
    is_true (eval Erd sigma
      (coerce_expression QuicksortVAR_DecSetoid
(sort_update d nat
  (AccSort QuicksortSORT
    (sort_update r nat PrepareLSORT)
    (Env_add E r)))
(AccSort QuicksortSORT
  (sort_update d nat
    (sort_update r nat PrepareLSORT))
  (Env_add (Env_add E r) d))
(AccSort_update QuicksortSORT nat (Env_add E r)
  (sort_update r nat PrepareLSORT) d)
(coerce_expression QuicksortVAR_DecSetoid
  (sort_update d nat
    (sort_update r nat
      (AccSort QuicksortSORT PrepareLSORT E)))
  (sort_update d nat
    (AccSort QuicksortSORT
      (sort_update r nat PrepareLSORT)
      (Env_add E r)))
  (coerce_sort_update (AccSort_update QuicksortSORT
nat E PrepareLSORT r)
    nat d)
  (coerce_expression QuicksortVAR_DecSetoid
    (sort_update d nat
      (sort_update r nat QuicksortSORT))
    (sort_update d nat
      (sort_update r nat
(AccSort QuicksortSORT PrepareLSORT E)))
    (coerce_sort_update (coerce_sort_update Prepare_coerce'
  nat r) nat d)
    ([sigma'3:FS (sort_update d nat
   (sort_update r nat
     QuicksortSORT))]
      le (sigma'3 c) (sigma'3 d)))))) ∧
    Eq (suc (lookup Erd sigma d) - lookup Erd sigma c) t ∧
    is_false (eval Erd sigma
       (coerce_expression QuicksortVAR_DecSetoid
 (sort_update d nat
   (AccSort QuicksortSORT
     (sort_update r nat PrepareLSORT)
     (Env_add E r)))
 (AccSort QuicksortSORT
   (sort_update d nat
     (sort_update r nat PrepareLSORT))
   (Env_add (Env_add E r) d))
 (AccSort_update QuicksortSORT nat (Env_add E r)
   (sort_update r nat PrepareLSORT) d)
 (coerce_expression QuicksortVAR_DecSetoid
   (sort_update d nat
     (sort_update r nat
       (AccSort QuicksortSORT PrepareLSORT E)))
   (sort_update d nat
     (AccSort QuicksortSORT
       (sort_update r nat PrepareLSORT)
       (Env_add E r)))
   (coerce_sort_update (AccSort_update QuicksortSORT
 nat E PrepareLSORT r)
     nat d)
```

```
   (coerce_expression QuicksortVAR_DecSetoid
     (sort_update d nat
       (sort_update r nat QuicksortSORT))
     (sort_update d nat
       (sort_update r nat
 (AccSort QuicksortSORT PrepareLSORT E))
       )
     (coerce_sort_update (coerce_sort_update Prepare_coerce'
   nat r) nat d)
     ([sigma'3:FS (sort_update d nat
    (sort_update r nat
     QuicksortSORT))]
      le (sigma'3 X (sigma'3 c)) (sigma'3 r)))))
      ) ∧
    is_true (eval Erd sigma
      (coerce_expression QuicksortVAR_DecSetoid
(sort_update d nat
  (AccSort QuicksortSORT
    (sort_update r nat PrepareLSORT)
    (Env_add E r)))
(AccSort QuicksortSORT
  (sort_update d nat
    (sort_update r nat PrepareLSORT))
  (Env_add (Env_add E r) d))
(AccSort_update QuicksortSORT nat (Env_add E r)
  (sort_update r nat PrepareLSORT) d)
(coerce_expression QuicksortVAR_DecSetoid
  (sort_update d nat
    (sort_update r nat
      (AccSort QuicksortSORT PrepareLSORT E)))
  (sort_update d nat
    (AccSort QuicksortSORT
      (sort_update r nat PrepareLSORT)
      (Env_add E r)))
  (coerce_sort_update (AccSort_update QuicksortSORT
nat E PrepareLSORT r)
    nat d)
  (coerce_expression QuicksortVAR_DecSetoid
    (sort_update d nat
      (sort_update r nat QuicksortSORT))
    (sort_update d nat
      (sort_update r nat
(AccSort QuicksortSORT PrepareLSORT E)))
    (coerce_sort_update (coerce_sort_update Prepare_coerce'
  nat r) nat d)
    ([sigma'3:FS (sort_update d nat
   (sort_update r nat
    QuicksortSORT))]
     lt (sigma'3 r) (sigma'3 X (sigma'3 d)))))))
    ))
  ([Z:qsSTATE']
    [sigma:STATE QuicksortSORT
      (sort_update d nat (sort_update r nat PrepareLSORT))]
    (Spec_Prepare_inv ([sigma'3:qsSTATE]local sigma'3 d) Z.1.1.1
      (drPrepareLSORT2qsSTATE (STATE_update Erd d
(eval Erd sigma
  (coerce_expression' QuicksortVAR_DecSetoid
    d
    (sort_update d nat
      (AccSort QuicksortSORT
(sort_update r nat PrepareLSORT
  ) (Env_add E r)))
    (AccSort QuicksortSORT
      (sort_update d nat
(sort_update r nat PrepareLSORT
  )) (Env_add (Env_add E r) d))
    (AccSort_update QuicksortSORT nat
      (Env_add E r)
      (sort_update r nat PrepareLSORT)
      d)
```

```
    (coerce_expression' QuicksortVAR_DecSetoid
      d
      (sort_update d nat
(sort_update r nat
  (AccSort QuicksortSORT
    PrepareLSORT E)))
      (sort_update d nat
(AccSort QuicksortSORT
  (sort_update r nat
    PrepareLSORT) (Env_add E r)
  ))
      (coerce_sort_update (AccSort_update QuicksortSORT
    nat E
    PrepareLSORT
    r) nat d)
      (coerce_expression' QuicksortVAR_DecSetoid
d
(sort_update d nat
  (sort_update r nat
    QuicksortSORT))
(sort_update d nat
  (sort_update r nat
    (AccSort QuicksortSORT
      PrepareLSORT E)))
(coerce_sort_update (coerce_sort_update Prepare_coerce'
      nat r)
  nat d)
([sigma'3:FS (sort_update d nat
       (sort_update r
 nat
 QuicksortSORT)
       )]
  pred (sigma'3 d)))))) sigma))
      ∧
      Lt (suc (lookup Erd
(STATE_update Erd d
  (eval Erd sigma
    (coerce_expression' QuicksortVAR_DecSetoid d
      (sort_update d nat
(AccSort QuicksortSORT
  (sort_update r nat PrepareLSORT) (Env_add E r)
  ))
      (AccSort QuicksortSORT
(sort_update d nat
  (sort_update r nat PrepareLSORT))
(Env_add (Env_add E r) d))
      (AccSort_update QuicksortSORT nat (Env_add E r)
(sort_update r nat PrepareLSORT) d)
      (coerce_expression' QuicksortVAR_DecSetoid d
(sort_update d nat
  (sort_update r nat
    (AccSort QuicksortSORT PrepareLSORT E)))
(sort_update d nat
  (AccSort QuicksortSORT
    (sort_update r nat PrepareLSORT)
    (Env_add E r)))
(coerce_sort_update (AccSort_update QuicksortSORT
      nat E PrepareLSORT r) nat
  d)
(coerce_expression' QuicksortVAR_DecSetoid d
  (sort_update d nat
    (sort_update r nat QuicksortSORT))
  (sort_update d nat
    (sort_update r nat
      (AccSort QuicksortSORT PrepareLSORT E)))
  (coerce_sort_update (coerce_sort_update Prepare_coerce'
nat r) nat d)
  ([sigma'3:FS (sort_update d nat
 (sort_update r nat
   QuicksortSORT))]
```

```
    pred (sigma'3 d)))))) sigma) d) -
   lookup Erd
   (STATE_update Erd d
     (eval Erd sigma
       (coerce_expression' QuicksortVAR_DecSetoid d
 (sort_update d nat
   (AccSort QuicksortSORT
     (sort_update r nat PrepareLSORT) (Env_add E r)))
 (AccSort QuicksortSORT
   (sort_update d nat (sort_update r nat PrepareLSORT))
   (Env_add (Env_add E r) d))
 (AccSort_update QuicksortSORT nat (Env_add E r)
   (sort_update r nat PrepareLSORT) d)
 (coerce_expression' QuicksortVAR_DecSetoid d
   (sort_update d nat
     (sort_update r nat
       (AccSort QuicksortSORT PrepareLSORT E)))
   (sort_update d nat
     (AccSort QuicksortSORT
       (sort_update r nat PrepareLSORT) (Env_add E r)))
   (coerce_sort_update (AccSort_update QuicksortSORT nat
 E PrepareLSORT r) nat d)
   (coerce_expression' QuicksortVAR_DecSetoid d
     (sort_update d nat
       (sort_update r nat QuicksortSORT))
     (sort_update d nat
       (sort_update r nat
 (AccSort QuicksortSORT PrepareLSORT E)))
     (coerce_sort_update (coerce_sort_update Prepare_coerce'
   nat r) nat d)
     ([sigma'3:FS (sort_update d nat
    (sort_update r nat QuicksortSORT))]
       pred (sigma'3 d)))))) sigma) c) t));
...

Discharge t;


Module  Prepare_vc3 Import quicksort_spec;

[t:nat];
[vexch : sort_update d nat (sort_update r nat PrepareLSORT) exch];

(* corresponds to the case x[c]>r and x[d]<= r *)
Goal Pvc3 :
   SubRel ([Z:qsSTATE']
    [sigma:STATE QuicksortSORT
      (sort_update exch nat (sort_update d nat (sort_update r nat PrepareLSORT)))]
    (Fr exch
      ([Z'3:qsSTATE']
[sigma'4:STATE QuicksortSORT (sort_update d nat (sort_update r nat PrepareLSORT))
  ]
(Spec_Prepare_inv ([sigma'5:qsSTATE]local sigma'5 d) Z'3.1.1.1
  (drPrepareLSORT2qsSTATE sigma'4) ∧
  is_true (eval Erd sigma'4
    (coerce_expression QuicksortVAR_DecSetoid
      (sort_update d nat
(AccSort QuicksortSORT (sort_update r nat PrepareLSORT)
  (Env_add E r)))
      (AccSort QuicksortSORT
(sort_update d nat (sort_update r nat PrepareLSORT))
(Env_add (Env_add E r) d))
      (AccSort_update QuicksortSORT nat (Env_add E r)
(sort_update r nat PrepareLSORT) d)
      (coerce_expression QuicksortVAR_DecSetoid
(sort_update d nat
  (sort_update r nat (AccSort QuicksortSORT PrepareLSORT E)))
(sort_update d nat
  (AccSort QuicksortSORT (sort_update r nat PrepareLSORT)
    (Env_add E r)))
```

```
(coerce_sort_update (AccSort_update QuicksortSORT nat E
      PrepareLSORT r) nat d)
(coerce_expression QuicksortVAR_DecSetoid
  (sort_update d nat (sort_update r nat QuicksortSORT))
  (sort_update d nat
    (sort_update r nat (AccSort QuicksortSORT PrepareLSORT E)))
  (coerce_sort_update (coerce_sort_update Prepare_coerce' nat r)
    nat d)
  ([sigma'5:FS (sort_update d nat
 (sort_update r nat QuicksortSORT))]
    le (sigma'5 c) (sigma'5 d)))))) ∧
  Eq (suc (lookup Erd sigma'4 d) - lookup Erd sigma'4 c) t ∧
  is_false (eval Erd sigma'4
     (coerce_expression QuicksortVAR_DecSetoid
       (sort_update d nat
 (AccSort QuicksortSORT (sort_update r nat PrepareLSORT)
   (Env_add E r)))
       (AccSort QuicksortSORT
 (sort_update d nat (sort_update r nat PrepareLSORT))
 (Env_add (Env_add E r) d))
       (AccSort_update QuicksortSORT nat (Env_add E r)
 (sort_update r nat PrepareLSORT) d)
       (coerce_expression QuicksortVAR_DecSetoid
 (sort_update d nat
   (sort_update r nat (AccSort QuicksortSORT PrepareLSORT E)))
 (sort_update d nat
   (AccSort QuicksortSORT (sort_update r nat PrepareLSORT)
     (Env_add E r)))
 (coerce_sort_update (AccSort_update QuicksortSORT nat E
       PrepareLSORT r) nat d)
 (coerce_expression QuicksortVAR_DecSetoid
   (sort_update d nat (sort_update r nat QuicksortSORT))
   (sort_update d nat
     (sort_update r nat (AccSort QuicksortSORT PrepareLSORT E)))
   (coerce_sort_update (coerce_sort_update Prepare_coerce' nat r)
     nat d)
   ([sigma'5:FS (sort_update d nat
  (sort_update r nat QuicksortSORT))]
     le (sigma'5 X (sigma'5 c)) (sigma'5 r)))))) ∧
  is_false (eval Erd sigma'4
     (coerce_expression QuicksortVAR_DecSetoid
       (sort_update d nat
 (AccSort QuicksortSORT (sort_update r nat PrepareLSORT)
   (Env_add E r)))
       (AccSort QuicksortSORT
 (sort_update d nat (sort_update r nat PrepareLSORT))
 (Env_add (Env_add E r) d))
       (AccSort_update QuicksortSORT nat (Env_add E r)
 (sort_update r nat PrepareLSORT) d)
       (coerce_expression QuicksortVAR_DecSetoid
 (sort_update d nat
   (sort_update r nat (AccSort QuicksortSORT PrepareLSORT E)))
 (sort_update d nat
   (AccSort QuicksortSORT (sort_update r nat PrepareLSORT)
     (Env_add E r)))
 (coerce_sort_update (AccSort_update QuicksortSORT nat E
       PrepareLSORT r) nat d)
 (coerce_expression QuicksortVAR_DecSetoid
   (sort_update d nat (sort_update r nat QuicksortSORT))
   (sort_update d nat
     (sort_update r nat (AccSort QuicksortSORT PrepareLSORT E)))
   (coerce_sort_update (coerce_sort_update Prepare_coerce' nat r)
     nat d)
   ([sigma'5:FS (sort_update d nat
  (sort_update r nat QuicksortSORT))]
     lt (sigma'5 r) (sigma'5 X (sigma'5 d))))))))) vexch Z sigma ∧
      Eq (coerce (sort_update_lookup exch nat
   (sort_update d nat (sort_update r nat PrepareLSORT)))
   (local sigma exch))
      (eval Erd (STATE_remove_local exch vexch sigma)
```

```
(coerce_expression QuicksortVAR_DecSetoid
  (sort_update d nat
    (AccSort QuicksortSORT (sort_update r nat PrepareLSORT) (Env_add E r)))
  (AccSort QuicksortSORT (sort_update d nat (sort_update r nat PrepareLSORT))
    (Env_add (Env_add E r) d))
  (AccSort_update QuicksortSORT nat (Env_add E r)
    (sort_update r nat PrepareLSORT) d)
  (coerce_expression QuicksortVAR_DecSetoid
    (sort_update d nat
      (sort_update r nat (AccSort QuicksortSORT PrepareLSORT E)))
    (sort_update d nat
      (AccSort QuicksortSORT (sort_update r nat PrepareLSORT) (Env_add E r)))
    (coerce_sort_update (AccSort_update QuicksortSORT nat E PrepareLSORT r) nat
      d)
    (coerce_expression QuicksortVAR_DecSetoid
      (sort_update d nat (sort_update r nat QuicksortSORT))
      (sort_update d nat
(sort_update r nat (AccSort QuicksortSORT PrepareLSORT E)))
      (coerce_sort_update (coerce_sort_update Prepare_coerce' nat r) nat d)
      ([sigma'3:FS (sort_update d nat (sort_update r nat QuicksortSORT))]
sigma'3 X (sigma'3 c))))))))
  ([Z:qsSTATE']
    [sigma:STATE QuicksortSORT
      (sort_update exch nat (sort_update d nat (sort_update r nat PrepareLSORT)))]
    Assertion_update (Env_add Erd exch) X
    (coerce_expression' QuicksortVAR_DecSetoid X
      (sort_update exch nat
(AccSort QuicksortSORT (sort_update d nat (sort_update r nat PrepareLSORT)) Erd))
      (AccSort QuicksortSORT
(sort_update exch nat (sort_update d nat (sort_update r nat PrepareLSORT)))
(Env_add Erd exch))
      (AccSort_update QuicksortSORT nat Erd
(sort_update d nat (sort_update r nat PrepareLSORT)) exch)
      (coerce_expression' QuicksortVAR_DecSetoid X
(sort_update exch nat
  (sort_update d nat
    (AccSort QuicksortSORT (sort_update r nat PrepareLSORT) (Env_add E r))))
(sort_update exch nat
  (AccSort QuicksortSORT (sort_update d nat (sort_update r nat PrepareLSORT))
    (Env_add (Env_add E r) d)))
(coerce_sort_update (AccSort_update QuicksortSORT nat (Env_add E r)
      (sort_update r nat PrepareLSORT) d) nat exch)
(coerce_expression' QuicksortVAR_DecSetoid X
  (sort_update exch nat
    (sort_update d nat (sort_update r nat (AccSort QuicksortSORT PrepareLSORT E))))
  (sort_update exch nat
    (sort_update d nat
      (AccSort QuicksortSORT (sort_update r nat PrepareLSORT) (Env_add E r))))
  (coerce_sort_update (coerce_sort_update (AccSort_update QuicksortSORT nat E
    PrepareLSORT r) nat d) nat exch)
  (coerce_expression' QuicksortVAR_DecSetoid X
    (sort_update exch nat (sort_update d nat (sort_update r nat QuicksortSORT)))
    (sort_update exch nat
      (sort_update d nat (sort_update r nat (AccSort QuicksortSORT PrepareLSORT E))))
    (coerce_sort_update (coerce_sort_update (coerce_sort_update Prepare_coerce' nat r
      ) nat d) nat exch)
    ([sigma'3:FS (sort_update exch nat
   (sort_update d nat (sort_update r nat QuicksortSORT)))]
      update (sigma'3 X) (sigma'3 d) (sigma'3 exch))))))
    (Fr exch
      (Assertion_update Erd c
(coerce_expression' QuicksortVAR_DecSetoid c
  (sort_update d nat
    (AccSort QuicksortSORT (sort_update r nat PrepareLSORT) (Env_add E r)))
  (AccSort QuicksortSORT (sort_update d nat (sort_update r nat PrepareLSORT))
    (Env_add (Env_add E r) d))
  (AccSort_update QuicksortSORT nat (Env_add E r) (sort_update r nat PrepareLSORT) d)
  (coerce_expression' QuicksortVAR_DecSetoid c
    (sort_update d nat (sort_update r nat (AccSort QuicksortSORT PrepareLSORT E)))
    (sort_update d nat
```

```
      (AccSort QuicksortSORT (sort_update r nat PrepareLSORT) (Env_add E r)))
    (coerce_sort_update (AccSort_update QuicksortSORT nat E PrepareLSORT r) nat d)
    (coerce_expression' QuicksortVAR_DecSetoid c
      (sort_update d nat (sort_update r nat QuicksortSORT))
      (sort_update d nat (sort_update r nat (AccSort QuicksortSORT PrepareLSORT E)))
      (coerce_sort_update (coerce_sort_update Prepare_coerce' nat r) nat d)
      ([sigma'3:FS (sort_update d nat (sort_update r nat QuicksortSORT))]
suc (sigma'3 c)))))
(Assertion_update Erd d
  (coerce_expression' QuicksortVAR_DecSetoid d
    (sort_update d nat
      (AccSort QuicksortSORT (sort_update r nat PrepareLSORT) (Env_add E r)))
    (AccSort QuicksortSORT (sort_update d nat (sort_update r nat PrepareLSORT))
      (Env_add (Env_add E r) d))
    (AccSort_update QuicksortSORT nat (Env_add E r) (sort_update r nat PrepareLSORT)
      d)
    (coerce_expression' QuicksortVAR_DecSetoid d
      (sort_update d nat (sort_update r nat (AccSort QuicksortSORT PrepareLSORT E)))
      (sort_update d nat
(AccSort QuicksortSORT (sort_update r nat PrepareLSORT) (Env_add E r)))
      (coerce_sort_update (AccSort_update QuicksortSORT nat E PrepareLSORT r) nat d)
      (coerce_expression' QuicksortVAR_DecSetoid d
(sort_update d nat (sort_update r nat QuicksortSORT))
(sort_update d nat (sort_update r nat (AccSort QuicksortSORT PrepareLSORT E))
  ) (coerce_sort_update (coerce_sort_update Prepare_coerce' nat r) nat d)
([sigma'3:FS (sort_update d nat (sort_update r nat QuicksortSORT))]
  pred (sigma'3 d)))))
  ([Z'3:qsSTATE']
    [tau:STATE QuicksortSORT (sort_update d nat (sort_update r nat PrepareLSORT))]
    (Spec_Prepare_inv ([sigma'5:qsSTATE]local sigma'5 d) Z'3.1.1.1
      (drPrepareLSORT2qsSTATE tau) ∧
      Lt (suc (lookup Erd tau d) - lookup Erd tau c) t)))) vexch) Z
    (STATE_update (Env_add Erd exch) X
      (eval (Env_add Erd exch) sigma
(coerce_expression' QuicksortVAR_DecSetoid X
  (sort_update exch nat
    (AccSort QuicksortSORT (sort_update d nat (sort_update r nat PrepareLSORT)) Erd)
    )
  (AccSort QuicksortSORT
    (sort_update exch nat (sort_update d nat (sort_update r nat PrepareLSORT)))
    (Env_add Erd exch))
  (AccSort_update QuicksortSORT nat Erd
    (sort_update d nat (sort_update r nat PrepareLSORT)) exch)
  (coerce_expression' QuicksortVAR_DecSetoid X
    (sort_update exch nat
      (sort_update d nat
(AccSort QuicksortSORT (sort_update r nat PrepareLSORT) (Env_add E r))))
    (sort_update exch nat
      (AccSort QuicksortSORT (sort_update d nat (sort_update r nat PrepareLSORT))
(Env_add (Env_add E r) d)))
    (coerce_sort_update (AccSort_update QuicksortSORT nat (Env_add E r)
  (sort_update r nat PrepareLSORT) d) nat exch)
    (coerce_expression' QuicksortVAR_DecSetoid X
      (sort_update exch nat
(sort_update d nat (sort_update r nat (AccSort QuicksortSORT PrepareLSORT E))
  ))
      (sort_update exch nat
(sort_update d nat
  (AccSort QuicksortSORT (sort_update r nat PrepareLSORT) (Env_add E r))))
      (coerce_sort_update (coerce_sort_update (AccSort_update QuicksortSORT nat E
PrepareLSORT r) nat d) nat exch)
      (coerce_expression' QuicksortVAR_DecSetoid X
(sort_update exch nat (sort_update d nat (sort_update r nat QuicksortSORT)))
(sort_update exch nat
  (sort_update d nat
    (sort_update r nat (AccSort QuicksortSORT PrepareLSORT E))))
(coerce_sort_update (coerce_sort_update (coerce_sort_update Prepare_coerce'
  nat r) nat d) nat exch)
([sigma'3:FS (sort_update exch nat
       (sort_update d nat (sort_update r nat QuicksortSORT)))]
```

```
  update (sigma'3 X) (sigma'3 c) (sigma'3 X (sigma'3 d)))))))) sigma));
...

Discharge t;


Module  Prepare_vc4 Import quicksort_spec;

[vr,vd:nat];

Goal Pvc4 :
  SubRel ([Z:qsSTATE']
   [tau:STATE QuicksortSORT
     (sort_update d nat (sort_update r nat PrepareLSORT))]
   (Spec_Prepare_inv ([sigma:qsSTATE]local sigma d) Z.1.1.1
     (drPrepareLSORT2qsSTATE tau) ∧
     is_false (eval Erd tau
([sigma:FS (AccSort QuicksortSORT
     (sort_update d nat
       (sort_update r nat PrepareLSORT))
     (Env_add (Env_add E r) d))]
  le (sigma c) (sigma d)))))
  (Fr d (Fr r Spec_Prepare_post vr) vd);
...

Discharge vr;


Module  Prepare_vc5 Import quicksort_spec;

[vexch:PrepareLSORT exch];

Goal Pvc5 :
  SubRel ([Z:qsSTATE']
                 [sigma:STATE QuicksortSORT (sort_update exch nat PrepareLSORT)]
                 (Fr exch
                   ([Z'3:qsSTATE'][sigma'4:STATE QuicksortSORT PrepareLSORT]
                    (Spec_Prepare_pre Z'3 sigma'4 ∧
                     is_true (eval E sigma'4
                               ([sigma'5:FS (AccSort QuicksortSORT PrepareLSORT E)]
                                lt (sigma'5 X (sigma'5 b)) (sigma'5 X (sigma'5 a))))))
                   vexch Z sigma ∧
                  Eq (coerce (sort_update_lookup exch nat PrepareLSORT)
                       (local sigma exch))
                   (eval E (STATE_remove_local exch vexch sigma)
                     ([sigma'3:FS (AccSort QuicksortSORT PrepareLSORT E)]
                      sigma'3 X (sigma'3 a)))))
          ([Z:qsSTATE'][sigma:STATE QuicksortSORT (sort_update exch nat PrepareLSORT)]
           Assertion_update (Env_add E exch) X
            ([sigma'3:FS (AccSort QuicksortSORT (sort_update exch nat PrepareLSORT)
                           (Env_add E exch))]
             update (sigma'3 X) (sigma'3 b) (sigma'3 exch))
            (Fr exch
              ([Z'3:qsSTATE'][sigma'4:STATE QuicksortSORT PrepareLSORT]
               Assertion_update (Env_add E r) c
                ([sigma'5:FS (AccSort QuicksortSORT (sort_update r nat PrepareLSORT)
                               (Env_add E r))]suc (sigma'5 a))
                ([Z'5:qsSTATE']
                 [sigma'6:STATE QuicksortSORT (sort_update r nat PrepareLSORT)]
                 Spec_Prepare_inv ([sigma'7:qsSTATE]local sigma'7 d) Z'5.1.1.1
                  (drPrepareLSORT2qsSTATE (STATE_add_local (Env_add E r) sigma'6 d
                                            ([sigma'7:FS (AccSort QuicksortSORT
                                                           (sort_update r nat
                                                             PrepareLSORT) (Env_add E r)
                                                         )]pred (sigma'7 b))))) Z'3
                (STATE_add_local E sigma'4 r
                  ([sigma'5:FS (AccSort QuicksortSORT PrepareLSORT E)]
                   sigma'5 X (sigma'5 a)))) vexch) Z
              (STATE_update_internal QuicksortSORT (sort_update exch nat PrepareLSORT)
```

```
                (Env_add E exch) X
                (eval (Env_add E exch) sigma
                  ([sigma'3:FS (AccSort QuicksortSORT (sort_update exch nat PrepareLSORT)
                                (Env_add E exch))]
                   update (sigma'3 X) (sigma'3 a) (sigma'3 X (sigma'3 b)))) sigma));
...

Discharge vexch;



Module Prepare_vc6 Import quicksort_spec;

Goal Pvc6 :
  SubRel ([Z:qsSTATE'][sigma:STATE QuicksortSORT PrepareLSORT]
                  (Spec_Prepare_pre Z sigma ∧
                   is_false (eval E sigma
                              ([sigma'3:FS (AccSort QuicksortSORT PrepareLSORT E)]
                               lt (sigma'3 X (sigma'3 b)) (sigma'3 X (sigma'3 a))))))
           ([Z:qsSTATE'][sigma:STATE QuicksortSORT PrepareLSORT]
            Assertion_update (Env_add E r) c
             ([sigma'3:FS (AccSort QuicksortSORT (sort_update r nat PrepareLSORT)
                           (Env_add E r))]suc (sigma'3 a))
             ([Z'3:qsSTATE']
              [sigma'4:STATE QuicksortSORT (sort_update r nat PrepareLSORT)]
              Spec_Prepare_inv ([sigma'5:qsSTATE]local sigma'5 d) Z'3.1.1.1
               (drPrepareLSORT2qsSTATE (STATE_add_local (Env_add E r) sigma'4 d
                                        ([sigma'5:FS (AccSort QuicksortSORT
                                                      (sort_update r nat PrepareLSORT)
                                                      (Env_add E r))]pred (sigma'5 b)))
                )) Z
             (STATE_add_local E sigma r
               ([sigma'3:FS (AccSort QuicksortSORT PrepareLSORT E)]sigma'3 X (sigma'3 a))
              ));
...



Module Prepare_correct
  Import quicksort_spec Hadmiss
        Prepare_vc1 Prepare_vc2 Prepare_vc3
     Prepare_vc4 Prepare_vc5 Prepare_vc6;

[Gamma : HContext QuicksortSORT PrepareLSORT];

(*  Corresponds to Lemma 4.15 on page 121. *)
Goal Prepare_correct :
  HD S0 E Gamma Spec_Prepare_pre (coerce_prog Prepare_coerce' Prepare)
          Spec_Prepare_post;

Refine HDSeq;
Refine -0 HDNew_LC; intros -0 vr;
Refine -0 HDSeq;
Refine -0 HDNew_LC;

Intros -1 z sigma;
  Refine Spec_Prepare_inv ([sigma:qsSTATE]sigma.local d) z.1.1.1;
  Refine drPrepareLSORT2qsSTATE sigma;

intros -0 vd Erd; Refine -0 HDConr;
Refine -0 HDWhile;

  (** termination measure **)
  Refine -2 complete_induction;
  Intros -1 sigma;
    Refine (suc (lookup Erd sigma d)) - (lookup Erd sigma c);


intros -0 Then
  Repeat (Refine -0 HDIfthenelse Else Refine -0 HDSeq Else
          Refine -0 HDNew Else Refine -0 HDAssign);
```

```
Refine HDAssign_lemma; Refine +1 HDAssign_lemma;

Refine Pvc1; Refine Pvc2;
intros vexch; Refine HDSeq; Refine -0 HDAssign;
Refine HDAssign_lemma;
Refine Pvc3;

Expand coerce_expression coerce_expression';
Refine Pvc4;

Refine HDAssign;

Repeat (Refine HDIfthenelse Else Refine HDSeq Else
          Refine HDNew Else Refine HDAssign);

intros vexch; Refine HDSeq; Refine -0 HDAssign;
Refine HDAssign_lemma;

Expand coerce_expression coerce_expression'; Expand STATE_update;
Refine Pvc5;

Refine HDInv_lemma;

Expand coerce_expression coerce_expression';
Refine Pvc6;

SaveFrozen Prepare_correct;
(*Show*)

Discharge Gamma;



Module quicksort Import lib_nat_rels Prepare_correct;

[eq = QuicksortVAR_DecSetoid.DecSetoid_eq]
[E':Env QuicksortVAR_DecSetoid]
[EQS [x:QuicksortVAR_DecSetoid.DecSetoid_carrier] =
  andalso (inv (eq x Pa)) (andalso (inv (eq x Pb)) (andalso (inv (eq x X))
  (E' x)))]

[SQP = [z,sigma:qsSTATE]
       Spec_Quicksort_post (z.global Pa) (z.global Pb)
                           (sigma.global X) (z.global X)];

Freeze Prepare;

(*  Corresponds to Lemma 4.14 on page 120. *)

Goal quicksort_correct :
  HD  S0 EQS (HNone ??) (Spec_Quicksort_pre EQS) (call ?) SQP;
...
```

B.6 Additional Background Theory

```
Module lib_eq Import lib_logic;

[T|Type];

(*  Corresponds to Definition A.3 on page 135. *)
[Eq = [x,y:T]{P:T->Prop}(P x)->P y : T->T->Prop];

[Eq_refl = [t:T][P:T->Prop][h:P t]h : refl Eq];

(*  Corresponds to Axiom 1 on page 135. *)
$[Eq_subst : {x,y|T}{p:Eq x y}{C:T->Type}(C(x)) -> C(y)];
```

```
DischargeKeep T;

$[Eq_subst_comp: {C:T->Type}{a:T}{b:C a}Eq (Eq_subst (Eq_refl a) C b) b];

[[T|Type][C:T->Type][a:T][b:C a]
 (Eq_subst (Eq_refl a) C) b ==> b
];

(*  Corresponds to Axiom 2 on page 136. *)
$[Pr_Ir:{P|Prop}{x,y:P}Eq x y];

(*  Corresponds to Axiom 3 on page 137. *)
$[Extensionality_dep :
  {A|Type}{B|A->Type}{f,g:{x:A}B x}
  ({x:A}Eq (f x) (g x)) -> Eq f g]

[Extensionality [A,B|Type][f,g:A->B] = Extensionality_dep|A|([_:A]B) f g];

Discharge T;

(* Make Theorems as powerful as possible *)
[trivType={T|Type}T->T];
[emptyType={T|Type}T];
Configure Theorems trivType emptyType Eq_subst;
```

```
Module tms_rel Import lib_rel;

[T|Type];

[reflclose [R:Rel T T]
  = [tau,sigma:T](R tau sigma) ∨ (Eq tau sigma) : Rel T T];

Discharge T;
```

```
Module decsetoid Import lib_bool_thms;

[P|Type(0)][eq:P->P->bool]
[Eqchar  = {x,y:P}iff (Eq x y) (is_true (eq x y))];

Goal Eqchar' : Eqchar -> {x,y:P}iff (not (Eq x y)) (is_false (eq x y));
...

Discharge P;

(*  Corresponds to Definition 2.1 on page 10. *)
Record [DecSetoid : Type(0)]
Fields [DecSetoid_carrier|Type(0)]
       [DecSetoid_eq|DecSetoid_carrier->DecSetoid_carrier->bool]
       [DecSetoid_char:Eqchar DecSetoid_eq];

[decsetoid:DecSetoid];

Goal DecSetoid_eq_refl :
  {x:decsetoid.DecSetoid_carrier}is_true (decsetoid.DecSetoid_eq x x);
...

Goal DecSetoid_eq_sym :
  {x,y:decsetoid.DecSetoid_carrier}
  Eq (decsetoid.DecSetoid_eq x y) (decsetoid.DecSetoid_eq y x);
...

Discharge decsetoid;
```

```
Module DepUpdate Import lib_eq decsetoid;

Goal diagsubstlem :
  {A|Type(1)}{B:A->Type(1)}{a:A}{p:Eq a a}{b:B a}Eq (Eq_subst p B b) b;
...

Unfreeze Eq;

(*  Corresponds to Lemma A.4 on page 136. *)
Goal depsubst :
  {A:Type}{B:A->Type}{C:{a:A}(B a)->Type}
  {a,a':A}{p:Eq a a'}{b:B a}(C a' (Eq_subst p B b)) -> C a b;
intros _______;
Refine Eq_subst ? [a:A]({p:Eq a a'}{b:B a}(C a' (Eq_subst p B b))->C a b);
Refine +1 Eq_sym; Assumption +1;
intros p'; Qrepl Pr_Ir p' (Eq_refl a');
intros; Refine Eq_subst; Refine +1 Eq_subst_comp;
Immed;
Save depsubst;

Goal Eq_elim :
  {t|Type}{P:{x,y:t}(Eq x y)->Type}({x:t}P x x (Eq_refl x))->
  {x,y:t}{p:Eq x y}P x y p;
...

Goal Eq_subst_trans :
  {T|Type}{x,y|T}{xy:Eq x y}{yx:Eq y x}{C:T->Type}{b:C ?}
  Eq (Eq_subst xy C (Eq_subst yx C b)) b;
...

[A|Type][B,C|A -> Type][x,y|A][c1:Eq x y][c2:Eq (C x) (B x)];

Goal Eq_subst_trans_internal : Eq (C x) (B y);
...

DischargeKeep x;

Goal Eq_subst_trans' :
  {v:C x}
  Eq (Eq_subst c1 B (Eq_subst c2 (Id|Type(0)) v))
     (Eq_subst (Eq_subst_trans_internal c1 c2) (Id|Type(0)) v);
...

Discharge A;

[VAR|DecSetoid];
[SORT_internal = {x:VAR.DecSetoid_carrier}Type(0)]
[sort:SORT_internal]
[FS = {x:VAR.DecSetoid_carrier}sort x];

Goal sort_update :
  {y:VAR.DecSetoid_carrier}{T:Type(0)}{f:SORT_internal}SORT_internal;
Intros; Refine if (VAR.DecSetoid_eq x y) T (f x);
Save sort_update;

Goal sort_update_lookup :
  {y:VAR.DecSetoid_carrier}{T:Type(0)}{f:SORT_internal}
  Eq ((sort_update y T f) y) T;
...

Goal sort_update_lemma' :
  {x,y|VAR.DecSetoid_carrier}{different:is_false (VAR.DecSetoid_eq x y)}
  {T:Type(0)}{f:SORT_internal}Eq ((sort_update y T f) x) (f x);
...

[x:VAR.DecSetoid_carrier][t:sort x][sigma:FS];

Goal FS_update_internal : FS;
intros; Intros y;
Refine bool_elim [c:bool](Eq (VAR.DecSetoid_eq y x) c)->sort y;
```

```
  Refine -0 Eq_refl;

  (* x=y *)
  intros xEQy; Refine Eq_subst (Eq_sym (snd (VAR.DecSetoid_char ??) xEQy));
  Refine t;

  (* x<>y *)
  intros; Refine sigma y;
Save FS_update_internal;

Goal FS_update_lemma : Eq (FS_update_internal x) t;
Expand FS_update_internal;

[B [b:bool][p:Eq (VAR.DecSetoid_eq x x) b]
  = Eq (bool_elim ([c:bool](Eq (DecSetoid_eq VAR x x) c)->sort x)
             ([xEQy:Eq (DecSetoid_eq VAR x x) true]
              Eq_subst (Eq_sym (snd (DecSetoid_char VAR x x) xEQy)) sort t)
             ([_:Eq (DecSetoid_eq VAR x x) false]sigma x)
           b p) t];

Equiv B ? ?;
Refine depsubst ??;

Refine true; Refine DecSetoid_eq_refl;
Refine diagsubstlem;
Save FS_update_lemma;

Goal FS_update_lemma' :
  {y:VAR.DecSetoid_carrier}{different:not (Eq x y)}
  Eq (FS_update_internal y) (sigma y);
intros; Expand FS_update_internal;

[B [b:bool][p:Eq (VAR.DecSetoid_eq y x) b]
  = Eq (bool_elim ([c:bool](Eq (DecSetoid_eq VAR y x) c)->sort y)
             ([xEQy:Eq (DecSetoid_eq VAR y x) true]
              Eq_subst (Eq_sym (snd (DecSetoid_char VAR y x) xEQy)) sort t)
             ([_:Eq (DecSetoid_eq VAR y x) false]sigma y)
            b p) (sigma y)];

Equiv B ??; Refine depsubst;
Refine +1 fst (Eqchar' ? VAR.DecSetoid_char ??);
Intros _; Refine different; Refine Eq_sym; Immed;
Expand B; Refine Eq_refl;
Save FS_update_lemma';

Discharge sort;

Goal FS_lookup_update :
  {sort|SORT_internal}{y:VAR.DecSetoid_carrier}{sigma:FS sort}
  Eq (FS_update_internal sort y (sigma y) sigma)
     sigma;
...

Goal FS_update_dyn
  : {sort|SORT_internal}{y:VAR.DecSetoid_carrier}
    {T|Type(0)}{t:T}{sigma:FS sort}
    FS (sort_update y T sort);
Intros; Expand sort_update;
Induction (DecSetoid_eq VAR x y);

  (** x=y **)
  Refine t;

  (** x<>y **)
  Refine sigma x;
Save FS_update_dyn;

Goal FS_update_dyn'
  : {sort|SORT_internal}{y:VAR.DecSetoid_carrier}{T|Type(0)}{t:sort y}
    {sigma:FS (sort_update y T sort)}FS sort;
```

```
...

Goal FS_update_dyn'_update_dyn :
  {sort|SORT_internal}{y:VAR.DecSetoid_carrier}{t:y.sort}{U|Type(0)}
  {u:U}{sigma:FS sort}
  Eq (FS_update_dyn' y t (FS_update_dyn y u sigma))
     (FS_update_internal sort y t sigma);
...

Goal coerce : {T,U|Type}{eq:Eq T U}{a:T}U;
  intros; Qrepl Eq_sym eq; Immed;
Save coerce;

Goal coerce2 : {T,U|Type(2)}{eq:Eq T U}{a:T}U;
  intros; Qrepl Eq_sym eq; Immed;
Save coerce2;

Goal ifcoerce :
  {b,c|bool}{T,U|Type}{eq:Eq b c}Eq (if b T U) (if c T U);
...

Goal  FS_update_dyn'_lookup :
  {vardecl | (DecSetoid_carrier VAR)->Type(0)}{T | Type(0)}
  {x : DecSetoid_carrier VAR}{v : vardecl x}
  {sigma : FS (sort_update x T vardecl)}
  Eq (FS_update_dyn' x v sigma x) v;
...

[sort|SORT_internal][y:VAR.DecSetoid_carrier]
  [U|Type(0)][v:U][sigma:FS sort]
  $[c = sort_update_lookup y U sort];

Goal FS_update_dyn_lookup
  : Eq (FS_update_dyn y v sigma y)
       (coerce (Eq_sym (sort_update_lookup y U sort)) v);
...

Goal FS_lookup_update_dyn
  : Eq (coerce (sort_update_lookup y U sort) (FS_update_dyn y v sigma y)) v;
...

[x|VAR.DecSetoid_carrier];

Goal FS_update_dyn_lemma':
  {different:is_false (VAR.DecSetoid_eq y x)}
  Eq (FS_update_dyn x v sigma y)
     (Eq_subst (Eq_sym (sort_update_lemma' different U sort))
       (Id|Type(0)) (sigma y));
...



Discharge sort;

Goal FS_update_dyn_dyn' :
  {vardecl | (DecSetoid_carrier VAR)->Type(0)}{T | Type(0)}
  {x : DecSetoid_carrier VAR}{v : vardecl x}
  {sigma : FS (sort_update x T vardecl)}
  Eq sigma
    (FS_update_dyn x (coerce (sort_update_lookup x T vardecl) (sigma x))
                   (FS_update_dyn' x v sigma));
...

[EXTEQ [sortA,sortB:SORT_internal]
  = {x:DecSetoid_carrier VAR}Eq (sortA x) (sortB x)];

Goal EXTEQrefl : {sort:SORT_internal}EXTEQ sort sort;
...

Goal sort_dupdate :
```

```
  {y:DecSetoid_carrier VAR}{sort:SORT_internal}{T:Type(0)}
  EXTEQ (sort_update y (sort y) (sort_update y T sort)) sort;
...


[lsortA|SORT_internal];

Goal FS_coerce :
  {lsortB|SORT_internal}
  {coerce:EXTEQ lsortA lsortB}{f:FS lsortA}FS lsortB;

Intros; Refine Eq_subst (coerce1 x) [T:Type]T; Refine f x;
Save FS_coerce;

Goal FS_coerce_redundant :
  {f:FS lsortA}Eq (FS_coerce (EXTEQrefl ?) f) f;
...

Discharge lsortA;

[y:VAR.DecSetoid_carrier][T:Type(0)][sort:SORT_internal];
[v:sort y][sigma:FS (sort_update y T sort)];

Goal FS_update_dyn'2dyn
  : Eq (FS_update_dyn' y v sigma)
       (FS_coerce (sort_dupdate y sort T) (FS_update_dyn y v sigma));
...

[FS_update [sort|SORT_internal] = FS_update_internal sort];

Discharge VAR;

[SORT [VAR:DecSetoid] = SORT_internal|VAR];

Goal sort_update_can :
  {VAR|DecSetoid}{C:(SORT VAR)->Prop}{sort|SORT VAR}
  {x,y|VAR.DecSetoid_carrier}{Tx,Ty:Type(0)}
  {sort_update_refl : C (sort_update x Tx sort)}
  {sort_update_sym  : C (sort_update y Ty (sort_update x Tx sort))}
  C (sort_update x Tx (sort_update y Ty sort));
...
```

```
(* Transitive Closure of a Relation *)

Module TransClosure Import lib_rel;

Inductive [TransClosure : T->T->Prop]
Relation NoReductions Inversion
Parameters [T|Type][R:Rel T T]

Constructors
  [TransClosure_emb : {x,y:T}(R x y) -> TransClosure x y]
  [TransClosure_plus: {x,y,z:T}(R x y) -> (TransClosure y z) ->
                                    TransClosure x z];

Goal TransClosureTrans : trans TransClosure;
...

Discharge T;
```

Bibliography

Abrial, J.-R. (1996), *The B Book - Assigning Programs to Meanings*, Cambridge University Press.

Aczel, P. (1982*a*), A note on program verification. Unpublished, see also (Jones 1986).

Aczel, P. (1982*b*), A system of proof rules for the correctness of iterative programs – some notational and organisational suggestions. Unpublished.

Ah-kee, J. A. (1990), 'Proof obligations for blocks and procedures', *Formal Aspects of Computing* **2**, 312–330.

America, P. & de Boer, F. (1990), 'Proving total correctness of recursive procedures', *Information and Computation* **84**(2), 129–162.

Apt, K. R. (1981), 'Ten years of Hoare's logic: A survey – part I', *ACM Transactions on Programming Languages and Systems* **3**(4), 431–483.

Apt, K. R. & Meertens, L. G. L. T. (1980), 'Completeness with finite systems of intermediate assertions for recursive program schemes', *SIAM Journal on Computing* **9**(4), 665–671.

Apt, K. R. & Olderog, E.-R. (1991), *Verification of Sequential and Concurrent Programs*, Texts and Monographs in Computer Science, Springer, New York.

Bergstra, J. A. & Tucker, J. V. (1982), 'Some natural structures which fail to prossess a sound and decidable Hoare-like logic for their while-programs', *Theoretical Computer Science* **17**, 303–315.

Burstall, R. & McKinna, J. (1993), Deliverables: a categorical approach to program development in type theory, *in* A. M. Borzyszkowski & S. Sokołowski, eds, 'Proceedings of MFCS '93', Vol. 711 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 32–67. An earlier version appeared as LFCS technical report ECS-LFCS-92-242 in 1992.

Caplan, J. E. (1995), Formalizing Hoare logic, *in* U. S. Reddy, ed., 'Proceedings of SIPL '95', University of Illinois at Urbana-Champaign, pp. 1–18. Available as Technical Report UIUCDCS-R-95-1900.
*http://acsl.cs.uiuc.edu/~caplan/hllf/sipl95/sipl95.dvi

Clarke Jr., E. M. (1979), 'Programming language constructs for which it is impossible to obtain good Hoare axiom systems', *Journal of the ACM* **26**(1), 129–147.

Clarke, Jr., E. M., German, S. M. & Halpern, J. Y. (1983), 'Effective axiomatizations of Hoare logics', *Journal of the ACM* **30**(3), 612–636.

Clint, M. (1981), 'On the use of history variables', *Acta Informatica* **16**, 15–30.

Cook, S. A. (1978), 'Soundness and completeness of an axiom system for program verification', *SIAM Journal on Computing* **7**(1), 70–90.

Coq (1998), 'The Coq project'.
*http://pauillac.inria.fr/coq/

Coquand, T., Nordström, B., Smith, J. M. & von Sydow, B. (1994), 'Type theory and programming', *EATCS Bulletin* **52**, 203–228.

Cousot, P. (1990), Methods and logics for proving programs, *in* J. van Leeuwen, ed., 'Handbook of Theoretical Computer Science', Vol. B: Formal Models and Semantics, Elsevier, chapter 15, pp. 841–993.

Dahl, O.-J. (1992), *Verifiable Programming*, International Series in Computer Science, Prentice Hall.

Dahl, O.-J., Dijkstra, E. & Hoare, C. A. R. (1972), *Structured Programming*, Academic Press.

de Bakker, J. (1980), *Mathematical Theory of Program Correctness*, Prentice Hall.

Dershowitz, N. & Manna, Z. (1979), 'Proving termination with multiset orderings', *Communications of the ACM* **22**(8), 465–475.

Dijkstra, E. W., ed. (1990), *Formal Development of Programs and Proofs*, University of Texas at Austin Year of Programming Series, Addison-Wesley.

Floyd, R. (1967), Assigning meanings to programs, *in* J. T. Schwartz, ed., 'Proc. Symp. in Applied Mathematics', Vol. 19, pp. 19–32.

Foley, M. & Hoare, C. A. R. (1971), 'Proof of a recursive program: QUICKSORT', *The Computer Journal* **14**(4), 391–395.

Gehani, N. & MacGettrick, A., eds (1986), *Software Specification Techniques*, Addison-Wesley.

Goguen, H. (1994), The metatheory of UTT, *in* P. P. Dybjer, B. Nordström & J. M. Smith, eds, 'Types for Proofs and Programs, International Workshop (TYPES '94); Bastad, Sweden', Vol. 996 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 60–82.

Gordon, M. J. (1989), Mechanizing programming logics in higher order logic, *in* G. Birtwhistle & P. Subrahmanyam, eds, 'Current Trends in Hardware Verification and Automated Theorem Proving (Banff, Alberta)', number 15 *in* 'Workshops in Computing', Springer-Verlag, pp. 387–439.

Gorelick, G. A. (1975), A complete axiomatic system for proving assertions about recursive and non-recursive programs, Technical Report 75, Department of Computer Science, University of Toronto.

Grabowski, M. (1985), On the relative incompleteness of logics, *in* Parikh (1985), pp. 118–127. Extended Abstract.

Gries, D. (1981), *The Science of Computer Programming*, Springer.

Gries, D. (1982), 'A note on a standard strategy for developing loop invariants and loops', *Science of Computer Programming* **2**, 207–214.

Gries, D. & Levin, G. M. (1980), 'Assignment and procedure call proof rules', *ACM Transactions on Programming Languages and Systems* **2**(4), 564–579.

Harel, D. (1980), 'Proving the correctness of regular deterministic programs: A unifying survey using dynamic logic', *Theoretical Computer Science* **12**, 61–81.

Harper, R., Honsell, F. & Plotkin, G. (1993), 'A framework for defining logics', *Journal of the ACM* **40**(1), 143–184. Preliminary version appeared in Proc. 2nd IEEE Symposium on Logic in Computer Science, 1987, 194–204.

Hoare, C. A. R. (1961), 'Algorithm 63, partition; algorithm 64, quicksort; algorithm 65, find', *Communications of the ACM* **4**(7), 321–322.

Hoare, C. A. R. (1969), 'An axiomatic basis for computer programming', *Communications of the ACM* **12**, 576–580. Also in (Hoare & Jones 1989).

Hoare, C. A. R. (1971), Procedures and parameters: An axiomatic approach, *in* E. Engeler, ed., 'Symposium on Semantics of Algorithmic Languages', Vol. 188 of *Lecture Notes in Mathematics*, Springer-Verlag, pp. 102–116. Also in (Hoare & Jones 1989).

Hoare, C. A. R. & Jones, C. B., eds (1989), *Essays in Computing Science*, International Series in Computer Science, Prentice Hall.

Hofmann, M. (1995), Extensional concepts in intensional type theory, PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh.
*http://www.dcs.ed.ac.uk/lfcsreps/EXPORT/95/ECS-LFCS-95-327/

Hofmann, M. (1997), 'Semantik und Verifikation'. Lecture Notes for a course held in Winter term 1997/98 at the University of Marburg. In German.
*http://www.mathematik.tu-darmstadt.de/~mh/skript.ps.gz

HOL (1998), 'The HOL system'.
*http://www.cl.cam.ac.uk/Research/HVG/HOL/

Homeier, P. V. (1995), Trustworthy Tools for Trustworthy Programs: A Mechanically Verified Verification Condition Generator for the Total Correctness of Procedures, PhD thesis, University of California, Los Angeles.
*http://www.cis.upenn.edu/~homeier/phd.html

Homeier, P. V. & Martin, D. F. (1996), Mechanical verification of mutually recursive procedures, *in* M. A. McRobbie & J. K. Slaney, eds, 'Automated Deduction – CADE-13', Vol. 1104 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, New Brunswick, NJ, USA, pp. 201–215. 13th International Conference on Automated Deduction.

Huet, G. & Plotkin, G. D., eds (1990), *Electronic Proceedings of the First Annual BRA Workshop on Logical Frameworks (Antibes, France)*.
*http://www.dcs.ed.ac.uk/lfcsinfo/research/types_bra/proc/index.html

Ireland, A. & Stark, J. (1997), On the Automatic Discovery of Loop Invariants, *in* 'Proceedings of the Fourth NASA Langley Formal Methods Workshop – NASA Conference Publication 3356'. Also available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/1.

Jones, C. B. (1986), Systematic program development, *in* Gehani & MacGettrick (1986), pp. 89–109.

Jones, C. B. (1990), *Systematic Software Development Using VDM*, International Series in Computer Science, 2 edn, Prentice Hall.

Jones, C. B. & Shaw, R. C. (1990), *Case Studies in Systematic Software Development*, Prentice Hall.

Knuth, D. E. (1986), *The TEXbook*, Addison-Wesley.

Lauer, P. E. (1971), Consistent formal theories of the semantics of programming languages, Technical Report TR.25.121, IBM Lab, Vienna.

Lego (1998), 'The LEGO proof assistant'.
*http://www.dcs.ed.ac.uk/home/lego

Liskov, B. H. & Berzins, V. (1986), An appraisal of program specifications, *in* Gehani & MacGettrick (1986), pp. 3–23.

London, R. L., Guttag, J. V., Horning, J. J., Lampson, B. W., Mitchell, J. G. & Popek, G. J. (1978), 'Proof rules for the programming language Euclid', *Acta Informatica* **10**, 1–26.

Luo, Z. (1990), An extended Calculus of Constructions, Thesis ECS-LFCS-90-118, Laboratory for Foundations of Computer Science, University of Edinburgh.

Luo, Z. (1994), *Computation and Reasoning: A Type Theory for Computer Science*, Oxford University Press.

Manna, Z. & Pnueli, A. (1974), 'Axiomatic approach to total correctness of programs', *Acta Informatica* **3**, 243–263.

Martin-Löf, P. (1984), *Intuitionistic Type Theory*, Bibliopolis·Napoli.

Mason, I. A. (1987), Hoare's logic in the LF, Technical Report 32, Laboratory for Foundations of Computer Science, University of Edinburgh.

McBride, C. (1998), Inverting inductively defined relations in LEGO. Still to appear in the proceedings of TYPES'96.
*ftp://ftp.dcs.ed.ac.uk/pub/lego/McBrideTypes96.ps

McKinna, J. H. (1992), Delivarables: A Categorical Approach to Program Development in Type Theory, PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh.
*ftp://ftp.dcs.ed.ac.uk/pub/jhm/thesis.ps.gz

Morgan, C. (1988), 'Procedures, parameters, and abstraction: separate concerns', *Science of Computer Programming* **11**, 17–27.

Morris, J. H. (n.d.), Comments on "procedures and parameters". Undated and unpublished.

Nipkow, T. (1996), Winskel is (almost) right: Towards a mechanized semantics textbook, *in* V. Chandru & V. Vinay, eds, 'Proceedings of FSTTCS '96', Vol. 1180 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 180–192.
*http://www4.informatik.tu-muenchen.de/~nipkow/pubs/fsttcs96.dvi.gz

Nipkow, T. (1998), 'Winskel is (almost) right: Towards a mechanized semantics textbook', *Formal Aspects of Computing* **?** To appear. This is an extended version of (Nipkow 1996).
*http://www4.informatik.tu-muenchen.de/~nipkow/pubs/winskel.html

Norrish, M. (1996), Derivation of verification rules for C from operational definitions, *in* J. von Wright, J. Grundy & J. Harrison, eds, 'Supplementary Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics: TPHOLs'96', number 1 *in* 'TUCS General Publications', Turku Centre for Computer Science, pp. 69–75.

Norrish, M. (1997), An abstract dynamic semantics for C, Technical Report 421, Computer Laboratory, University of Cambridge.
*http://www.cl.cam.ac.uk/ftp/papers/reports/TR421-mn200-c-semantics.dvi.gz

Olderog, E.-R. (1981), 'Sound and complete Hoare-like calculi based on copy rules', *Acta Informatica* **16**, 161–197.

Olderog, E.-R. (1983), 'On the notion of expressiveness and the rule of adaptation', *Theoretical Computer Science* **24**, 337–347.

Pandya, P. & Joseph, M. (1986), 'A structure-directed total correctness proof rule for recursive procedure calls', *The Computer Journal* **29**(6), 531–537.

Parikh, R., ed. (1985), *Logics of Programs*, Vol. 193 of *Lecture Notes in Computer Science*, Springer-Verlag.

Paulson, L. C. (1990), ML *for the Working Programmer*, Cambridge University Press.

Paulson, L. C. & Nipkow, T. (1998), Isabelle.
*http://www.cl.cam.ac.uk/Research/HVG/isabelle.html

Plotkin, G. (1981), A structural approach to operational semantics, Report DAIMI FN-19, Computer Science Department, Aarhus University.

Pollack, R. (1992), Implicit syntax. An earlier version of this paper appeared in (Huet & Plotkin 1990).
*ftp://ftp.dcs.ed.ac.uk/pub/lego/ImplicitSyntax.ps.Z

Pollack, R. (1994), The Theory of LEGO, A Proof Checker for the Extended Calculus of Constructions, PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh.

Pollack, R. (1998), How to believe a machine-checked proof, *in* G. Sambin & J. M. Smith, eds, 'Twenty-Five Years of Constructive Type Theory', Oxford University Press. To appear.
*ftp://ftp.dcs.ed.ac.uk/pub/lego/pollack-belief.ps.gz

Reynolds, J. C. (1981), *The Craft of Programming*, International Series in Computer Science, Prentice Hall.

Reynolds, J. C. (1982), Idealized Algol and its specification logic, *in* D. Néel, ed., 'Tools & Notions for Program Construction', Cambridge University Press.

Rushby, J. (1998), 'SRI-CSL-PVS'.
*http://www.csl.sri.com/sri-csl-pvs.html

Schildt, H., American National Standards Institute, International Organization for Standardization, International Electrotechnical Commission & ISO/IEC JTC 1 (1990), *The annotated ANSI C standard: American National Standard for Programming Languages C: ANSI/ISO 9899-1990*, Osborne/McGraw-Hill, Berkeley, CA, USA.

Schreiber, T. (1997), Auxiliary variables and recursive procedures, *in* M. Bidoit & M. Dauchet, eds, 'Proceedings of TAPSOFT '97', Vol. 1214 of *Lecture Notes in Computer Science*, Springer-Verlag, Lille, France, pp. 697–711.
*http://www.dcs.ed.ac.uk/home/tms/lego/tapsoft97/

Sieber, K. (1981), A new Hoare-calculus for programs with recursive parameterless procedures, Technical Report A 81/02, Fachbereich 10 – Informatik, Universität des Saarlandes, Saarbrücken.

Sieber, K. (1985), A partial correctness logic for procedures, *in* Parikh (1985), pp. 320–342.

Sokołowski, S. (1977), Total correctness for procedures, *in* J. Gruska, ed., 'Sixth Mathematical Foundations of Computer Science (Tatranská Lomnica)', Vol. 53 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 475–483.

Takeuti, G. (1975), *Proof Theory*, North-Holland.

Tarlecki, A. (1985), 'A language of specified programs', *Science of Computer Programming* **5**, 59–81.

Trakhtenbrot, B. A., Halpern, J. Y. & Meyer, A. R. (1984), From denotational to operational and axiomatic semantics for ALGOL-like languages: An overview, *in* E. Clarke & D. Kozen, eds, 'Logics of Programs 1983', Vol. 164 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 474–500.

Underwood, J. (1997), 'Computer aided formal reasoning'. Lecture Notes. *http://www.dcs.ed.ac.uk/home/lego/html/CAFR/96-97/lecture.html

Vickers, S. (1991), Formal implementation, *in* J. A. McDermid, ed., 'Software engineer's reference book', Butterworth-Heinemann, London, chapter 25.

Wall, L., Christiansen, T. & Schwartz, R. L. (1996), *Programming Perl*, second edn, O'Reilly & Associates, Inc.

Winskel, G. (1993), *The Formal Semantics of Programming Languages*, MIT Press.

Index

Π, 5, 133
∘, 21
λ, 133
$\mapsto_E$, 89

AccSort, 89
adaptation
    rule of, 70–73
address, 78, 84, 88, 110
aliasing, 110
array, 15, 114–117
    sorted, 116
assertion, 3, 36, 42, 49–50, 90

begin , 15
binding
    dynamic, 85, 87, 99
    static, 85, 87, 99, 109
block, *see* local program variable

C, 55, 77
classical logic, 37, 38
completeness, 2, 46–50, 90
    of Hoare Logic, 32, 39–42, 65–69, 83, 103–107
    of VDM, 26–29, 75, 85, 110
    relative, 2, 47–50
completeness step, 67, 104
computation rule, 134
confidence, 3, 130
context, 59–61, 64
correctness
    partial, 18, 30, 51, 73
    total, 9, 18, 62–64, 73
correctness formula, 1, 20, 34, 35, 38, 52, 53, 74, 87, 98, 100, 102

dependent function
    updating, 137–139
dependent type, 4–5, 51
determinism, 16–17, 19, 27, 55

Edinburgh Logical Framework, 51
elimination principle, 134
embedding
    deep, 10, 52, 54, 55, 129
    shallow, 10, 14, 49, 52, 119
end, 15
environment, 89, 98, 100
equality, 135–137
    decidable, 10–11, 15, 137
    extensional, 137
    Leibniz, 135
exponentiation, 17, 23
expression, 13–14
expressiveness, 27, 46–50, 56

factorial, 35, 58, 69
freezing, *see* variable, program, freezing the value of

Gödels's incompleteness theorem, 47
Gentzen, 59
global, 88
global program variable
    backup of, 88, 93

Hilbert, 59, 60
Hoare Logic, 29–46, 59–75, 81–84, 98–110
  semantics, 30, 36, 47
HOL, 51

induction, 6
inductively defined data type, 5
inductively defined relation, 5–7, 61
interpretation, 12–13, 47–50, 52
  Herbrand-definable, 48
  standard, 48, 49, 55
  with finite domains, 48
invariant, 22, 24, 47, 50–51, 121
inversion, 6–7
Isabelle, 51

Java, 77

left-constructive, 70
left-maximal, 71
LEGO, 133
lexicographical binding, *see* binding, static
local, 88
local program variable, 75–112
  preserving the value of a, 95, 100
  uninitialised, 77, 119
lookup, 89

MGF, *see* Most General Formula
model theory, 47
Most General Formula, 27, 39–42, 55, 107
mutually recursive procedures, 73–75

non-interference, 53

operation decomposition, 20
ordered list, 114–115

parameter, 110–112
  call-by-name, 58, 110, 112, 114, 117, 119
  call-by-value, 58, 110–112, 117, 119
partition, 115–118
Perl, 77
permutation, 116
pivot element, 118, 120
Prepare, 117, 120, 121, 125–128
procedure
  header, 74
  non-recursive, 107
  recursive, 35, 57–75, 85–110
procedure call graph, 74
proof irrelevance, 136
proof theory, 47
pseudo-induction hypothesis, *see* completeness step

quantification, 64
  explicit, 35, 53
  implicit, 7, 35, 38, 81
Quicksort, 113–129

recursion, 134–135
recursive depth, 66
  preserving, 96
reference point, 42–44, 102

S_0, 58
scoping, 88, 91, 100
scoping intrusion, 99
semantics
  axiomatic, 1
  denotational, 17, 59, 65–66
  structural operational, 16–17, 59, 80, 91–98
side-effect, 54–55
software reuse, 35
soundness, 2

of Hoare Logic, 32, 39, 65, 81, 102
of VDM, 25–26, 75, 85, 110
specification
entrance, 74
input/output, 17, 29
specification logic, 53
stack
explicit, 78, 88
implicit, 79, 85, 88, 110
standard proof, 60
state space, 11–12, 78–79, 88
stepwise-refinement, 124
substitution, 135–136
dependent, 136
first-order, 12
in assertion, 10, 14, 31, 52–55, 64, 99
in program, 99
substitution lemma, 31, 55
syntax-directed methodology, 60

termination, 9, 17–18, 24, 27, 30–31, 62, 74, 114, 118, 122, 127, 135
type coercion, 93, 137

uL, 89, 100
updating, *see* substitution

variable
auxiliary, 3–4, 19, 34–43, 50–51, 62, 71, 72, 131
domain, 38, 41, 42, 100, 102
existential, 129
global, *see* global program variable
implicit, 133
local, *see* local program variable
program, 10–11
freezing the value of, 34, 46, 50, 87, 100
hooked, 20
VCG, *see* Verification Condition Generator
VDM, *see* Vienna Development Method
Verification Condition Generator, 56, 113, 122
Vienna Development Method, 19–29, 42–46, 50, 51, 75, 84–85, 110
semantics, 22

weakest precondition, 32, 40, 47, 56, 73
well-founded induction, 135
well-founded relation, 135