

Mechanical Verification of Compiler Correctness

David William John Stringer-Calvert

*A thesis submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science.*

University of York
Department of Computer Science

March 1998

To Sam

Abstract

A large investment is made in the development, testing, validation and verification of source code for critical applications. But there remains a *semantic gap* between the source code produced and the object code which is executed. Standards for the development of critical systems recognize this, and mandate either the use of a trusted compiler (one that has been proven to produce correct object code) or the demonstration that the object code is a correct refinement of the source code (a lengthy and complex process).

This thesis examines:

1. the extent to which tool support is an essential ingredient in proofs of compiler correctness;
2. the extent to which these proofs may be automated;
3. the relationship between the complexity of the source language and the proof effort required;
4. the scalability of a mechanical method of compiler verification.

To do this, we present the development of the proof of correctness of a compiler for a small imperative language Tosca, targeted at an imaginary assembler Aida. The work presented here is an extension of the Z specification and hand proof presented in Stepney's book "High Integrity Compilation — A Case Study", recasting the specification and proofs within the framework of the PVS specification and verification system from SRI International.

We also assess the lessons learnt in the translation of a hand treatment of the problem in a partial logic, to an automatic treatment in a total logic with a richly expressive type system; from hand proof to automatic checking, and the inter-relationship between the specification and the ability to automate the proof process.

Contents

Acknowledgments	1
Declaration	2
1 Introduction	3
1.1 Compiler Verification	5
1.2 Thesis	6
1.3 Map of the thesis	6
I Background Material and Review	8
2 Compilation	9
2.0.1 Bootstrapping	12
2.1 Semantics	12
2.1.1 Operational Semantics	12
2.1.2 Denotational Semantics	13
2.1.3 Axiomatic Semantics	13
2.1.4 Action Semantics	13
2.1.5 Problems with Semantic Definitions	14
2.2 Summary	15

3	Trusted Compilation	16
3.1	Overview	16
3.2	Object Code Verification	17
3.2.1	Yu's M68020 Verifier	17
3.2.2	A Lisp Compiler	19
3.2.3	Sizewell PPS	19
3.2.4	STERNOL	21
3.2.5	Summary	21
3.3	Compiler Generators	22
3.3.1	Early Work	22
3.3.2	MESS	23
3.3.3	CANTOR — A Verified Compiler Generator	23
3.3.4	Summary	24
3.4	Analytical Compiler Verification	24
3.4.1	Verified Pascal	26
3.4.2	CLI Verified Stack	26
3.4.3	ProCoS	27
3.4.4	Verifix	27
3.4.5	Verified Scheme	28
3.4.6	A Verified Assembler	29
3.4.7	A Demonstrably Correct Compiler	30
3.5	Summary	31
4	Tool Support	32
4.1	Formal Methods	32
4.2	Verification Support for Z	34
4.2.1	The Choice of PVS	37
4.3	PVS	38
4.3.1	Specification Language	39
4.3.2	Verification Engine	42
4.3.3	Further Information	45
4.4	Summary	45

II	Mechanizing the DCC Method	46
5	The Starting Point	47
5.1	The Languages	47
5.2	The DCC Method	48
5.3	Industrial Usage of the DCC Method	49
5.4	Example Semantic Functions — Assignment	50
5.5	Example Aida semantics	52
5.6	Example Compilation Function	53
5.7	Proofs	53
5.8	Summary	58
6	From Z to PVS	59
6.1	Translation to PVS	59
6.1.1	PVS Types	60
6.1.2	Total functions	61
6.1.3	Initial Environment	63
6.2	Changes to the Original Semantics	64
6.2.1	Environment and Store	64
6.2.2	Aida State	66
6.2.3	Consequences	69
6.2.4	Exceptional Behaviour	70
6.2.5	Termination	71
6.2.6	Labels and Continuations	75
6.3	Summary	76
7	Mechanizing the DCC Proofs	77
7.1	Initial Proof Attempts	77
7.2	Assignment	78
7.3	Initial Strategy Development	101
7.4	Limitation of the Strategy Approach	103
7.5	Are Strategies Necessary?	105
7.6	Summary	106
7.7	An Aside — Proof Presentation	106

III	Extending Tosca	108
8	Language Extension — Local Scope	109
8.1	Tosca Semantics	112
8.1.1	Declaration Before Use Semantics	112
8.1.2	Type Checking Semantics	113
8.1.3	Dynamic Semantics	114
8.2	Compiler	115
8.3	Aida Semantics	117
8.4	Proof	119
8.5	Summary	119
9	Language Extension — Parameterless Procedures	120
9.1	Tosca Semantics	120
9.1.1	Declaration Before Use Semantics	123
9.1.2	Type Checking Semantics	124
9.1.3	Dynamic Semantics	125
9.2	Compiler	127
9.3	Aida Semantics	129
9.4	Proof	130
9.5	Summary	130
10	Language Extension — Procedure Parameters	131
10.1	Tosca Semantics	132
10.1.1	Declaration Before Use Semantics	134
10.1.2	Type Checking Semantics	135
10.1.3	Dynamic Semantics	137
10.2	Compiler	139
10.3	Aida Semantics	140
10.4	Proof	141
10.5	Summary	141

IV	Discussion and Conclusions	142
11	Discussion	143
11.1	Formal Analysis of Compilers	143
11.2	Tool Support	145
11.2.1	Z in PVS	145
11.2.2	Use of PVS	146
11.2.3	Necessary Tool Features	147
11.3	Comparison with DCC Development	149
11.4	Scalability	149
11.5	Relation to Previous Work	150
11.6	Summary	151
12	Conclusions and Future Work	152
12.1	Mechanical Formal Methods	152
12.2	Scalability and Complexity	153
12.3	Automation	153
12.4	Resilience	154
12.5	Mechanizing a Hand Development	154
12.6	PVS for Z	154
12.7	Suggestions for Future Work	154
12.7.1	Compiler Verification	154
12.7.2	Semantic Descriptions	155
A	A worked example of PVS	171
A.1	Specification of GCD	171
A.2	Challenging the Specification	172
A.3	Correctness Properties	175
A.3.1	GCD is divisible	175
A.3.2	GCD is greatest	186
A.3.3	Divides Lemma	192

B	Functional Relations Specification	197
C	Z types, functions and domains in the DCC method	199
C.1	Tosca — States and Environments	200
C.2	Tosca — Semantics	201
C.3	Aida — The Target Language	203
C.4	Operational Semantics	204

List of Figures

2.1	Stages in a typical compiler	10
3.1	Outline of the Sizewell B source/object-code comparison process.	20
3.2	Commuting Diagram	25
5.1	Pictorial Representation of the DCC Compiler Proof	54
7.1	Structure of Assignment Lemma Proof	101
7.2	Structure of Assignment Lemma Proof Using Strategies	104

Acknowledgments

I am most grateful to my supervisor, Professor Ian Wand, for his helpful advice and suggestions. Also to my assessor, Jeremy Jacob, for providing a different perspective and sounding board. Thanks also to Professor Susan Stepney for her help and encouragement, and to my external examiner, Brian Wichmann, for an interesting examination and useful comments.

For help with verification in PVS, my thanks to John Rushby, Sam Owre, Natarajan Shankar and all at the Computer Science Laboratory, SRI International.

For creating a friendly working environment, thanks to Darren Buttle, Ken Chan, Tina Conkar, Simon Fowler, Bill Freeman, Julia Hill, Fiona Polack, Tony Powell, Divya Prasad, Darren Priddin, Michael Thyer, and Christine Wykes. Also, thanks to Margaret Ferguson (Language Centre, York) for her useful advice.

For things too numerous to mention, thanks to the secretaries and software experimental officers of the Department of Computer Science at the University of York.

For useful comments on earlier drafts of this thesis I thank John McDermid (York), John Rushby (SRI), Susan Stepney (Logica), and Bill Young (EDS). Also thanks to Sree Rajan and Axel Dold for their helpful explanations of their work.

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) and Siemens Plessey Systems Ltd. under CASE studentship number 94315332.

Declaration

I declare that (except where explicitly stated) the work contained herein is my own original contribution, and previously unpublished with the following exceptions:

- The chapter “From Z to PVS” is based on a paper presented at FME ’97: “Using PVS to prove a Z refinement: A Case Study”[1] which I co-authored with Professors Susan Stepney and Ian Wand.
- The description of PVS given in the chapter “Tool Support”, is based on that given in SRI technical report CSL-95-10 “A less elementary tutorial for the PVS specification and verification system”[2] which I co-authored with John Rushby.
- The chapter “The Starting Point” draws upon on a summary of Stepney’s book[3], written by her for the above FME paper.

This work is based in part on previous work by Susan Stepney, David Whitley, David Cooper and Colin Grant from Logica UK Ltd., and this thesis contains fragments of specifications and proofs previously published in Susan Stepney’s book “High Integrity Compilation — A Case Study”[3]. These are clearly marked as being from Stepney, and are reproduced here with permission of the author.

Chapter 1

Introduction

Computer systems are increasingly being used in applications where failure could lead to loss of life, financial or environmental disaster. Such systems are called *critical* and must be engineered to the highest quality to anticipate potential faults and to reduce the possibility of erroneous and unexpected behaviour.

The major enemy in seeking assurance in a computer system is its complexity. Complex systems are difficult to specify and reason about, hence mistakes and omissions *will* be made during the design process. Resigning ourselves to the fact that we cannot replace the human designer with some infallible machine (who would design it?) we try to minimize the possibility of these errors going unchecked.

Checking for these errors is undertaken as part of a constant verification and validation process during the system development. We verify that the system is being built according to its requirements specification and validate those requirements to ensure that the system specified is what the customer intended. The requirements specification is successively refined either formally (the usual meaning of refinement) or informally, to what is likely to be a very large piece of software, coded in some high-level computer language, such as Ada or C.

Assuming we have, at great expense, formally refined our customers requirements to an Ada program we now have a dilemma. As noted, Ada is a *high-level* language, indicating that it contains constructs intended to make the programming task easier and less error prone. The payment for this ease is made by the need to use a compiler[4, 5] or interpreter (another very large

piece of software) in order for the program we have written to be executed on a real computer.

Thus the compiler (and associated tools including an assembler, linker and some libraries) is in a position of great trust in the building of our software system, and the potential problems of both unintentional (bugs) and intentional deviations[6] from *correct* compilation (i.e. that which preserves functional meaning) can cause harmful effects. Testing mitigates these problems to an extent, as it is the executable code produced by the compiler that is tested and not the source code. Thus compiler introduced errors may be caught by good testing procedures.

Standards for the development of critical systems[7, 8] demand that compilers shall have been developed to the same degree of confidence as the application. Alternatively, you must show *by formal arguments* that the object code the compiler has produced is equivalent to the source code you provided. Thus a distinction is drawn between the development of a trusted compiler and placing trust in a particular compilation.

Having argued that writing programs in a high-level language opens up all of the problems of untrustworthy compilers, then why should we use high-level languages? Writing directly in assembler we can remove the need for a compiler. However, assemblers are not simple pieces of software either, although formal verification of assemblers has been shown to be within the state of the art and has been performed, for example, for a simple assembler for the Viper microprocessor[9, 10, 11, 12, 13] and for an assembler as part of the CLI stack[14].

Also, manual production of low-level code is notoriously error prone, as effectively we are replacing a mechanical compiler with an unverified human being, and thereby removing all of the advantages of productivity and ease of understanding that are brought about with a high-level language.

In this thesis, we address the problem of trusted compilation. We assume that the benefits of high-level languages are a positive asset to the development of a critical system, and we provide a *mechanical* approach to the development of trusted compilers, building upon an existing hand-crafted approach due to Stepney et al[15, 3].

1.1 Compiler Verification

There are four essential ingredients in any verification of a compiler:

1. a semantics of the source language;
2. a semantics of the target language;
3. a specification of the compiler itself; and
4. a proof that the compiler is meaning preserving, with respect to the semantics of the source and target languages.

From a scientific viewpoint, there is nothing in the above which is beyond the state of the art or challenging. We know how to give consistent, unambiguous specifications to high-level languages; we know how to specify (and verify) microprocessors; we can specify compiler transformations and verify them with respect to the semantic descriptions of the source and target languages.

So, why isn't every compiler verified? The answer to this relates to the sheer scale of modern programming languages and their compilers. We do not yet have the necessary *engineering* ability to make compiler verification a feasible prospect for languages of industrial scale.

Take as an example the Ada 83 programming language. This has a language reference manual[16] which is 340 pages of natural language description, and between its publication and the subsequent updating of the language to Ada 95, 829 'issues' regarding clarification of the semantics were raised[17]. A formal semantics of the SPARK¹ subset[18] of the language has been presented in Z, covering some 525 pages of specifications[19, 20]. It has not been machine checked, and is therefore likely to contain inconsistencies².

If we cannot provide a formal semantics for our source language, then we have nothing to verify with respect to, so we fail at the first hurdle. This is not necessarily because it is technically difficult to do each part in isolation, but the size of the language gives a complexity problem which is difficult to effectively control. A report by Forsyth et al[21] on the use of Ada 83 in safety critical applications presented the view that trusted compilers for languages like Ada are never likely to appear due to their semantic complexity.

¹Southampton Portable Ada Runtime Kernel.

²It is an incomplete semantics as well, as it makes no attempt to define real (fixed and floating) point arithmetic.

1.2 Thesis

Our thesis is that mechanical verification can provide effective, scalable, support to compiler verification, at a similar cost to hand proofs of correctness.

The objectives for the work reported here are:

- to determine the extent to which tool support is an essential ingredient of compiler correctness proofs and to estimate the value of mechanical proofs over hand proofs of compiler correctness;
- to determine which parts of the mechanical proofs may be automated;
- to examine the resilience of the mechanical proofs to augmentation of the source language, and how the specification of the semantics may better be structured for this; and
- to examine the relationship between the complexity of the source language and the proof work required, and if there are any limits of the scale of language and target machine that may be handled by this approach.

The chosen research method resulted in two subsidiary aims:

- to explore some of the issues in translating a by-hand proof development into a mechanical theorem prover; and
- to determine the utility of PVS as a proof tool for conjectures cast in Z .

1.3 Map of the thesis

This thesis is organized into four main parts:

Part I **Background Material and Review**

- Chapter two describes the structure and organization of a typical compiler, and compares the different methods of providing formal representations of programming language meaning.

- Chapter three presents a review of previous work in the area of trusted compilation.
- Chapter four describes the issues involved in tool support for formal methods, and for Z in particular. It also provides an overview of the PVS specification and verification system.

Part II **Mechanizing the DCC Method**

- Chapter five describes the DCC method for the construction of demonstrably correct compilers, the starting point for our work here.
- Chapter six presents the key steps in the translation of the DCC specifications from Z into PVS.
- Chapter seven describes the mechanization of the DCC compiler proofs.

Part III **Extending Tosca**

- Chapter eight describes the rationale for extending the DCC source language (Tosca), and then presents the first extension — local scope.
- Chapter nine describes the extension with parameterless procedures.
- Chapter ten describes the extension with procedure parameters.

Part IV **Discussion and Conclusions**

- Chapter eleven presents a discussion of some of the key aspects of our work, and provides comparisons with other approaches.
- Chapter twelve presents the conclusions of the thesis and suggestions for future work.

Appendices provide a worked example of specification and verification in PVS; the definition of functions as relations in PVS; and a key to the Z types, domains and functions in the DCC method.

Part I

**Background Material and
Review**

Chapter 2

Compilation

Conceptually, a typical compiler[4, 5] operates in several phases, as shown in Figure 2.1. The lexical analysis (or scanning) stage involves converting a raw stream of characters¹ into a stream of *tokens*, which are sequences of characters which have a collective meaning, for example a keyword or an identifier.

Syntax Analysis (parsing) then builds these tokens into a parse tree. Parse trees are hierarchical groupings of tokens which (again) have a collective meaning — usually those of the grammatical phrases in the source program. Lexing and parsing are very well understood areas of compilation, and tools are available to generate lexers and parsers of an efficiency comparable to those which can be written by hand[23, 24].

Semantic analysis involves checking the static semantic consistency of the program, with respect to the language definition. This can involve type checking (ensuring operators are applied to appropriate operands), flow of control checks (for example checking that a **break** in C has an enclosing **while**, **for** or **switch** statement), uniqueness checks (for example disjointness of options on a **case** statement in Pascal) and name related checks (for example the identifier tags on **begin** and **end** statements must match in Ada).

After semantic analysis, some compilers use an explicit intermediate representation for the program. This intermediate language (IL) can be considered as code for an abstract machine, for example a simple stack machine

¹Some programming languages use a more elaborate representation than plain text. For example the IEC 1131-3 standard[22] for PLC programming languages defines several graphical programming languages in addition to a structured text language.

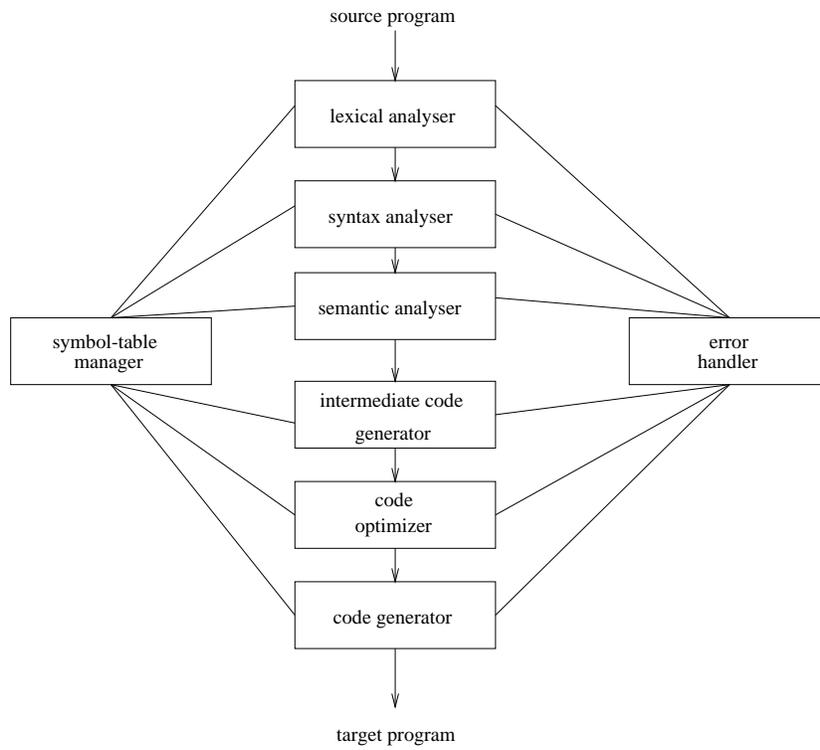


Figure 2.1: Stages in a typical compiler

or a three address code. Such ILs can allow the language specific compiler front-end to be attached to optimizers and code generators for various target architectures, thereby allowing a degree of reuse of compiler components.

Optimizations are performed on this intermediate code (and often on the resultant target code as well) in an attempt to improve efficiency. Optimization is commonly the most expensive phase of compilation, and the most unreliable[25, 26]². Standards for the construction of safety critical software often prohibit the use of optimizers as they can cloud the traceability between source code and object code. However, optimization is sometimes essential, for example in a real-time system with limited resources. In this work, we assume this is not the case, and that traceability is the key issue.

The code generator usually produces assembly code or relocatable machine code for the target machine. It involves the selection of target machine instructions, allocation of variables to memory locations and allocation of registers, if any. This phase of compilation is much simplified for what are known as RISC machines (Reduced Instruction Set Computers), where there are usually only a few, simple instructions to choose between and a very large set of registers to use.

The compilation process is supported by a symbol table manager, which is used to record the identifiers used in the program and to collect information about the various attributes of the identifiers, such as their type, scope, storage allocation, and so on. The symbol table data structure is built up by the lexical analyzer as identifiers are detected in the source text. This data structure is then ‘decorated’ by the semantic analyser, by which we mean the attributes of the identifiers are assigned.

All parts of the compiler can find errors in the source program. Error detection must be performed in such a way that errors are reported to the user in the context of the initial source program rather than the compiler’s current internal representation, so that the offending construct can be more easily located. In certain cases it is possible for the compiler to recover from an error in order to check the rest of the program text (but not generate any object code) by, for example, skipping ahead in the source text or assuming an implicit declaration.

²Production compilers often produce less errors in object code when optimization is turned on. This apparent contradiction to our assertion that optimization is unsafe is due to the compiler being more heavily used with optimization on, giving much better testing coverage.

2.0.1 Bootstrapping

Compilers are themselves just programs, and are usually written in a high-level language. Therefore, even compilers will have at some stage been compiled, but by what? This apparent chicken-and-egg problem is known as bootstrapping.

The problem of bootstrapping is of paramount importance in trusted compilation. If we have produced a verified implementation of our compiler, how do we compile this program into a trusted executable? We would appear to require another trusted compiler in order to do this, but several interesting methods have been devised to overcome this and are reported in the next chapter, in the discussion of related work.

2.1 Semantics

For any discussion of the correctness of compilation, we require a method for assigning a precise meaning to any particular source or target program. This section presents a brief overview of the various styles of formal semantics available and their advantages and disadvantages for performing reasoning about the congruence of two semantic definitions.

2.1.1 Operational Semantics

Operational semantics describes the effect of programs on the operations of some abstract machine. It is therefore very concrete — a specification of how the computation is to be performed, rather than just what computation is to be done. Due to this, operational definitions tend not to be very modifiable or extensible and can be rather verbose and repetitive.

Correctness proofs with respect to an operational semantics are performed by structural induction over the source language constructs and then verifying that an implementation preserves the execution sequence of the abstract machine. As operational semantics is so concrete it can be difficult to compare even correct specifications which have followed a different method of computing the same result.

2.1.2 Denotational Semantics

Denotational semantics was developed in the early 1970s by Strachey and Scott[27]. It is based on a mathematical abstraction of the computations involved and builds on a well understood underlying theory (fixed-point theory).

The meaning of each program phrase (and the whole program) is represented by a suitable mathematical entity, called the *denotation* of the phrase. The semantics of a language is therefore given by a set of *semantic functions* which map phrases of the programming language to their denotations. Denotations are often themselves functions.

Due to its abstract nature, it can be difficult to relate particular entities to familiar programming concepts in imperative and concurrent languages, as their operational implications may not be readily understood. The notation is also esoteric, but may be improved by some ‘syntactic sugar’ or auxiliary notation such as that presented in Watt’s book[28].

As denotational definitions are compositional, correctness proofs of translation mechanisms can use structural induction on the constructs of the language. Proofs of individual phrases follow by showing (by extensionality) that the implementation produces a function equivalent to its denotation.

2.1.3 Axiomatic Semantics

Axiomatic semantics are similar to operational semantics in that they deal with state. Instead of relating states, axiomatic semantics relates assertions about those states.

It is helpful for reasoning about the correctness of individual programs (cf. Hoare Calculus[29, 30]) but it can be difficult to infer the operational behaviour of a specification. It is also very easy to introduce inconsistencies in the axioms — and anything is provable from a contradiction in the axiom set.

2.1.4 Action Semantics

Action semantics[31] was developed by Mosses to allow formal semantics to be more accessible to non-formalists. It is based around the notion of actions,

which relate directly to familiar operational concepts in programming languages. For example, there are primitive actions for storing values, binding values to identifiers, sequencing, iteration and so on.

It differs from operational semantics in that concepts such as control flow, storage and bindings are specified directly rather than through the operations of some abstract machine.

Action semantics is modular, and each action operates in one (or more) of four facets, which are independent — functional, declarative, imperative and communicative. The *functional* facet deals with transient values: a functional action is given and produces data. The *declarative* facet deals with scope: a declarative action is given and produces bindings. The *imperative* facet concerns stable information: allocation and deallocation of cells, storage and retrieval of data. The *communicative* facet deals with permanent information: communicative actions send and receive messages, and contract work to agents.

The advantage of this modularity and separability is that action semantic specifications are very easily modified and extended. The formal basis for action semantics lies in unified algebras[32].

2.1.5 Problems with Semantic Definitions

The difficulties of providing formal semantics for real-world programming languages are well documented[28]. Most of the issues here are due to cases where formal semantics have been devised for an extant, informally defined language. The formal specification is therefore cluttered with special cases in order to cope with inconsistency, lack of orthogonality and incompleteness in the original informal descriptions.

With increasing complexity of programming languages it would be desirable if they were developed in conjunction with a fully formal semantics so that this problem may be averted. This post-formalism problem is not specific to this domain — there is much evidence[33] to support the claim that the most productive and cost effective time to apply any formal approach is as early as possible in the project life-cycle.

The semantics of low-level languages (assemblers) are just as problematical as those for high-level languages. In many cases the published descriptions

of processors are incomplete and incorrect³, so any attempt to reason about the operation of their assembly languages (and thereby the target processor) is meaningless. In the next chapter (Section 3.2.1) we report the work of Yu, who noted such problems with the Motorola 68020.

2.2 Summary

This chapter has presented an overview of the structure of a typical compiler, and described some of the methods of providing formal semantics to programming languages. We also noted the difficulties involved with formal semantic definitions, with particular reference to post-formalisation of existing programming languages whose semantics are only defined by natural language (or worse, defined by the compiler).

³The semantics of assembly languages are a subset (public interface) of the semantics of the target processor.

Chapter 3

Trusted Compilation

Programmers have been concerned about the correctness of their programs since the invention of programmable systems. Babbage described the verification of the formulae used on the operation cards of the analytical engine in the 19th century[34], and Turing provided a proof of program correctness in 1949[35].

Concern for the correctness of compilers, themselves just programs, is reported in the literature from the 1960s. This chapter provides a brief overview of the three main approaches to trusted compilation that we have identified in the literature and then gives more detail on the particular approach upon which this thesis is based — analytical compiler verification.

3.1 Overview

The literature on trusted compilation fits neatly into three distinct types:

1. verification of object code;
2. compiler synthesis; and
3. analytical compiler verification.

The latter is what we would think of as a traditional proof of program correctness, where the compiler is proven correct with respect to the formal semantics of the source and target languages. That is to say, the compiler

provides a *meaning preserving* transformation of the source code into object code. It is this approach which is discussed in detail in this chapter. The first two sections give a brief discussion of the other areas to show how they contrast with the analytical approach and their relative strengths and weaknesses in attaining the goal of trusted compilation.

3.2 Object Code Verification

We noted in the introduction that one effective method of bypassing the problem of trusting a compiler is instead to write directly in assembler. We can then perform a verification that the *object code* satisfies the original requirements specification in much the same way as testing does, but allowing us to do something that testing can never allow (except on trivial examples), i.e. the opportunity to explore *every* behaviour of the target system.

3.2.1 Yu's M68020 Verifier

The most complete and impressive work in the area of object code verification is that by Yu. In his thesis [36] and subsequent technical report[37] Yu describes a system to reason about the correctness of arbitrary Motorola 68020 machine code. He describes the building of a formal definitional model of most of the user level of the M68020 in NQTHM[38], the logic of the Boyer-Moore theorem prover. The model is an operational semantics in the strictest sense, providing an interpreter for M68020 code (albeit a very slow one).

A lemma library was developed to prove that:

1. the M68020 code¹ correctly implements a given algorithm, specified in NQTHM; and
2. that the NQTHM algorithm satisfies a given requirements specification.

The first part of the proof obligation is specified as:

$$\text{p-statep}(s) \Rightarrow \text{p-req}(s, \text{step-n}(s, \text{p-t}(s)))$$

¹The NQTHM 'interpreter' works directly from the relocatable binary machine code.

That is, if the precondition $p\text{-statep}$ is satisfied, the properties specified by the relation $p\text{-req}$ will hold. $p\text{-req}$ relates the initial state s and the resulting state $\text{step-n}(s, p\text{-t}(s))$, where step-n provides the execution semantics of $p\text{-t}(s)$ instructions from the state s .

The precondition $p\text{-statep}$ asserts that: the machine is in user mode; the program counter is word aligned; the program to be executed is stored in memory starting at the current program counter value; there is sufficient memory space available for the computation; and any necessary preconditions specific to the program. The relation $p\text{-req}$ asserts some properties about the resultant state of the machine and the register file, that values have been stored correctly and that no exceptions (e.g. illegal memory access) occurred.

The second half of the proof obligation is performed by providing putative theorems about the NQTHM specification of the algorithm. For example, in the case of an algorithm to compute the greatest common divisor (GCD) of two non-negative integers a and b you could use the following theorems:

Theorem : gcd-is-cd

$$((a \bmod \text{gcd}(a, b)) = 0) \wedge ((b \bmod \text{gcd}(a, b)) = 0)$$

Theorem : gcd-the-greatest

$$\begin{aligned} &((a \neq 0) \wedge (b \neq 0) \wedge ((a \bmod x) = 0) \wedge ((b \bmod x) = 0)) \\ &\Rightarrow (\text{gcd}(a, b) \not\leq x) \end{aligned}$$

Yu makes several simplifying assumptions in his model, for example that the instruction space is read only (hence write outs of the instruction cache can be ignored).

The NQTHM specification of the M68020 is 80 pages long, and contains 569 function definitions. Yu claims this complexity is due to the CISC nature of the M68020, and not to his choice of specification vehicle. Even this magnitude of specification does not provide a complete description of the M68020 — Motorola engineers were asked to clarify the evaluation order of multiple effective addresses in relevant instructions. Despite this co-operation with Motorola there were several user level instructions omitted from the formal model as they could not be precisely defined from the Motorola literature.

Since moving to DEC SRC, Yu has built another object code verification system for the DEC Alpha processor². The interpreter model was about

²Noted by Yu, in private communication, 1994.

the same complexity as the M68020 model (approximately 100 pages). He performed the verification of about 500 lines of Alpha code using the system³.

If we have a general purpose environment for the verification of arbitrary object code, we can gain confidence in the output of a non-trusted compiler by applying such verification techniques to the object code it generates. Thus, we bypass the need to verify a compiler, and instead verify those particular *compilations* in which we require trust. Yu's method, described above, has been successfully applied to the object code generated from C, Ada, and AKCL Lisp compilers for the M68020 based Sun-3.

3.2.2 A Lisp Compiler

Other work on verification of compiler generated object code has taken a slightly different approach to the problem. An early example is the work of Samet[39, 40] who developed a system for proving that programs written in a high level language (Lisp) are correctly compiled and optimized for Lap (a PDP-10 assembly language). The approach involves translating both the high and low level representations of the program into an intermediate form and demonstrating their equivalence.

Equivalence is defined (to avoid halting problem like arguments about equivalence of programs) as meaning “*that the two programs must be capable of being proved to be structurally equivalent, that is, they have identical execution sequences (e.g. they must test the same conditions) except for certain valid rearrangements of computations*”. The equivalence proof does not use any form of theorem proving, as the equivalence of the (restricted) normal form of Lap code is decidable.

3.2.3 Sizewell PPS

A similar approach to Samet has been used more recently in the verification of the Primary Protection System (PPS) of the Sizewell B nuclear power station[41, 42, 43], for programs written in PL/M-86 targeted at the Intel i8086 processor.

A large amount of effort was spent demonstrating that the PL/M source code correctly implemented the requirements specification. To continue this

³Scale of work noted by Boyer, in private communication, 1998.

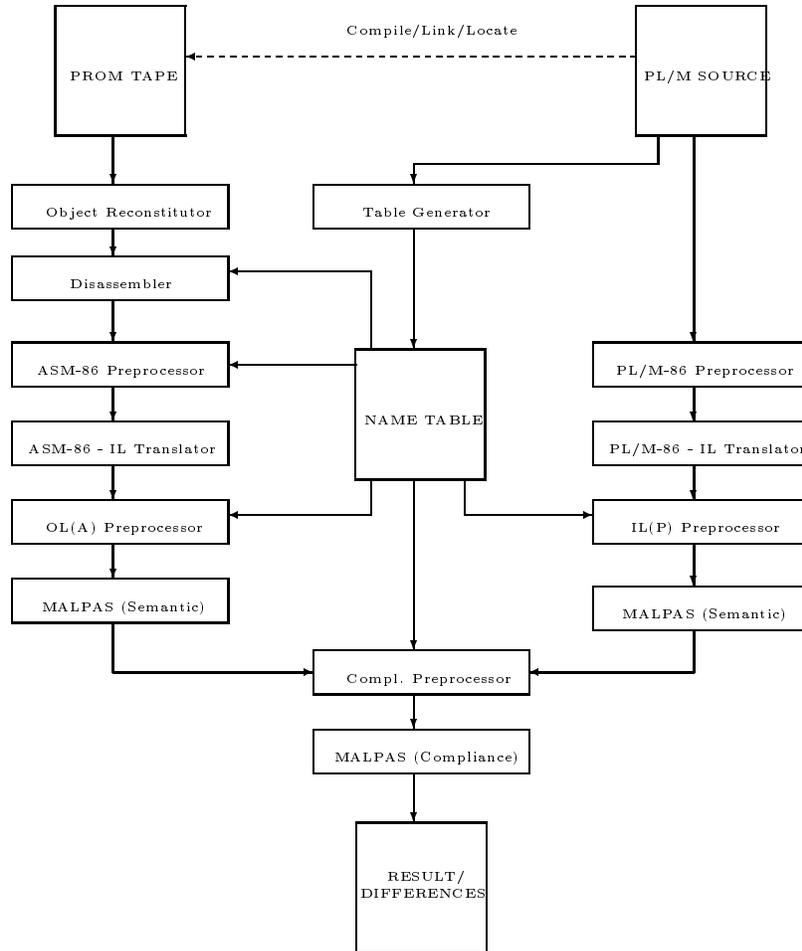


Figure 3.1: Outline of the Sizewell B source/object-code comparison process.

rigour to the level of delivered object code in PROM a decompilation approach was introduced[41].

The approach is outlined in Figure 3.1⁴. The PL/M source and the object code from the PROM are both translated via a number of steps into the intermediate language (MALPAS-IL) of the MALPAS static analysis tool[44]. The MALPAS-IL representations are then submitted to a preprocessor which marks various points (nodes) in the code such as loop heads and procedure

⁴This diagram is due to Pavey and Winsborrow, from their paper[41].

calls (some manual intervention was necessary to perform this task). These nodes identify matching points in the PL/M source and the object code from the PROM.

For each path between marked nodes the MALPAS semantic analyzer produces expressions for output variables in terms of input values. These expressions are then passed to the compliance analyzer which generates *threat* expressions. These represent conditions under which the PROM code may not conform to the PL/M code.

It is the *simplicity* of PL/M-86 which makes this approach feasible. The compiler performed few optimizations, which in a more complex compiler could make the matching of source to object code constructs very difficult. Memory allocation is performed in a static manner which simplified the recognition of variable references.

This technique was applied to over 85000 lines of PL/M-86 and the corresponding PROM images. A few modules passed through automatically, the rest requiring human intervention to match nodes, and in one case the entire compliance analysis was done by hand. Pavey and Winsborrow's original paper[41] does not mention the number of bugs located by this method, although a leaked internal report[45] from the Nuclear Installations Inspectorate (NII) notes that 11 problems were located, one of which was described as a 'significant' compiler error. Later papers[42, 43] note two real errors in pointer comparison and register assignment.

3.2.4 STERNOL

A similar approach has been adopted by ABB Signals (Sweden) for the checking of a compiler for the language STERNOL used in railway signaling interlocking systems. They verified compilations by using a reverse compiler, and then checking the equivalence of the two source programs using Prover[46]. This is possible due to STERNOL programs being equivalent to propositional logic, for which equivalence is decidable.

3.2.5 Summary

Pavey and Winsborrow demonstrated that object code verification is applicable to industrial size examples in a timely manner. However, their approach

was only feasible because of the simplicity of PL/M-86 and is unlikely to scale to more complex programming languages without an exponential increase in effort.

Ongoing research in this area includes that of Buttle[47] who is investigating the correct compilation of SPARK Ada into M68020 code. His method is independent of the compiler, and does not use translation to an intermediate language. Verification is performed by the propagation of SPARK proof annotations into the object code, where they are discharged. Initial results are promising as although SPARK Ada is complex (although not as complex as full Ada), the translations used in the compiler are quite straightforward (no optimization performed) so matching execution paths is simple.

3.3 Compiler Generators

One of the advantages of performing a proof of a compiler over the proof of individual compilations is that it is a ‘one-off’ event and does not require us to expend verification effort for each execution of the compiler. Taking this argument one stage further, why perform verifications of compilers for each source and target language combination we require, when we could instead verify a compiler generator — a ‘compiler compiler’?

To verify a compiler we require a formal semantics of the source and target languages and a formal description of a translation mechanism between them. We then prove the translation to be meaning preserving with respect to the semantics. With a verified compiler generator we use the formal semantics as input ‘programs’ and generate a trusted compiler in the same manner as a verified compiler produces trusted object code.

3.3.1 Early Work

Early work in semantics directed compiler generation[48, 49, 50, 51, 52, 53] concentrated on the use of denotational semantics. These systems produced compilers whose object code ran at least three orders of magnitude slower than that produced by handwritten compilers and only the system of Gormard and Jones[51] was proven correct. This system, however, produced compilers which generated code for the untyped λ -calculus, leaving considerable (unproven) compilation still to be done before it could be executed on a realistic processor.

3.3.2 MESS

A more realistic compiler generator is the MESS system, due to Pleban and Lee[54, 55], which is capable of generating compilers for realistically sized imperative programming languages targeted at standard architectures. Languages are specified in ‘high-level semantics’[56]. They claim the core of their generator is verifiable with respect to the semantics of the input language, although they did not perform such a proof.

The example compiler presented in their paper[54] is for a language called Sol/C⁵. Sol/C contains two-level binding, recursive procedures (with value and reference parameters), multidimensional arrays, simple input/output and the usual control constructs. The language specification contains approximately 5400 well commented lines.

3.3.3 CANTOR — A Verified Compiler Generator

Palsberg’s compiler generator Cantor[57, 58, 59] is a verified compiler generator that has been used to generate compilers for a realistic imperative programming language (a subset of Ada) for real target machines (SPARC and HP Precision). However, during the development of Cantor the principle that correctness is more important than efficiency was used and hence the resultant compilers generate object code that runs at least two orders of magnitude slower than that from handwritten production compilers. A major gain over some other compiler generators is that Cantor generated compilers perform type checking at compile time.

Cantor takes specifications in a subset of Action Semantics and produces compilers written in Scheme targeted at an abstract RISC machine PseudoSPARC. The abstract target machine is a model of the SPARC architecture[60] with two idealizations:

- **Unbounded word and memory size:** data values are unbounded integers. Program and memory sizes, number of registers in a register window and number of register windows are unbounded.
- **Read-only code:** code cannot be overwritten and data cannot be executed.

⁵Sort of like C.

Code for the PseudoSPARC is then assembled into code for the full SPARC or HP Precision[61] architectures.

The specification of the mini-Ada compiler takes 56 pages, plus 13 pages detailing the semantics of action notation. The correctness statement, including various lemmata, takes 28 pages (not including the proofs).

The generator proof is produced by hand and Palsberg suggests as future work that the proofs be machine checked but has not done so to date.

3.3.4 Summary

Compiler generators are an attractive prospect, but the emergence of a verifiable and efficient compiler generator still seems a long way off. The efficiency of automatically generated compilers is beginning to converge with that available from hand written compilers. However, the best available from a verified compiler generator is two orders of magnitude slower.

3.4 Analytical Compiler Verification

Having described the two major competing approaches we now concentrate on presenting previous work in the area of analytical compiler verification, by which we mean verifications that have been performed of particular compilers.

In such a verification we are trying to demonstrate that the object code output from the compiler is a correct refinement of the source code input, with respect to the semantics of the two languages. Hence we require a formal semantics of the source and target languages, a formal specification of the compiler and a proof that the compiler specification provides a meaning preserving transformation with respect to the semantics. This is represented pictorially in Figure 3.2, generally known as a commuting diagram.

Much of the early work in analytical compiler verification[62, 63, 64, 65, 66, 67, 68] appeals to algebraic theory to help prove that a diagram such as that given in Figure 3.2 commutes, i.e. that the compiler provides a semantic preserving translation function (actually a homomorphism) from source to target language. This work was merely a paper exercise, and was more concerned with developing the algebraic theory than with tackling practical compilers.

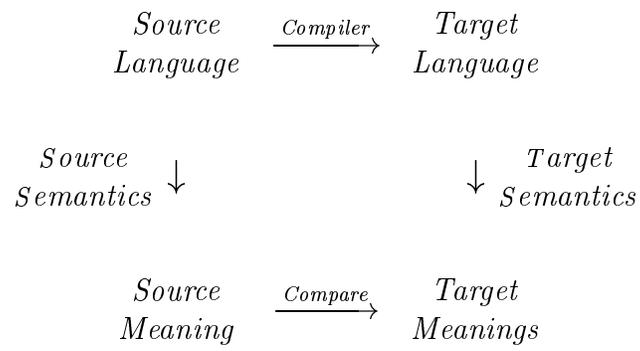


Figure 3.2: Commuting Diagram

3.4.1 Verified Pascal

The first realistic contribution to analytical compiler verification was by Polak in 1981. In his thesis[69], Polak describes the mechanical verification (using the Stanford Pascal Verifier[70]) of the partial correctness of a non-optimizing compiler for a substantial subset of Pascal, targeted at a fictional stack machine similar to the B6700. The source and target languages are defined denotationally and verification conditions are defined by means of pre and post conditions in the (quantifier free) assertion language of the Stanford verifier.

The verification work is large — the compiler specification is about 70 pages, the source semantics 30 pages, the target semantics 4 pages and the proof itself takes 50 pages. It has been reported by Young[71] that there is a large collection of unproven assumptions in the formal theory and that Boyer has found several inconsistencies in Polak’s axioms.

3.4.2 CLI Verified Stack

Following Polak’s work, Computational Logic Inc. (CLI) undertook a compiler verification[72] as part of their work on a trusted stack of system components. This stack approach[73, 74] to systems verification is meant to prove the entire chain from the correctness of the source code to the correct execution of the microprocessor at register transfer level.

All of the stack verification at CLI was performed using the Boyer-Moore theorem prover[38] over definitions of finite state machines (FSMs). These FSMs are known as interpreter functions and the proof obligations are such that the lower level of the stack (the concrete interpreter function) is proved equivalent to the higher level (the abstract interpreter function).

The compiler used in the CLI stack is for a subset of their language Micro-Gypsy[75], which contains integers, booleans and characters, one dimensional arrays, if and loop control, signal handlers, recursive procedures, expressions and assignment. The target is a high-level assembly language, Piton, for the FM8502 processor. It is high-level in the sense that it has typed data and recursive procedures. The verification of the Piton assembler is described in [76].

Both Micro-Gypsy and Piton are defined operationally (via the FSMs referred to earlier) expressed in NQTHM, the logic of the Boyer-Moore prover.

The proof is performed by structural induction over source language constructs. The specification of micro-Gypsy was 850 lines, Piton 1100 lines and the translator function a further 1000 lines. The verification of the translator took approximately 10 person-months.

3.4.3 ProCoS

The European response to the CLI work was to establish an ESPRIT basic research project — ProCoS (Provably Correct Systems)[77]. This ongoing work started by investigating the proof of a compiler for a sequential subset of Occam, targeted at an approximation of the Transputer. A novel feature of the Oxford part of the ProCoS work[78, 79, 80] is the use of an interpreter *written in the source language* to define the semantics of the target machine. Thus, the compiler proof is reduced to showing a refinement relation between the source code and the interpreter for the generated target code. ProCoS investigated the use of mechanical theorem proving (Boyer-Moore) in their initial work with little success[81, 82].

3.4.4 Verifix

An ongoing compiler verification project at the Universities of Karlsruhe, Kiel and Ulm (known as Verifix) consists of several different approaches. The group at Karlsruhe is working on the correct compilation of an imperative C like language (IS) to DEC Alpha code[83]. The language is specified operationally using Montage, a new approach based on Gurevich abstract state machines. The IS language is first compiled into a basic block intermediate language known as MIS. MIS is then compiled into DEC Alpha code by specifying a set of term-rewriting rules and using a back end generator.

At Kiel, a group is working on the development of a correct compiler for a subset of common Lisp[84]. Their compiler first translates the Lisp code into an imperative language C_Int, similar to the language IS used at Karlsruhe. C_Int is then translated into code for the Transputer.

The work at Ulm is more general and concentrates on the formalizing, structuring and automation of proofs of compiler correctness theorems[85, 25, 26]. Their work is based on the assumption that compilation of standard programming constructs (and therefore the corresponding correctness proofs)

usually follow a scheme that differs only in small details between languages. They therefore concentrate on the essence of compilation steps and abstract from detail, creating generic theories for correct compilation of various constructs. They have developed a hierarchy of generic compilation patterns for elementary constructs within the PVS specification and verification system. A comprehensive treatment of various styles of semantics has been developed within PVS to support the Ulm work, including the appropriate formal mathematical basis (e.g. fixed-point theory)[85].

3.4.5 Verified Scheme

Based on the Clinger-Rees denotational semantics of Scheme[86] and the Scheme48 compiler[87] a group at Mitre and Northeastern University have produced a verified compiler for Scheme, called VLISP[88, 89, 90]. The VLISP implementation consists of three elements:

1. a simple compiler from Scheme to an intermediate level byte code language;
2. an interpreter for this byte code language written in a subset of Scheme called PreScheme; and
3. a compiler for PreScheme to the assembly language of the target workstation.

The verification is performed rigorously, but not fully formally, and concentrates on the algorithms and data structures, not the implementation. The proofs are structured following the structure of the Scheme48 compiler and uses four major techniques:

1. semantics preserving source-to-source transformations;
2. structural inductions using the denotational semantics, following Wand and Clinger[91, 92];
3. verification of representations and refinements using operational semantics and the storage layout relations[93]; and
4. soundness proofs of the gap between the denotational and operational semantics.

The specifications and proofs total about 600 pages of technical reports.

The implementation is about a factor of 5 slower than a similar system in C, compiled with the optimizing GNU gcc compiler.

A further compiler for an extension/revision of pure PreScheme (ν -Lisp) is described in Oliva's thesis[94]. This compiler is implemented (in ML) using the same techniques as the earlier PreScheme compiler, using denotational reasoning for the front end and operational reasoning for the back end. Instead of defining a *particular* back end, a set of criteria is developed relating code for an intermediate stack machine to code for the target register machine. These criteria are parameterized by register assumptions which formalize how the stack is cached in registers. The criteria therefore characterize a class of back ends, so that any back end which can be shown to satisfy these will preserve the semantics.

3.4.6 A Verified Assembler

Curzon describes[9, 10, 11, 12, 13] the verification of an assembler for a subset of the structured assembly language Vista[95], targeted at the Viper microprocessor. Vista's instructions are those of the underlying machine, and the machine's general purpose registers are visible. Non-recursive procedures and while loops are available.

The source semantics are specified in a relational style, to allow for non-determinism, and non-termination. An oracle is also included, to specify I/O operations. The target is specified operationally – by an interpreter. All of these specifications are given within the HOL system.

The 'compilation' is split into three stages. The first stage translates Vista programs to a flat unstructured intermediate language (Visa). Visa has four infinite stores: a constant store, store, a data store and a link store for procedure link addresses. Only the last two of these are writable. The second stage translates Visa into the Viper assembly code Kaa, which has only one store (of finite size). The final stage (which was neither implemented nor verified) is a simple assembler to Viper machine code.

The proof is intended to be reusable to a certain extent, by the use of source and target language schemas rather than concrete languages. The compilation theorems are therefore generic, i.e. applicable to other target

machines within the same family with a minimum of new proof work. Instantiating the proof for a particular processor involves filling in the gaps of the syntax and semantics of the language, and discharging some simple assumptions in the correctness theorem.

The compiler specification was about 10 pages of HOL (at approx. 80 lines/page). The semantics of the source, target and intermediate language total to 27 pages and the correctness definitions total 15 pages. The HOL proofs consist of 130 additional theories, and were performed over a period of 3 years (although not full time).

The reason we report here this work on an assembler, as opposed to a compiler, is its successful use of mechanical verification and also its interesting approach to bootstrapping. For bootstrapping, the HOL specification has been used to mechanically derive an implementation in ML by the method of Rajan[96]. Rajan's method involves walking a tree of HOL terms and producing an ML term tree corresponding to the HOL formula. It is reasonable to have more confidence in this than a hand coded compiler, but it still relies on the correctness of the ML implementation and the translator.

Curzon's compilation specification can also be *executed by proof* within the HOL system, by using the semantic functions as a compiler for themselves — using HOL to 'execute' them by expanding their definitions. This is painfully slow, but could perhaps be used to generate a small compiler for implementing a larger language (similar to the use of PreScheme described in Section 3.4.5).

3.4.7 A Demonstrably Correct Compiler

A group at Logica Cambridge have generated a *demonstrably correct* compiler for a small, but non-trivial, high-level language Tosca⁶. Two versions of the compiler are reported in the literature: the first is targeted at a fictional assembly language Aida⁷[3]; and the second at Vital, an assembly language for the VIPER[97, 98] microprocessor[15]. They give a denotational semantics to both the source and target languages, and define a compiler by giving an operational semantics for the source language in terms of target language constructs.

⁶Totally Okay for Safety Critical Applications.

⁷An Imaginary Denotational Assembler

The verification is performed by hand and the issue of bootstrapping is addressed here by a translation of the specification into Prolog grammar rules (DCTG Clauses) to allow execution of the specification. Again though, this does not provide a fully satisfactory solution as it relies on the correctness of the Prolog implementation.

The specification of their compiler was approximately (there isn't really a well defined notion of 'lines' in a Z specification):

- source language specification — 325 lines;
- target language specification — 50 lines;
- compiler specification — 70 lines;
- hand Proof of meaning preservation — 100 lines.

The work was performed by hand with the denotational specifications expressed within the framework of Z. The Z specifications were checked using the `fuzz`[99] tool, showing type correctness. Human review was also used for validation and also the translation of the semantics into Prolog (allowing them to be executed) provided extra assurance that not only were the specifications correct, but described what was intended.

This method (which we will refer to herein as the DCC method) forms the basis for our work in this thesis and is described further in Chapter 5.

3.5 Summary

The analytical approach to compiler verification is (at present) the most promising for the production of verified compilers for industrial scale programming languages. The survey above shows that it is possible (with some effort) to verify compilers for large languages, targeted at real microprocessors.

The current state of the art in compiler verification is the Verifix project, reported in Section 3.4.4. Their approach (which was developed concurrently with ours) builds from a complete embedding of various semantic frameworks within the logic of PVS. Initial results are encouraging the belief that their work will scale to realistic industrial compilers in a *generic* and reusable manner.

Chapter 4

Tool Support

The work reported in this thesis makes significant use of the Prototype Verification System (PVS) from SRI International. This chapter provides some background information for the reader on mechanical specification and verification systems as a way of introducing why we chose to use PVS. It then provides some details of the PVS specification language and verification environment to aid understanding of the specification and proof fragments appearing in the following chapters.

Throughout this thesis, PVS specifications are given in boxes with any omitted sections marked with an ellipsis. The specifications have been passed through PVS' \LaTeX feature, with a substitution file devised¹ to make the PVS specifications appear more like the traditional notation for denotational semantics.

4.1 Formal Methods

There is a wide spectrum of 'formality' in the development of computer systems. In one sense, every computer system is formal in that it has been eventually specified in some programming language, be it Z80 assembler or Ada. These are structured representations which have an agreed meaning

¹The substitutions are straightforward and mainly involve the use of Greek letters, subscripts and such. The output was edited by hand to insert emphatic brackets (\llbracket , $\langle\rangle$) where appropriate to indicate the phrase denoted by that semantic function.

which is unambiguous (but that meaning may vary between different implementations).

To focus this discussion, we now give a definition of what we mean by ‘formal methods’. There is a wide spectrum of ‘formal methods’ and some include only a subset of those features in this definition. For further discussion of this diversity and its implications on those choosing a formal method for a particular application the reader is referred to Rushby’s survey[33].

A formal method is something which:

- employs concepts and sometimes notation from discrete mathematics;
- allows the presentation of an unambiguous specification of the behaviour of a system;
- provides the necessary logical infrastructure to reason about its specification notation; and
- provides a method for translating abstract specifications to concrete, executable specifications (i.e. programming languages).

This definition embodies the notions of formality, rigour and refinement. Its aim is to bring what is known as traditional software engineering (usually associated with structured methods of developing large scale software systems) from an art form to its necessary status (particularly for critical systems) as true engineering. It is the prospect of being able to *calculate* systems (and their properties) in a repeatable and analytical manner that makes formal methods attractive.

Rushby[33, p 86] notes that development of such formal methods began in three different ways:

1. expressive specification languages with little or no support for reasoning;
2. theorem proving systems for raw logics;
3. *integrated* development of specification language and reasoning support.

The first approach describes ‘methods’ such as Z[100] and VDM[101] which attempt to provide very expressive and attractive specification languages with little interest in the ability to perform reasoning about specifications. The second approach was driven by the state of the art in verification technology and presents a specification language which is limited to those constructs for which reasoning can be effectively mechanized, resulting in systems such as Boyer-Moore[38]. The third approach takes a middle ground, trading off between expressiveness and the ability to automate efficiently reasoning in the specification language/logic.

Proponents of the second and third approaches would argue that efficient *mechanization* is one of the fundamental points of formal methods. Formalization gives us the opportunity to calculate and there is little point in providing opportunity to calculate without providing mechanical support for this. More than this, it is not sufficient to provide mechanization for raw logics — we need mechanization for specification languages which allow us to reason about industrial scale problems of practical interest in a manner in which a human can comprehend and review.

Armed now with a definition of formal methods and some background on the diversity of approaches to tool construction, we now present a brief survey of tool support for verification of Z specifications.

4.2 Verification Support for Z

Having stated that one of our goals in this work is to reduce the burden on those wishing to perform rigorous verification of compilers, we therefore require a high level of efficient and effective automation. As we are starting from a development in Z described in the previous chapter, we now briefly review previous work on providing mechanical verification support for Z and see how it compares with the opportunity for translating the Z specifications and hand proofs into a different system.

This review draws from a survey of Z tools by Imperial Software Technology (IST) written in June 1994[102], shortly before we started the research presented here. There has been a great deal of development in this area since then and a more recent review of verification support for Z has been published[103]. Hence this review should not be taken as a statement of the current state of the art in verification support for Z specifications but as a

justification of the choice of tool support for our particular application from the state of these systems in 1994².

There are two distinct routes towards theorem proving support for the Z specification language — tools designed specifically for Z and general purpose theorem provers which have been ‘tailored’ for Z. The IST report covered both approaches and presented a comparison of the tools available against a set of criteria which they considered important following their own experience in developing Zola[104], a Z support tool.

The basic criteria were as follows.

- How easy is it to write specifications using the tool?
- How easy is it to read specifications using the tool?
- How can one prove properties of specifications using the tool?
- Can one refine specifications to code using the tool?
- Does the tool support the Z mathematical toolkit?
- Does the tool support large, multi-author specifications?
- Is the tool ‘open’ and able to inter-operate with other tools?
- How much does it cost?

The important feature here for us is the ability to reason about the specification, and the support for large developments (as we wish our results to scale to realistic industrial scale languages). Of the five tools in the survey, only four provided any support for verification:

1. **ProofPower**: an embedding of the Z language in HOL[105];
2. **Zola**: a purpose built system[104];
3. **CADiZ**: a purpose built system[106];
4. **Z/HOL**: another embedding of Z in HOL[107, 108].

²For a survey of current tool support for Z, the interested reader is referred to a report by Martin[103].

Examining the systems which have been specifically designed for reasoning about Z specifications, we noted that the proof support in CADiZ was in the early stages of development and neither stable nor mature enough for our purposes. The prover in Zola was more advanced and included a tactic language, facilities for the reuse and modification of proofs and the tracking of assumptions and unproven lemmas/theorems. Libraries of low and medium-level tactics were available; but more general, high-level, tactics were still under development.

In terms of highly automated support for verification, it is unsurprising that the embeddings of the Z language into a mature, general purpose theorem prover will provide more advanced facilities than immature purpose-built developments. This comment has been made by Paulson in his tutorial paper on writing a theorem prover[109], in which he sums up the advice as “Don’t write a theorem prover. Try to use someone else’s”.

The fundamental difference between the two embeddings of Z surveyed in the IST report is the *level* of embedding of the Z language in HOL. By ‘level of embedding’ we mean the degree of correspondence between the terms of the Z language and the defined terms in the target system. A very deep embedding would define a semantic function relating Z terms to terms of the target logic, and could therefore allow the user to interact with the system in terms of Z . In a very shallow embedding, the terms of Z would be translated to the terms of the target logic, the user then being required to interact with the system in terms of its native logic.

It is easy to justify the use of both styles of embedding. For deep embedding, the justification is that users need only be familiar with Z , as all interaction can take place in that style. ICL (the authors of ProofPower) claim this to be true for their system, although their user community tends to know both Z and HOL.

A shallow embedding is justified in terms of the efficiency and effectiveness of the underlying theorem prover. It is quite reasonable to suppose that the authors of a general-purpose theorem proving system have their own ‘style’ of writing specifications, which they would recommend to users of their system. The reasoning and automation in the prover will (by design) be more effective on specifications written in this style.

Translations from another specification notation into the logic of the theorem prover are likely not to be the most natural and effective way of representing the intent of the specifications within the native logic, and so miss

out on a lot of the most effective reasoning support in the theorem prover. Thus, in a shallow embedding the translation is made to preserve the intended meaning of the specification, but ignoring the specific method of representing that meaning. Another way of distinguishing these embeddings is to label them as syntactic embeddings (deep) and semantic embeddings (shallow). Details of the specific issues and tradeoffs involved in embedding Z in other logics can be found in a paper by Bowen and Gordon[107].

The two embeddings of Z in HOL represent a deep (ProofPower) and a shallow (Z/HOL) embedding. Z in HOL is quite unusual for a shallow embedding of one logic in another, in that it maintains a table of reverse translations, so that results of tactic applications are as far as possible returned to the user in terms of the original Z notation. However in 1994, Z/HOL and ProofPower were very much still in development, and ProofPower was very expensive, even for academic use.

The major disadvantage of embedding Z in the framework of an existing verification system is the need to show that the translation is meaning preserving, i.e. that the resultant specification in, say, HOL has the same meaning as the Z specification from which it originates. This is very similar to the compiler correctness problem, especially as the problem of giving a formal semantics exists for specification languages as much as it does for programming languages.

4.2.1 The Choice of PVS

Following the above review of verification support for Z specifications, it is clear that (at the time) purpose-built support for reasoning in Z was not mature enough and did not have sufficient support for highly automated proofs to be of effective use in our goal of attaining more automatic proofs of compiler translations. The only system in which Z had been embedded by that time was HOL, and those embeddings were also immature, so support for highly-automated proofs in HOL was not well developed.

Our previous experience with using verification systems inclined us towards using the general purpose verification system PVS, which had performed very well on industrial-scale problems in, for example, hardware verification[110] and fault tolerance[111]. The ability to embed Z within a HOL framework indicated that the higher-order logic of PVS would be a suitable reasoning system.

Thus, we chose to test the suitability of PVS for this problem. As our intent was to develop a system for proofs of compiler correctness rather than to develop a general purpose proof tool for Z specifications, we chose to perform a very shallow embedding of the Z specification of the DCC compiler into PVS. Thus we retain the intent and meaning³ of the specification, but make available all of the advanced theorem proving capabilities of PVS. This must however be balanced against the traceability from Z to PVS, by which we mean the PVS specification must be able to be compared directly with the Z specification so as to affirm that we have translated the semantics correctly. The result of this experiment is reported in the discussion chapter.

4.3 PVS

PVS is the most recent in a line of specification languages, theorem provers and verification systems developed at SRI, dating back over 20 years. PVS consists of a specification language, a library of predefined theories, a theorem prover and various utilities. In our earlier classification of formal methods PVS lies in classification three — the specification language and verification engine were developed together, based on SRI’s previous experiences with (inter alia) the Jovial Verification System[113], the Hierarchical Development Methodology (HDM)[114], STP[115] and EHDM[116].

This section presents a brief overview of the constructs available in the PVS specification language to make it easier for the reader to understand the specification fragments given later in this thesis. The available proof commands are also introduced in a table. An example of a full specification and proof for a small example (GCD) is given in Appendix A.

³Z does not currently have a well-defined formal semantics, so it could be argued that the meaning of any given Z specification is not precise and is open to interpretation. Here we are using what we feel to be the ‘intuitive’ meaning of the specification in the translation and do not concern ourselves with the issues of the dark corners of the precise Z semantics (whatever they may turn out to be). This has the corollary that we cannot prove any correspondence between our specification of the semantics and the original DCC specification.

For further discussion of the emerging formal semantics of Z the reader is referred to[112].

4.3.1 Specification Language

The specification language of PVS is based on classical, typed higher-order logic. It contains the usual constructs found in specification languages (such as functions, arithmetic and logical expressions) and familiar programming idioms such as records and arrays. The tight integration of specification language and theorem prover allows PVS to exploit theorem proving in the process of type checking, thus allowing for constructs whose type correctness is undecidable. One such example is predicate subtypes, where a type may be defined based on an arbitrary predicate over a base type.

Specifications are organized into parameterized theories, with an optional list of assumptions constraining the parameters. Theories may also be imported, creating a hierarchy. Theories may contain definitions, axioms and theorems. For example, the following theory has two parameters, one of which is constrained with an assumption:

```
Example [X : TYPE, Y : nat] : THEORY
BEGIN

  IMPORTING ParentTheory

  ASSUMING
    y_is_square : ASSUMPTION
      EXISTS (i : nat) : i * i = Y
  END ASSUMING

  ...

END Example
```

The PVS base types include integers, booleans, reals and the ordinals up to ϵ_0 . In addition, users may introduce uninterpreted types, about which no assumption is made, including the fact that they may not have any members. Non-empty uninterpreted types may be declared using the keyword `TYPE+` or `NONEMPTY_TYPE`:

```
TypeOne   : TYPE           % Potentially Empty Type
TypeTwo   : NONEMPTY_TYPE  % Nonempty type
TypeThree : TYPE+         % Another Nonempty type
```

Simple type constructors include functions, tuples, records and enumerations:

```
f : TYPE = [int -> int]           % function from int to int
r : TYPE = [# a : int,           % two component record of
             b : bool #]         % integer and boolean
t : TYPE = [int, bool]           % two-tuple of int and bool
e : TYPE = {red, white, blue}    % Enumeration type of 3 elements
```

Components of records can be accessed according to their field names. Components of a tuple type are accessed using projection functions. For example, consider the types `r` and `t` above, which can be converted with the following functions:

```
r_to_t (my_r : r) : t
  ( a(my_r), b(my_r) )

t_to_r (my_t : t) : r
  (# a := proj_1(my_t),
   b := proj_2(my_t) #)
```

The most complex type constructor is the recursively-defined abstract data type. This can allow for the description of types such as binary trees (shown here as a tree whose leaves contain elements of type `T`):

```
tree[T : TYPE] : DATATYPE
BEGIN
  node (left, right : tree) : node?
  leaf (element      : T)   : leaf?
END tree
```

Here, `node` and `leaf` are known as *constructors*: functions which return an object of type `tree`. `left`, `right` and `element` are *accessors*: functions that access the components of a tree. `node?` and `leaf?` are *recognizers*: predicates which allow determination of an object of type `tree` as a `node` or a `leaf`.

PVS also supplies dependent types and predicate subtypes, which may be used to introduce types with general constraints, such as the type of the square numbers:

```
squares : TYPE = {i : nat | EXISTS (j : nat) : j*j = i}
```

These constrained types may incur proof obligations during type checking but greatly increase the expressiveness and naturalness of specifications. In practice most of the obligations are discharged automatically by the theorem prover. These obligations (known as Type Correctness Conditions, or TCCs) are also generated to ensure the termination of recursive function definitions, to ensure type compatibility and so on.

For example, if we declare a constant of the `squares` type above, a TCC is generated obliging us to prove that the type is non-empty (it is unsound to declare a constant of a non-empty type):

```
a : squares

% Existence TCC generated (line 9) for a: squares
%
a_TCC1: OBLIGATION (EXISTS (x: squares): TRUE);
```

PVS expressions provide the usual arithmetic and logical operators, function application, lambda abstraction and quantifiers, all following a traditional syntax. Names may be freely overloaded, including those of the built-in operators such as `AND` and `+`. A case expression provides pattern-matching over the constructors of abstract data types:

```
light : TYPE = {red, amber, green}

LightNumber (l : light) : int =
  CASES l OF
    green : 2,
    amber : 1,
    red   : 0
  ENDCASES
```

Functions may be defined as `RECURSIVE`. To ensure they are total, i.e. that the recursion terminates, a `MEASURE` function must be provided. This measure function must return a natural number which decreases on all recursive calls made. For example, a function to append one list to another:

```

l1, l2 : VAR list[T]

append(l1, l2): RECURSIVE list[T] =
  CASES l1 OF
    null: l2,
    cons(x, y): cons(x, append(y, l2))
  ENDCASES
MEASURE length(l1)

```

4.3.2 Verification Engine

The PVS theorem prover provides a collection of powerful primitive inference procedures that are applied interactively within a sequent calculus framework. The primitive inferences include propositional and quantifier rules, induction, rewriting, and decision procedures for linear arithmetic over both integers and reals and for Park's μ -calculus[117]. The implementations of these primitive inferences are optimized for large proofs: for example, propositional simplification and μ -calculus use BDDs⁴ and auto-rewrites are cached for efficiency.

User-defined procedures can combine these primitive inference steps (and previously defined procedures) to yield higher-level proof strategies, such as those developed for induction and CTL⁵ model checking[121]. Such proof strategies are sound with respect to the primitive inference steps. The primitive inferences are themselves quite complex and some involve calls to external packages such as the BDD simplifier and CTL model checker.

Proofs (complete and partial) yield scripts that can be edited, attached to additional formulas and rerun, either automatically or under the direction of the user in a step-wise manner. This allows many similar theorems

⁴Binary Decision Diagrams: an efficient method for the simplification of large propositional expressions[118, 119, 120].

⁵Computational Tree Logic.

to be proved efficiently, permits proofs to be adjusted economically to follow changes in requirements or design and encourages the development of readable proofs.

The following table gives a brief description of most of the basic commands that are used in the presentation of proofs later in this thesis.

Rule Class	PVS commands
Apply decision procedures and auto-rewrites	assert, simplify, do-rewrites, record
Propositional reasoning	bddsimp, prop, iff, flatten, split
Combine prop and assert	ground
Simplify if-then-else and WITH	lift-if
Iterate lift-if with bddsimp	smash
Case split	case-replace, case
Note type information	typepred
Skolemization	skosimp*, skosimp, skolem-typepred, skolem!, skolem
Instantiation	inst?, inst
Combine inst? with skosimp*, smash and assert	bash
Iterate bash	reduce
Setup auto-rewrites	install-rewrites, auto-rewrite, auto-rewrite-theory, auto-rewrite-theories
Setup auto-rewrites then reduce	grind
Beta reduction	beta
Lemma introduction	use*, use, forward-chain, lemma
Equality reasoning	replace, replace*
Definition expansion	expand
Conditional rewriting	rewrite, simplify-with-rewrites
Extensionality	replace-extensionality, apply-extensionality
Induction	induct-and-simplify, measure-induct-and-simplify, generalize, measure-induct, name-induct, induct
Naming	name-replace*, name, same-name

Rule Class	PVS commands
Apply eta rule	replace-eta, apply-eta, eta
CTL model checking	model-check
Control	quit, undo, postpone, rerun, help, hide, reveal
Command Combinators	apply, then, repeat, try, branch, spread, else, let, skip, fail

4.3.3 Further Information

PVS is further described in [122, 123, 124]. The use of the more powerful strategies for more automatic theorem proving (with the user only directing the key steps) is described in a technical report by the author [2].

PVS is freely available, under licence to SRI International. For further details and to download a copy via ftp, please refer to the SRI Computer Science Laboratory's web site at <http://www.cs1.sri.com/pvs.html>.

4.4 Summary

This chapter has explored some of the issues involved in the development of tool support for verification and given a brief summary of the tool support available for verification in the Z specification language. We have also justified the choice of PVS for this work and given some details of the specification language and theorem prover of PVS.

Part II

Mechanizing the DCC Method

Chapter 5

The Starting Point

This chapter gives an overview of the DCC method developed at Logica Cambridge, upon which our work is based. The group at Logica developed a *demonstrably correct* compiler for a small (but non-trivial) high-level language Tosca targeted at the fictional assembly language Aida (described in Stepney's book[3]) and at Vital, an assembly language for the VIPER[97, 98] microprocessor (described in a paper in *Formal Aspects of Computing*[15]).

We refer the reader to Appendix C, which gives a key to the function names, types and domains used in the fragments of Z specifications appearing here, which may be useful in understanding their meaning.

5.1 The Languages

The source language, Tosca, is a simple high-level language. It contains the usual features of a high-level language — sequencing, assignment, choice, while loops, and a simple model of input/output based on streams of integers. The target language, Aida, is a simple assembly language with load/store, input/output, and arithmetic/logical operations.

Both languages are defined denotationally within the Z specification language[100] and the source language is given an operational semantics in terms of templates of target instructions. This operational semantics defines the compiler — the sequence of target instructions it should generate for each source language construct.

5.2 The DCC Method

The development of a compiler by the DCC method has three components¹:

1. **Specification:** The denotational semantics of both the high-level source language (Tosca) and the low-level assembly language (Aida) are specified in Z , as is the operational semantics of the high-level language in the form of a set of templates of low-level language instructions.

The denotational semantics act as the language definition and so the specification style should be as clear and abstract as possible, to make the definition comprehensible to human readers[125].

2. **Implementation:** The Z specification of the Tosca semantics are translated into Prolog, where they are executable. Executing the denotational semantics gives an interpreter; executing the operational semantics gives a compiler (i.e. the execution over a Tosca program yields an Aida program).

The declarative language Prolog, rather than some imperative language, is chosen as the target language in order to minimize this implementation step.

3. **Proof:** The operational semantics are proved to be equivalent to the denotational semantics: the compiler transformation is *meaning preserving* and hence the compiler is correct.

The various semantics need to be manipulated mathematically, in order to perform the correctness proofs, and so should be written as abstractly as possible.

Having stated that our aim here is to reduce the effort required in the proof process, the implementation of the compiler does not concern us here. The interested reader is referred to Stepney's book[3] for further details of the Prolog implementation.

The DCC method assumes the correctness of the lexical analysis and syntax analysis phases of compilation (described in Section 2). The semantic functions presented in this chapter represent the semantic analysis and code generation phases of compilation. There is no intermediate code used, nor any optimization performed.

¹In the original work the only tool available was *fuzz*, a Z type checker[99]. No proof tools were used and hence 'suitable for tools' was not a design criterion.

5.3 Industrial Usage of the DCC Method

The compiler presented here is small and thereby provides a suitable vehicle for the demonstration of our principles without becoming overwhelmed with the scale of a typical industrial programming language. However, it is definitely not a ‘toy’ compiler — an extension of it is in current industrial usage by a client of Logica (AWE plc) for programming a safety critical embedded system on a simple 8-bit processor. The extended version contains:

- data types:
 - 16 bit signed and unsigned;
 - 8 bit unsigned (bytes);
 - Boolean;
 - enumerated types;
 - subranges; and
 - arrays.
- Expressions:
 - literals;
 - variable references;
 - arithmetic (+, −, ×, ÷, mod, =); and
 - function call.
- statements:
 - block (sequencing);
 - conditional (if-then-else);
 - while loop;
 - selection (case); and
 - procedure call.
- structure:
 - separately compiled modules with import/export.

- and special hardware specific support for read only (input stream), write only (output stream) and specifically located variables.

The extensions to the language from that presented in Stepney’s book have not been treated with the same degree of rigour, in that the translation proofs are merely sketches. The extended compiler has been subjected to ‘stress testing’ by NPL who noted several errors, but only in those areas of the compiler which had not been subjected to proof².

5.4 Example Semantic Functions — Assignment

To illustrate the DCC method we will use Tosca’s assignment statement and its compilation as a running example for the rest of this chapter. The meaning of an assignment $\mathbf{x} := \mathbf{e}$ is a state change: the state σ is updated so that the variable stored in x (denoted by $\rho[\mathbf{x}]$, where ρ is the environment mapping from variable names to store locations) takes on the value of the expression \mathbf{e} . In Z this can be written as³:

$$\left| \begin{array}{l} \mathcal{M}_C[\mathit{assign}(\mathbf{x}, \mathbf{e})] \rho \sigma = \\ \sigma \oplus \{ \rho[\mathbf{x}] \mapsto \mathcal{M}_E[\mathbf{e}] \rho \sigma \} \end{array} \right.$$

Note that this dynamic semantics for assignment is not a total function — it is only well-defined if it meets the following conditions:

- the variable \mathbf{x} and all variables in the expression have been previously declared;
- the variable \mathbf{x} is type compatible with the expression \mathbf{e} ; and
- all variables in the expression have been initialized before use.

These three properties are encompassed in the static semantic functions for Tosca. There is a ‘chain’ in the semantics such that later semantic functions are only meaningful if each of the earlier semantic functions are well defined. That is to say:

²Noted in private communication by Stepney, 1998.

³The square brackets $[\]$ are the conventional denotational semantics brackets, *not* Z ’s bag brackets.

- the declaration before use semantics must be well defined; then
- the type checking semantics must be well defined; then
- the initialization before use semantics must be well defined; then
- the dynamic semantics gives the meaning of the Tosca construct.

To illustrate this further, we present here the static semantic functions for the Tosca assignment statement.

Declaration before use semantics:⁴

$$\left| \begin{array}{l} \mathcal{D}_C[\mathit{assign}(\mathbf{x}, \mathbf{e})]\rho_\delta = \\ \quad (\mathbf{if} \ \mathbf{x} \in \text{dom } \rho_\delta \ \mathbf{then} \ \mathit{checkOK} \ \mathbf{else} \ \mathit{checkWrong}) \\ \quad \bowtie \mathcal{D}_E[\mathbf{e}]\rho_\delta \end{array} \right.$$

If the variable \mathbf{x} is in the domain of ρ_δ , then it has been previously declared, so it is safe to assign values to it. This check result is combined with the result from \mathcal{D}_E , which performs declaration before use checking of the expression \mathbf{e} within the same environment, ρ_δ .

Type-checking semantics:

$$\left| \begin{array}{l} \mathcal{T}_C[\mathit{assign}(\mathbf{x}, \mathbf{e})]\rho_\tau = \\ \quad \mathbf{if} \ \rho_\tau[\mathbf{x}] = \mathcal{T}_E[\mathbf{e}]\rho_\tau \neq \mathit{typeWrong} \\ \quad \mathbf{then} \ \mathit{checkOK} \\ \quad \mathbf{else} \ \mathit{checkWrong} \end{array} \right.$$

The type of the variable \mathbf{x} is stored in the environment ρ_τ when it is declared. This type should be the same as the type of the expression \mathbf{e} as given by the semantic function \mathcal{T}_E , and neither should type check to $\mathit{typeWrong}$.

⁴The operator \bowtie is a pessimistic combination of check results. The combination is $\mathit{checkOK}$ only if both of the arguments are $\mathit{checkOK}$.

Initialization before use semantics:⁵

$$\left| \begin{array}{l} \mathcal{U}_C[\mathit{assign}(\mathbf{x}, \mathbf{e})]_{\rho_\nu \sigma_\nu} = \\ \quad (\mathbf{let} \lambda == \rho_\nu[\mathbf{x}]; \sigma_{\nu 1} == \mathcal{U}_E[\mathbf{e}]_{\rho_\nu \sigma_\nu} \bullet \\ \quad \quad \mathbf{if} \lambda \in \text{dom}(\mathit{storeOf}_U \sigma_\nu) \\ \quad \quad \quad \mathbf{then} \sigma_{\nu 1} \\ \quad \quad \quad \mathbf{else} \sigma_{\nu 1} \boxplus_\nu \{\lambda \mapsto \mathit{checkOK}\}) \end{array} \right.$$

λ is the location at which the variable \mathbf{x} is stored, as given by the environment ρ_ν . $\sigma_{\nu 1}$ is the declaration before use state of the expression \mathbf{e} . We then calculate the declaration before use state of the assignment by checking if λ is in the domain of $\mathit{storeOf}_U \sigma_\nu$, the declaration before use store. If it is not, then this is the first assignment to variable \mathbf{x} , so we update the declaration before use store with $\mathit{checkOK}$ for this location (λ).

5.5 Example Aida semantics

In a similar manner to the Tosca meaning functions, the meaning of Aida's store instruction (store the contents of the accumulator at the location 1) updates its store⁶. In Z^7 :

$$\left| \mathcal{M}_I[\mathit{store} \ 1]_{\rho_i \vartheta \sigma_i} = \vartheta(\sigma_i \boxplus \{l \mapsto \mathit{storeOf}_I \sigma_i A\}) \right.$$

Here, σ_i represents the Aida store, ρ_i the environment mapping from program labels to continuations, ϑ the current continuation, and the term $\sigma_i A$ the value contained in the accumulator.

There are no static semantic functions given for Aida programs. This does not however imply that the Aida meaning functions give a total semantics. The definition of Aida given in Stepney's book is a partial semantics and defines just that portion of the language which is required as a target for the compiler. It is claimed (reasonably so) that if the Tosca source program

⁵ \boxplus_ν updates the store component of a use state, i.e. $(\varsigma_{\nu 1}, c_1) \boxplus_\nu \varsigma_{\nu 2} = (\varsigma_{\nu 1} \oplus \varsigma_{\nu 2}, c_1)$.

⁶The semantics of Aida is more complicated than Tosca's and needs to use 'continuation' arguments because the assembly language allows arbitrary jumps. See [3] for more explanation.

⁷ \boxplus updates the store component of a dynamic state, i.e. $(\varsigma_1, \mathit{in}, \mathit{out}) \boxplus \varsigma_2 = (\varsigma_1 \oplus \varsigma_2, \mathit{in}, \mathit{out})$.

is well defined and passes all of the static semantic checks, then an Aida program which is generated by the compiler from this must also be well formed⁸.

5.6 Example Compilation Function

In this section we give the operational semantics for the Tosca assignment statement. The operational semantics define the sequence of Aida instructions generated during the compilation of each Tosca instruction. The assignment statement is translated into a sequence of instructions to evaluate the expression, followed by an instruction to store that value at the appropriate location. In Z^9 :

$$\left| \mathcal{O}_C \langle \langle \text{assign}(\mathbf{x}, \mathbf{e}) \rangle \rangle_{\rho_o} \phi = (\phi, \mathcal{O}_E \langle \langle \mathbf{e} \rangle \rangle_{\rho_o \text{top}} \wedge \langle \text{store}(\rho_o[\mathbf{x}]) \rangle \rangle)$$

The ϕ in the above is a program label. Labels are used in Aida as the target of `goto` and `jump` instructions, and are generated in the output to support the program flow control commands such as `loop` and `if-then-else`.

5.7 Proofs

To show that the compiler is correct, we need to show that the translation into Aida preserves the semantics of the Tosca instruction. This is performed by structural induction over Tosca's abstract syntax (in the same manner as other compiler proofs from denotational semantics), proving that for each construct, the Aida meaning of the translation template is the same as the Tosca meaning of the original instruction. Pictorially, this can be represented as shown in Figure 5.1.

The left hand side of the diagram represents the state of the high and low-level programs and the compiler, before the execution of the command γ . The right hand side represents the states after execution of the command,

⁸In our mechanized treatment of the DCC method, the Aida semantics are total and include checks that Aida programs are well formed.

⁹The brackets $\langle \langle \rangle \rangle$ are another form of semantic brackets, used here to indicate operational semantics. They are not to be confused with the similar, standard Z , symbols for relational image ($\langle \langle \rangle \rangle$).

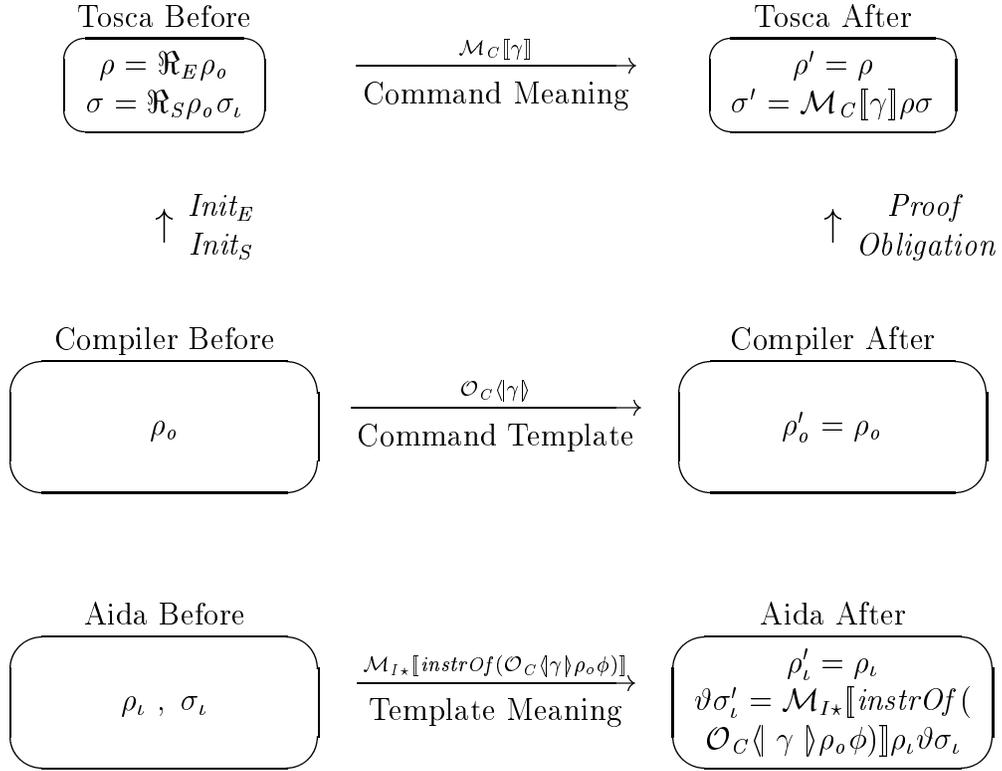


Figure 5.1: Pictorial Representation of the DCC Compiler Proof

according to the state transition represented by the various meaning functions for Tosca (\mathcal{M}_C), Aida (\mathcal{M}_{I^*}) and the compiler (\mathcal{O}_C).

$Init_S$ and $Init_E$ assert that the state and environment before execution of the command are ‘equivalent’ between the high and low-level, related by \mathfrak{R}_S (the state retrieve function) and \mathfrak{R}_E (the environment retrieve function). \mathfrak{R}_S and \mathfrak{R}_E are defined as follows:

$$Init_E == \rho = \mathfrak{R}_E \rho_o$$

$$\left| \begin{array}{l} \mathfrak{R}_E : Env_o \rightarrow Env \\ \hline \forall \rho_o : Env_o \bullet \mathfrak{R}_E \rho_o = \rho_o \end{array} \right.$$

\mathfrak{R}_E is an identity function. The memory allocation scheme used in the DCC compiler is the same at both the Tosca and Aida levels, therefore the environments Env_o and Env should be identical.

$$Init_S == \sigma = \mathfrak{R}_S \rho_o \sigma_i$$

$$\left| \begin{array}{l} \mathfrak{R}_S : Env_o \rightarrow State_I \rightarrow State \\ \hline \forall \rho_o : Env_o; \varsigma_i : Store_I; in : Input; out : Output \bullet \\ \mathfrak{R}_S \rho_o(\varsigma_i, in, out) = ((\text{ran } \rho_o) \triangleleft \varsigma_i, in, out) \end{array} \right.$$

\mathfrak{R}_S is more complex as the low-level state (σ_i) contains more information than the high-level state (σ). The range restriction performed on ς_i allows us to compare high and low-level states whilst ignoring the extra information — the accumulator and stack temporaries.

From this, the proof obligation is to show that the diagram commutes. Thus, we are required to show:

$$\begin{array}{l} Init_E; Init_S; \\ \rho'_i = \rho; \sigma'_i = \mathcal{M}_C[\![assign(\mathbf{x}, \mathbf{e})]\!] \rho \sigma; \\ \rho'_o = \rho_o; \\ \rho'_i = \rho_i; \\ \vartheta \sigma'_i = \mathcal{M}_{I^*}[\![instrOf(\mathcal{O}_C \langle \![assign(\mathbf{x}, \mathbf{e})]\!] \rho_o \phi)\!] \rho_i \vartheta \sigma_i \\ \vdash \sigma'_i = \mathfrak{R}_S(\rho_o \sigma'_i) \end{array}$$

We now present the proof for the Tosca assignment statement. This involves the use of a lemma, which shows the effect of the state transition at the low-level. This lemma is then used as the first step in the full correctness proof. First, the lemma:

$$\begin{aligned}
& \text{restrict}(\rho_o, \text{top}, \vartheta\sigma'_l) \\
&= \text{restrict}(\rho_o, \text{top}, \mathcal{M}_{I\star}[\text{instrOf}(\mathcal{O}_C \langle \text{assign}(\mathbf{x}, \mathbf{e}) \rangle \rho_o \phi)] \rho_l \vartheta\sigma_l) && \text{[Defn of } \sigma'_l\text{]} \\
&= \text{restrict}(\rho_o, \text{top}, \mathcal{M}_{I\star}[\mathcal{O}_E \langle \mathbf{e} \rangle \rho_o \text{top} \wedge \langle \text{store } \rho_o[\mathbf{e}] \rangle] \rho_l \vartheta\sigma_l) && \text{[}\mathcal{O}_C\text{ template]} \\
&= \text{restrict}(\rho_o, \text{top}, \mathcal{M}_{I\star}[\mathcal{O}_E \langle \mathbf{e} \rangle \rho_o \text{top}] \rho_l (\mathcal{M}_I[\text{store } \rho_o[\mathbf{e}]] \rho_l \vartheta) \sigma_l) && \text{[Defn of } \mathcal{M}_{I\star}\text{]} \\
&= \text{restrict}(\rho_o, \text{top}, \mathcal{M}_I[\text{store } \rho_o[\mathbf{e}]] \rho_l \vartheta(\sigma_l \boxplus \{A \mapsto \mathcal{M}_E[\mathbf{e}]\rho\sigma\})) && \text{[Induction Hypothesis]} \\
&= \text{restrict}(\rho_o, \text{top}, \vartheta(\sigma_l \boxplus \{A \mapsto \mathcal{M}_E[\mathbf{e}]\rho\sigma, \rho_o[\mathbf{e}] \mapsto \mathcal{M}_E[\mathbf{e}]\rho\sigma\})) && \text{[Defn of } \mathcal{M}_I\text{]}
\end{aligned}$$

□

The full definition of the steps used is given in Stepney's book[3]. To summarize the proof in English:

- replace $\vartheta\sigma'_l$ according to its definition (the final Aida state in Figure 5.1);
- expand the definition of the compiler (\mathcal{O}_C) for assignments;
- expand the definition of $\mathcal{M}_{I\star}$, the Aida meaning function for lists of Aida instructions;
- use the induction hypothesis that the embedded expression (\mathbf{e}) is correctly compiled, and leaves the value result of the expression ($\mathcal{M}_E[\mathbf{e}]\rho\sigma$) in the accumulator (A); and
- expand the Aida meaning function (\mathcal{M}_I) for the store instruction.

This lemma is then used in the following proof of the translation mechanism:

$$\begin{aligned}
& \mathfrak{R}_S \rho_o \sigma'_l \\
&= \mathfrak{R}_S \rho_o \mathit{restrict}(\rho_o, \mathit{top}, \sigma'_l) && \text{[Lemma r2]} \\
&= \mathfrak{R}_S \rho_o \mathit{restrict}(\rho_o, \mathit{top}, \\
&\quad \sigma_l \boxplus \{A \mapsto \mathcal{M}_E[\mathbf{e}]\rho\sigma, \rho_o[\mathbf{e}] \mapsto \mathcal{M}_E[\mathbf{e}]\rho\sigma\}) && \text{[Lemma]} \\
&= \mathfrak{R}_S \rho_o (\sigma_l \boxplus \{\rho_o[\mathbf{e}] \mapsto \mathcal{M}_E[\mathbf{e}]\rho\sigma\}) && \text{[Defn } \mathit{restrict}\text{]} \\
&= \sigma \boxplus \{\rho_o[\mathbf{e}] \mapsto \mathcal{M}_E[\mathbf{e}]\rho\sigma\} && \text{[Defn } \mathfrak{R}_S, \sigma\text{]} \\
&= \sigma \boxplus \{\rho[\mathbf{e}] \mapsto \mathcal{M}_E[\mathbf{e}]\rho\sigma\} && \text{[Defn } \rho\text{]} \\
&= \mathcal{M}_C[\mathit{assign}(\mathbf{x}, \mathbf{e})]\rho\sigma && \text{[Defn } \mathcal{M}_C\text{]} \\
&= \sigma' && \text{[Defn } \sigma'\text{]}
\end{aligned}$$

□

Again, to summarize in English:

- use lemma ‘r2’, which states that applying \mathfrak{R}_S to a restricted state has the same result as applying it to an unrestricted state;
- apply the lemma we have just shown above, with ϑ as identity;
- expand the definition of $\mathit{restrict}$, which removes the accumulator from consideration;
- using the definitions of \mathfrak{R}_S and σ , we move to reasoning about the high-level state;
- using the definitions of \mathfrak{R}_E and ρ , we know $\rho_o = \rho$;
- we have now manipulated the formula into the precise definition of \mathcal{M}_C for assignment functions, so we replace according to that definition; and
- that is equal to σ' , according to the Tosca final state.

This proof is typical of those presented in Stepney’s book. The interested reader is referred to her book for detail of the other proofs and the specification of the meaning functions.

5.8 Summary

This chapter has presented an overview of the development of a demonstrably correct compiler by the DCC method. We have followed an example (assignment) of the semantic functions, compiler definition, and the proof that the translation is meaning preserving with respect to the semantics.

In the next chapter, we present a discussion on tool support for verification in order to justify our selection of tool for the mechanization of the specification and proofs presented here.

Chapter 6

From Z to PVS

This chapter describes the translation of the DCC semantics for Tosca and Aida from Z into PVS. The description is based on that given in our paper presented at Formal Methods Europe '97, co-authored by Professors Susan Stepney and Ian Wand. The paper[1] was entitled “Using PVS to prove a Z refinement: A Case Study”.

6.1 Translation to PVS

During the process of translating the Z semantics of Tosca, Aida and the compiler into PVS we experimented widely with various methods of expressing the semantics in the total, higher order logic of PVS. In the following sections we illustrate some of the key points in this translation. Hence, this chapter is more an illustration of the techniques adopted than a coherent approach to the general problem of translating Z specifications into the logic of PVS.

At first we took a direct and naive approach to the translation and performed a simple direct mapping of the Z specifications into PVS. This allowed us to experiment quickly with the PVS type system and prover to see how suitable they would be to perform the compiler verification. However, this approach turned out not to be as naive as we had thought — the type system of PVS made explicit to us all of the incompleteness (by which we mean definitions of and applications of partial functions) in the translated specification. This was achieved by the Type Correctness Conditions (TCCs) emitted during type checking.

This behaviour demonstrates the value of the use of a specification language with such strong type checking in the process of language design, allowing designers to explore alternatives in the semantic functions with a rigorous degree of mechanical checking to ensure consistency and completeness of the definitions.

For our purposes this process allowed us to produce a complete semantics of Tosca and Aida, with a large confidence in the correctness of the translation from Z to PVS. We also extended the semantics to cover various cases in both languages which were not explicit in the DCC semantics, for example, the effect of arithmetic overflow.

The following sections describe some of the more interesting features of the translation and how the type system of PVS was exploited to overcome them.

6.1.1 PVS Types

As we noted previously, the proof engine of PVS is based on decision procedures, tightly integrated with rewriting, which allows it to automate many proof steps which are mundane and tedious in other verification systems. The ground prover makes heavy use of the type information available in a specification, which can be used to simplify and in some cases automatically prove a sequent.

A major part of our experiment into how to express the Z semantics of the Tosca compiler within PVS dealt with how various styles of specification relate to the ability to automate efficiently proofs of properties of that specification. In many cases, we were able to minimize side conditions generated during a proof (or reduce their generation to trivial conditions) by augmenting the type of various constructs. As a concrete example we present here the function head of \mathcal{O}_E , the operational semantics (i.e. compiler) for Tosca's expressions:

$$\mathcal{O}_E \langle \epsilon : Expr \rangle (\rho_o : Inj_{Env})(SP : \{l : Locn \mid l \geq top(\rho_o)\}) :$$

$$RECURSIVE \{l : list[Instr] \mid cons?(l) \wedge sequential?(l)\} =$$

...

In this example the function \mathcal{O}_E takes an expression ϵ and translates this into a list of Aida instructions to evaluate that expression. When the Aida meaning function is applied to this list of instructions, side conditions are generated to ensure that the sequence reaches a final state. This requires there to be no `jump` or `goto` instructions in the generated list which would cause control to pass elsewhere. To reduce these to trivial conditions we constrain the return type of the function so that the returned list of Aida instructions satisfies the predicate *sequential?*, which we defined as:

$$\begin{aligned} \textit{sequential?}(l : \textit{list}[\textit{Instr}]) : \textit{bool} = \\ \textit{every}(\lambda(\iota : \textit{Instr}) : \neg (\textit{goto?}(\iota) \vee \textit{jump?}(\iota)))(l) \end{aligned}$$

every is a predefined PVS function which takes a predicate and a list, and is true if every item in the list satisfies the predicate. It is defined by recursing over the target list and is equivalent to $\forall i : i \in \textit{list} \Rightarrow \textit{pred}(i)$.

The range type of \mathcal{O}_E is also constrained as a *cons?*, which is a non-empty list (terminology taken from the syntax of Lisp). From the augmented range type of \mathcal{O}_E the prover can now automatically determine that the Aida code generated for expressions does not contain any jumps, and will reach a final state.

6.1.2 Total functions

Partial functions are a natural method for the modelling of many situations. For example, division over the integers is a partial function as it does not admit ‘0’ as a divisor. The Z specification language supports partial functions and even provides notation for classifying further the type of function, for example as an injection, or surjection.

It is known to be difficult to reason efficiently about equality in the presence of partial functions[126]. Hence PVS does not support them, but instead provides predicate subtypes which can be used to *emulate* the effect of partial functions, by replacing them with total functions over a restricted domain.

In our compiler specification we need to convert the partial functions in the Z semantics into total functions in PVS. There are four basic methods for making a partial function into a total one:

1. cause the function to return an arbitrary value of the correct type for undefined cases;
2. cause the function to return a specific, fictitious value (a *bottom*) for undefined cases;
3. constrain the type of the function arguments so that it is a total function over a restricted domain¹; and
4. directly model partial functions as a subtype of relations.

The first option can be achieved by the use of Hilbert's epsilon operator[127], which, given a predicate, returns a value which satisfies that predicate (if possible) and if it is not satisfiable returns an arbitrary value of the appropriate type. Epsilon is a useful specification tool, as it allows for a clever abstraction from detail, but is not really effective for defining programming language semantics. We wish the semantics to be a valid, *correct* description of the behaviour of the language in all circumstances, so to return an arbitrary *valid* value in 'undefined' cases seems to imply that the behaviour is indeed defined².

In our specification of the compiler problem in PVS we have used a combination of methods two, three and four from the above list. We believe there is a tradeoff in their use between the readability of the specification, and ease of proof.

Whereas all of these methods make explicit the inherent partial nature of a function, the use of a bottom element (method two) tends to clutter the body of the function. The use of complex types (method three) tends to make the function 'head' quite opaque with the extra constraints on the domain or range types. The direct modelling of partial functions (method four) generates neat specifications but complicates the proof process as there is no direct support for this in PVS.

¹This can also be achieved in Z , but type checking in Z will not catch a restricted domain function applied out of domain (nor would any decidable type system).

²See section 2.4.2 of [2] for more discussion on the use of the epsilon operator in PVS.

A major gain in the use of complex types is that the type checker can provide help in noting errors in the specification before the expensive process of theorem proving has begun. Where the domain of a function is tightly constrained the application of a function outside its domain becomes a type error and is highlighted in PVS by the generation of an unprovable TCC. This is a similar argument as that for strong type systems in programming languages, in that errors which are noted earlier in the development life-cycle are cheaper and easier to rectify.

In our specification we have used direct modelling wherever we need to reason about the domain of a function (for example the environment types). In most other cases we use complex types, the exceptions being where the lack of mutual recursion in PVS will not allow this, when a bottom element is used.

6.1.3 Initial Environment

A clear example of the use of the last method of translating partial functions into total functions is the initial environment hypothesis. In the hand proofs, this hypothesis (denoted by $Init_E$, defined in Section 5.7) asserts that the environment that maps from variable names to locations is the same in the high and low-level languages.

In our treatment, we extend this function from an identity to the following:

$$\begin{aligned}
 &Init_E(\rho : Env_{Store}, \rho_o : Inj_{Env}, \rho_t : Env_T) : bool = \\
 &Env(\rho) = \rho_o \wedge \\
 &domain(map(\rho_o)) = domain(\rho_t) \wedge \\
 &(\forall(\xi : Name) : domain(\rho_t)(\xi) \Rightarrow \neg TypeWrong?(apply(\rho_t, \xi)))
 \end{aligned}$$

This includes the type checking environment as an additional argument and asserts the following.

- That the domain of ρ (the environment which maps from variable names to locations) is the same as the domain of ρ_t . This implies that all variables have been type checked.
- That all names in the domain of ρ_t do not have a type of $TypeWrong$.

The connection made between the domains of ρ and ρ_t enables us to reason about the type of objects during proof. Its utility will be demonstrated in chapter 7.

6.2 Changes to the Original Semantics

In the translation of the compiler semantics into PVS we have taken several liberties with the original structuring of the DCC semantics in order to make the process of theorem proving more automatic and in some cases to make the theorems more general. This section describes the extensions to the original semantics and how they have aided the more automatic proof of the compiler.

6.2.1 Environment and Store

The most significant change to the semantics involved the combination of store and environment into one type, so that constraints between the two could be expressed within the type definition itself rather than as external axioms. In this section we present a full worked example of how a semantic domain in Z is translated into a PVS type with full constraints.

It is necessary for the semantic description of a programming language to provide definitions of the environment that maps from variable names to locations in store and a store giving a mapping from store locations to values. The Stepney semantics provide these functions at the Tosca level as:

$$\begin{aligned} \text{Locn} &== \mathbb{Z} \\ \text{Env} &== \text{NAME} \rightsquigarrow \text{Locn} \\ \text{Store} &== \text{Locn} \rightarrow \text{VALUE} \end{aligned}$$

Naively, these translate into the following PVS definitions:

$$\begin{aligned} \text{Locn} : \text{TYPE} &= \text{int} \\ \text{Env} : \text{TYPE} &= [\text{Name} \rightarrow \text{Locn}] \\ \text{Store} : \text{TYPE} &= [\text{Locn} \rightarrow \text{Value}] \end{aligned}$$

In the translation we have lost the information that the two functions are partial and that Env is an injection. We use a general purpose definition of partial functions as an extension of relations, based on that given by Rushby and Stringer-Calvert in [2] (this specification is reproduced in Appendix B of this thesis). This gives us:

$$\begin{aligned} Locn \text{ } TYPE &= int \\ Env \text{ } TYPE &= (part_inj[Name, Locn]) \\ Store \text{ } TYPE &= (functional[Locn, Value]) \end{aligned}$$

We have now captured all of the information in the original Z function definitions. Using our knowledge of what these functions actually denote we could add the extra constraint that we are only interested in the values stored in locations which have been allocated to store variables. The original semantics defines a memory allocation scheme where variables are allocated to store locations sequentially from 1 to top , where top is the store location one above the highest currently allocated. Hence we have:

$$\begin{aligned} Locn \text{ } TYPE &= int \\ top &: Locn \\ Env \text{ } TYPE &= (part_inj[Name, Locn]) \\ Store \text{ } TYPE &= [\{l : Locn \mid l < top\} \rightarrow Value] \end{aligned}$$

Note that we have effectively removed the need to directly model the partial nature of the $Store$ function by explicitly stating the constraint on its domain which gives rise to its partial nature. We can also use a predicate subtype to define the constraint on the range of the environment Env , expressing it as a PVS record type in combination with top :

$$\begin{aligned} Inj_{Env} \text{ } TYPE &= \\ &[\# top : Locn, \\ &\quad map : \{m : (part_inj[Name, Locn]) \mid \\ &\quad\quad \forall(l : Locn) : (l \in range(m)) \Rightarrow l < top\} \#] \end{aligned}$$

We can now combine this with the store to give us a type which expresses all of the constraints on the Tosca store and environment:

$$\begin{aligned}
Locn &: TYPE = int \\
Inj_{Env} &: TYPE = \\
& \quad [\# top : Locn, \\
& \quad \quad map : \{m : (part_inj[Name, Locn]) \mid \\
& \quad \quad \quad \forall(l : Locn) : (l \in range(m)) \Rightarrow l < top\} \#] \\
Env_Store &: TYPE = \\
& \quad [\# Env : Inj_{Env}, \\
& \quad \quad Mem : [\{l : Locn \mid l < top(Env)\} \rightarrow Value] \#]
\end{aligned}$$

6.2.2 Aida State

A similar approach to the Aida semantic domains leaves us with the following PVS record type:

$$\begin{aligned}
Env_Store_I &: TYPE = [\# Env : Inj_Env, \\
& \quad A : Value, \\
& \quad SP : \{l : Locn \mid l \geq top(Env)\}, \\
& \quad Mem : [\{l : Locn \mid l < SP\} \rightarrow Value] \#]
\end{aligned}$$

Here ‘*A*’ represents the machine’s accumulator, which can hold any valid Tosca value³. Aida’s memory model is more complex, as it must assign storage to temporaries during the evaluation of binary expressions. These are allocated on a stack which grows from *top* towards high memory (which is unbounded).

After a stack temporary has been used in evaluating a binary expression it clearly has no further effect on the outcome of the computation. To model this, the Z specification defines a function *restrict*, which performs a domain restriction on the store to yield the locations that are currently in use. This is similar to the restriction involved in \mathfrak{R}_S to relate Tosca and Aida states.

³For simplicity the range of integers and the available expressions are the same in Tosca and Aida.

In this function the Aida state is restricted to consider just the accumulator, the locations to which Tosca variables are bound and any locations between top and $\lambda - 1$ (λ being the argument to $restrict$).

$$\left| \begin{array}{l} restrict : Env_O \times Locn \times State_I \rightarrow State_I \\ \hline \forall \rho_o : Env_O; \lambda : Locn; \varsigma_l : Store_I; in : Input; out : Output \bullet \\ \quad restrict(\rho_o, \lambda, (\varsigma_l, in, out)) = \\ \quad \quad ((\{A\} \cup \text{ran } \rho_o \cup (top .. \lambda - 1)) \triangleleft \varsigma_l, in, out) \end{array} \right.$$

Our new treatment replaces this function with a dependent type by introducing a stack pointer: **SP**. Note that this is merely a specification device for keeping track of active memory locations and does not necessarily need to be reflected in the underlying hardware.

A side effect of making our stack pointer part of the low-level state is that it becomes necessary to add two extra instructions to the instruction set of the low-level machine to increment (**spinc**) and decrement (**spdec**) the stack pointer. These were inserted into the operational semantics of Tosca at appropriate points in expression evaluation. **spinc** and **spdec** are defined as:

$$\begin{array}{l} \mathcal{M}_I \llbracket \gamma : Instr \rrbracket (\rho_i : Env_I) (\vartheta : Cont) : Cont = \\ \quad \lambda (\sigma_i : State_I) : \\ \quad \quad \text{CASES } \gamma \text{ OF} \\ \quad \quad \dots \\ \quad \quad \text{spinc :} \\ \quad \quad \quad \vartheta (\sigma_i \text{ WITH } [(StoreOf_I) := SP_INC(StoreOf_I(\sigma_i))]), \\ \quad \quad \text{spdec :} \\ \quad \quad \quad \vartheta (\sigma_i \text{ WITH } [(StoreOf_I) := SP_DEC(StoreOf_I(\sigma_i))]), \\ \quad \quad \dots \\ \quad \quad \text{ENDCASES} \end{array}$$

Where SP_INC and SP_DEC are defined as:

```

SP_INC( $\sigma_i : Env\_Store_I$ ) : Env_Store_I =
  (# Env := Env( $\sigma_i$ ),
   A := A( $\sigma_i$ ),
   SP := SP( $\sigma_i$ ) + 1,
   Mem := Mem( $\sigma_i$ ) WITH[(SP( $\sigma_i$ )) := IntV(0)]
  #)

SP_DEC( $\sigma_i : \{e : Env\_Store_I \mid SP(e) - 1 \geq top(Env(e))\}$ ) : Env_Store_I =
  (# Env := Env( $\sigma_i$ ),
   A := A( $\sigma_i$ ),
   SP := SP( $\sigma_i$ ) - 1,
   Mem := restrict(Mem( $\sigma_i$ ))
  #)

```

The decrement operation makes use of the PVS function *restrict*⁴, which takes a function defined over a type X and returns a function which has a domain which is a subtype of X :

```

% restrict is the restriction operator on functions, allowing a
% function defined on a super-type to be applied to a subtype. Note
% that it is a conversion, so will be inserted automatically when needed.
restrict[T : TYPE, S : TYPE FROM T, R : TYPE] : THEORY
BEGIN
  f : VAR [T → R]
  s : VAR S
  restrict(f)(s) : R = f(s)
  CONVERSION restrict

  injective_restrict : LEMMA
    injective?(f) ⇒ injective?(restrict(f))
END restrict

```

⁴This is not related to the Z function *restrict* defined earlier.

The explicit insertion of the PVS *restrict* in *SP_DEC* is not strictly necessary as *restrict* is defined in the prelude as a conversion. PVS conversions are automatically inserted by the type checker where necessary and appropriate in order to make two types compatible. Here we have a function (*Mem*) whose domain is too expansive, so it must be reduced appropriately.

6.2.3 Consequences

Earlier we stated that all declarations in Tosca are global and therefore the environment remains static during program execution. Having combined the environment and store in the same datatype the state transition functions in the dynamic semantics could (by virtue of their type) modify the environment. This could lead to unanticipated behaviour and invalidates the use (as a lemma) of any proof that the environment is correctly set-up before arbitrary command execution.

PVS points out this flaw by emitting TCCs during the proof of command translation obliging us to show that the environment remains static. For example, the last line of this TCC asks us to prove that the environment after executing $\rho_i(\phi)$ (a continuation) is the same as that in the initial state:

MI_TCC3 : *OBLIGATION*
 $(\forall(\phi : Label, \gamma : Instr, \rho_i : Env_I, \sigma_i : State_I, \vartheta : Cont) :$
 $\neg Step(\sigma_i) = 0 \wedge \gamma = goto(\phi) \Rightarrow$
 $Env(StoreOf_I(\rho_i(\phi)(\sigma_i))) = Env(StoreOf_I(\sigma_i))$

These TCCs are easily proved but it is also straightforward to avoid their generation at proof time by constraining the type of state transition functions so that they cannot modify the environment. The following definition is the augmented type of arbitrary state transitions (continuations) in the low-level language, including the condition that the environment must be static:

Cont : *TYPE* = $[s_1 : State_I \rightarrow \{s_2 : State_I \mid$
 $(Env(StoreOf_I(s_1)) = Env(StoreOf_I(s_2))) \wedge$
 $(Step(s_1) = 0 \Rightarrow s_1 = s_2)\}]$

6.2.4 Exceptional Behaviour

The original Z specification made heavy use of partial functions to structure the specification. For example, the dynamic meaning of Tosca's assignment statement ($\mathbf{x} := \mathbf{e}$) is partial: it says nothing about what state change occurs if, say,

- the lhs-name (\mathbf{x}) is undeclared
- there are some uninitialized values in the rhs-expression \mathbf{e}
- the lhs and rhs have different types

Making the dynamic meaning function total to include these checks would clutter the specification, and compromise the understandability of the language definition. The complete Tosca specification instead includes a static semantics which defines type checking, declaration before use checking and so on. If a Tosca statement passes these checks in the context of some program then it is guaranteed to be in the domain of the dynamic meaning function.

We have already noted that PVS does not admit partial functions and as such the direct implementation of the Tosca semantics generated many type correctness conditions highlighting the partial areas of the specification. Most of this partial nature is captured by the static checking semantics. For example all of the cases noted in the assignment example above.

However, there are cases which cannot be decided statically. Two which occur in the Tosca compiler are arithmetic overflow/underflow and 'running out' of input. The DCC semantics do not define what happens when arithmetic goes out of the range of Tosca integers so here we implemented a simple scheme that truncates out of range values:

```
Oflow(i : int) : Value =  
  IF i > MaxInt THEN  
    IntV(MaxInt)  
  ELSIF i < MinInt THEN  
    IntV(MinInt)  
  ELSE  
    IntV(i)  
  ENDIF
```

The other case, running out of input, occurs when the input stream (modelled as a list of integers) is not long enough to service all `input` statements in the code. We have explored two approaches to this:

1. cause an exception at the high-level (and correspondingly halting the low-level machine)⁵; and
2. assuming that the input stream will not run out at all.

The first of these clutters the specification terribly so that the high and low-level semantics must ensure that they perform no computation from exception states and that exception states are preserved by arbitrary continuations. The second is simpler, involving an axiom:

input_never_ends : AXIOM
 $\forall(\sigma : State) : cons?(InOf(\sigma))$

In the final version we have used this axiom to simplify our task and would suggest that the examination of exceptional behaviour be treated as an extension to the language — input handling being a special case of the more general problem of exception handling that could (for instance) be used in any necessary run-time type checking (e.g. array out-of-bounds accesses).

6.2.5 Termination

In her book[3], Stepney acknowledges that an assumption is made in her proofs that all loops are finite (i.e. will terminate) and hence commands and programs will have a final state. The following Z definition of the dynamic semantics of loops gives a potentially infinite recursion in the case where the expression evaluates to `TRUE` on each execution of the loop:

$$\left| \begin{array}{l} \mathcal{M}_C[\text{loop}(\epsilon, \gamma)]\rho\sigma = \\ \quad \text{if } \mathcal{M}_E[\epsilon]\rho\sigma = \text{bool}_v\text{T} \\ \quad \text{then } \mathcal{M}_C[\text{loop}(\epsilon, \gamma)]\rho(\mathcal{M}_C[\gamma]\rho\sigma) \\ \quad \text{else } \sigma \end{array} \right.$$

⁵This approach could also be used for out of range arithmetic.

To overcome this, we chose to implement a method similar to that described by Yu[36]. Yu used a counter which decremented on the execution of each 68020 instruction. This counter was initially set to *exactly* the necessary number of instructions required to perform the computation under consideration and would cause the NQTHM interpreter function to terminate at that point.

Our use is different, in that we do not have any knowledge of the particular algorithm being computed by the loop command, so we cannot assert any such upper bound on the counter. As such, our approach is closer to that used in some of the interpreter functions in the CLI stack work⁶.

The loop counter is introduced as part of the state at both the high and low-level. The counter is decremented every time the loop body is executed and when the counter reaches zero the loop is deemed to have reached a final state and is exited, even if the guard expression still evaluates to **TRUE**. Therefore we now have two methods of exiting from the loop: the guard expression evaluates to **FALSE**; or the loop counter reaches zero. Thus, the dynamic semantics of Tosca's loop become:

```

 $\mathcal{M}_C[\gamma : Cmd](\sigma : State) : RECURSIVE State =$ 
  IF Step( $\sigma$ ) = 0 THEN  $\sigma$  ELSE
    CASES  $\gamma$  OF
      ...
      loop( $\epsilon, \gamma$ ) : IF  $\mathcal{M}_E[\epsilon](\sigma) = BoolV(TRUE)$  THEN
        IF Step( $\mathcal{M}_C[\gamma](\sigma)$ ) = 0 THEN
           $\mathcal{M}_C[\gamma](\sigma)$ 
        ELSE
           $\mathcal{M}_C[loop(\epsilon, \gamma)](\mathcal{M}_C[\gamma](\sigma) WITH$ 
            [(Step) := Step( $\mathcal{M}_C[\gamma](\sigma)$ ) - 1])
          ENDIF
        ELSE
           $\sigma$ 
        ENDIF,
      ...
    ENDCASES
  ENDIF

```

⁶Noted by Young, in private communication (1998).

This new definition allows us to perform a ‘double-induction’ in order to prove that loops are correctly translated. The structural induction over the Tosca syntax gives us the induction hypothesis of the correct translation of the loop body and an additional lemma gives us the correctness of expression translation. A second induction is then performed over the loop counter to show that $\forall n : \mathbb{N}$, n executions of the loop give the same ‘result’ from both high and low-level languages.

This form of definition also allows the recursive function \mathcal{M}_C to be guaranteed to terminate. Non-terminating recursive functions are in essence partial functions and as such are not permitted in PVS as discussed earlier. Here we use the following **MEASURE** function to show that \mathcal{M}_C terminates:

$\mathit{MEASURE} \text{ sizeof}(\gamma) + \mathit{Step}(\sigma)$

sizeof is a function we have defined to give the ‘size’ of a Tosca command (i.e. the number of recursive calls required for a function to traverse an instance of that type). This decreases on the recursive call (for evaluating embedded commands) in every case except loop, where the ‘embedded’ command (when the loop ‘guard’ evaluates to **TRUE**) is not just the loop body but the *loop itself*, which has the same ‘size’. However, using the loop counter the *Step* decreases on each execution of the loop and hence the overall measure also decreases.

This approach is mirrored in the low-level language semantics with an augmentation to the machine state to include the step counter and a new instruction in a similar manner to that used to implement the stack pointer. As with the stack pointer the loop counter is merely a specification device and does not need to be reflected in the underlying hardware.

```

StateI : TYPE = [# StoreOfI : Env_StoreI,
                  InOfI : Input,
                  OutOfI : Output,
                  Step : nat #]

MI[[γ : Instr]](ρi : EnvI)(ϑ : Cont) : Cont =
  λ(σi : StateI) :
    IF Step(σi) = 0 ∨ ¬ stateok?(σi, γ, ρi) THEN
      σi
    ELSE
      CASES γ OF
        ...
        stepper : ϑ(σi WITH[(Step) := Step(σi) - 1])
      ENDCASES
    ENDIF

```

The **stepper** instruction is inserted after the code for the loop body by the compiler:

```

OC⟦ γ : Cmd ⟧(ρo : {ie : InjEnv | CmdOkay?(γ)(ie)})
  (SP : {l : Locn | l ≥ top(ρo)})
  (ϕ : Label) : RECURSIVE [Label, list[Instr]] =
  CASES γ OF
    ...
    loop(ε, γ1) : LET ϕ1 =
      proj1(OC[[γ1]](ρo)(SP)(ϕ)) IN
      (ϕ1 + 2, cons(label(ϕ1),
                  append(OE⟦ ε ⟧(ρo)(SP),
                        cons(jump(ϕ1 + 1),
                              append(proj2(OC⟦ γ1 ⟧(ρo)(SP)(ϕ)),
                                    cons(stepper,
                                          cons(goto(ϕ1),
                                                cons(label(ϕ1 + 1),
                                                      null))))))))),
      ...
  ENDCASES
MEASURE sizeof(γ)

```

6.2.6 Labels and Continuations

The original DCC semantics for Aida define an environment Env_I which maps from program labels to continuations. This is defined only in terms of an entire Aida program, viz:

$$\begin{array}{|l}
 \mathcal{M}_A[-] : AIDA_PROG \mapsto Input \mapsto Output \\
 \hline
 \exists \rho_\iota : Env_I; \vartheta_1, \vartheta_2, \vartheta_3, \dots, \vartheta_n : Cont \mid \\
 \rho_\iota = \{ \phi_1 \mapsto \vartheta_1, \phi_2 \mapsto \vartheta_2, \dots, \phi_n \mapsto \vartheta_n \} \\
 \wedge \vartheta_1 = \mathcal{M}_{I^*}[I_1]\rho_\iota\vartheta_2 \\
 \wedge \vartheta_2 = \mathcal{M}_{I^*}[I_2]\rho_\iota\vartheta_3 \\
 \wedge \dots \\
 \wedge \vartheta_n = \mathcal{M}_{I^*}[I_n]\rho_\iota(\text{id } State_I) \bullet \\
 \mathcal{M}_A[\text{Aida}(I_0 \hat{\ } \\
 \quad \langle \text{label } \phi_1 \rangle \hat{\ } I_1 \hat{\ } \\
 \quad \langle \text{label } \phi_2 \rangle \hat{\ } I_2 \hat{\ } \\
 \quad \dots \hat{\ } \\
 \quad \langle \text{label } \phi_n \rangle \hat{\ } I_n)]in = \\
 outOf_I(\mathcal{M}_{I^*}[I_0]\rho_\iota\vartheta_1(\emptyset, in, \langle \rangle))
 \end{array}$$

In the context of the proof of arbitrary Tosca command translation, we clearly do not have access to the entire Aida program text. The DCC approach does not concern itself with the correct construction of this environment, and we too treat it as an unproven assumption in our system.

Due to this, we have generated two axioms (for `loop` and `choice` instructions) which assert that this environment is correctly constructed with respect to the entire (unknown) object code of the program. These have been reviewed manually against the compiler specification but not proved. To do this would require a modification of the compiler to generate the Env_I environment as it passes over the source text, which we did not perform.

Loop_Env_I : AXIOM

$$\begin{aligned} & \rho_i(1 + PROJ_1(\mathcal{O}_C \downarrow \gamma \Downarrow (Env(StoreOf_I(\sigma_i)))(SP(StoreOf_I(\sigma_i)))(\phi)), \\ & \quad identity) \wedge \\ & \rho_i(PROJ_1(\mathcal{O}_C \downarrow \gamma \Downarrow (Env(StoreOf_I(\sigma_i)) \\ & \quad (SP(StoreOf_I(\sigma_i)))(\phi)), \\ & \quad \mathcal{M}_{I^*} \llbracket PROJ_2(\mathcal{O}_C \downarrow loop(\epsilon, \gamma) \Downarrow (Env(StoreOf_I(\sigma_i)) \\ & \quad (SP(StoreOf_I(\sigma_i)))(\phi)) \rrbracket (\rho_i)(identity)) \end{aligned}$$

Choice_Env_I : AXIOM

$$\begin{aligned} & LET \phi_1 : Label = PROJ_1(\mathcal{O}_C \downarrow \gamma_1 \Downarrow (Env(StoreOf_I(\sigma_i)) \\ & \quad (SP(StoreOf_I(\sigma_i)))(\phi)) IN \\ & \rho_i(PROJ_1(\mathcal{O}_C \downarrow \gamma_2 \Downarrow (Env(StoreOf_I(\sigma_i)))(SP(StoreOf_I(\sigma_i)) \\ & \quad (PROJ_1(\mathcal{O}_C \downarrow \gamma_1 \Downarrow (Env(StoreOf_I(\sigma_i)))(SP(StoreOf_I(\sigma_i)))(\phi))), \\ & \quad \mathcal{M}_{I^*} \llbracket PROJ_2(\mathcal{O}_C \downarrow \gamma_2 \Downarrow (Env(StoreOf_I(\sigma_i)) \\ & \quad (SP(StoreOf_I(\sigma_i)))(\phi)) \rrbracket (\rho_i)(\vartheta)) \wedge \\ & \rho_i(1 + PROJ_1(\mathcal{O}_C \downarrow \gamma_2 \Downarrow (Env(StoreOf_I(\sigma_i)) \\ & \quad (SP(StoreOf_I(\sigma_i)))(\phi_1)), \vartheta) \end{aligned}$$

6.3 Summary

This chapter has presented some of the key aspects of the translation of the DCC compiler from Z into the logic of the PVS system. We have seen how the strong type-system of PVS has allowed us to model in a natural manner the partial nature of the specification, and to extend our treatment to cover certain lacunae in the original development, such as termination.

Chapter 7

Mechanizing the DCC Proofs

This chapter presents the development of the mechanical proof of the DCC compiler, expressed in PVS. The aim of this part of the work was to develop a collection of proof strategies, which would allow the proof of the compiler to proceed in a more automatic manner, so relieving some of the burden commonly associated with mechanical proofs of correctness.

This is a very different aim from that of the DCC hand proofs. There, the proofs were structured in order for a human to read, review and comprehend. Human readers of proofs presented in what is commonly called ‘journal style’ tend to be intelligent, and apply their intuition selectively to follow complex steps. Mechanical provers have no such intelligence and intuition, and so we must present our proof in a very low level and step by step manner. This is the tedium referred to earlier, which we wish to address.

The proof of the translation of the Tosca assignment statement is again used to provide a worked example through this chapter. The other constructs of Tosca do not provide any extra insight into the problem, merely volume, and are not reported here.

7.1 Initial Proof Attempts

Following the initial direct (and clumsy) translation of the DCC Z specification into PVS, we attempted to prove the lemmas and theorems presented in Stepney’s book[3]. It was the attempted (and failed) proof of these theorems which in part led to the re-working of the compiler specification to

make it more reliant on heavily constrained types, and to correct inconsistencies, typographical errors and to make explicit the assumptions noted in the previous chapter.

In this initial stage, we noticed that the main branches of the correctness theorems followed closely the hand development. The exceptions to this noted where our augmentations came into play, and where the hand proof makes light of some tricky details in ‘leaps of faith’. The best examples of this are in the proof of the translation of `block` commands and entire programs, where the DCC presentation assumes that these will follow directly (and simply) from earlier lemmas, which is not entirely true. The correspondence between the hand proof and the main branch of the mechanical proof will be made explicit later in the next section.

The initial proof attempts were a very useful part of the process of translating the specification into PVS. As the theorem prover did not simply follow the hand proof and report Q.E.D., we were forced to carefully examine the precise behaviour of each function to determine the source of errors. This brought an insight and understanding that is difficult to achieve without attempting to prove conjectures about specifications.

7.2 Assignment

After integrating the knowledge gained from the first proof attempts back into the specification of the semantics, we were able to successfully perform the translation proofs within PVS. In this section, we step through the proofs involved in assignment translation in order to demonstrate:

- that proof ‘strategies’ can be used to automate the mundane parts of the proofs;
- that those proof strategies are reusable in the proofs of other constructs; and
- that due to the nature of the prover in PVS, the mechanical proofs are quite close to the level of the hand proofs.

Recall the lemma involved in the proof of assignment translation, as presented in Section 5.7 (Page 56). This is presented in PVS as follows:

Assignment_Lemma : **Lemma**

$\forall(\rho_i : Env_I, \phi : Label, \vartheta : Cont, \epsilon : Expr, \rho_t : Env_T, \xi : Name,$

$\sigma_i : \{s : State_I \mid Step(s) > 0\}, \sigma : State) :$

$Init_S(\sigma, \sigma_i) \wedge$

$Init_E(StoreOf(\sigma), Env(StoreOf_I(\sigma_i)), \rho_t) \wedge$

$CheckOk?(T_C \llbracket assign(\xi, \epsilon) \rrbracket(\rho_t))$

\Rightarrow

$$\begin{aligned} & \mathcal{M}_{I^*} \llbracket PROJ_2(\mathcal{O}_C \llbracket assign(\xi, \epsilon) \rrbracket(Env(StoreOf_I(\sigma_i))) \\ & \quad (SP(StoreOf_I(\sigma_i))(\phi)) \rrbracket(\rho_i)(\vartheta)(\sigma_i) = \\ & \vartheta(Update_{Store}(Update_A(\sigma_i)(\mathcal{M}_E \llbracket \epsilon \rrbracket(\sigma))) \\ & \quad (apply(map(Env(StoreOf_I(\sigma_i))), (\xi))) \\ & \quad (\mathcal{M}_E \llbracket \epsilon \rrbracket(\sigma))) \end{aligned}$$

The first two predicates ($Init_S$ and $Init_E$) are familiar from the hand proofs as the conditions which assert that the initial states and environments at high and low-level are equivalent. The third predicate on the left hand side of the implication asserts that the assignment statement has passed the type checking semantics (and thereby the declaration before use semantics).

If these pre-conditions are satisfied then the lemma should follow. The correspondence between the PVS specification of the lemma as presented here and the Z version (in Section 5.7) is straightforward, but the following points require some comment:

- the function *instrOf* is replaced by the PVS tuple projection operator, $PROJ_2$, which yields the second element from a tuple type (here the type is $[Label, Instr]$);
- replacing the ρ_o argument to \mathcal{O}_C by $Env(StoreOf_I(\sigma_i))$ and $SP(StoreOf_I(\sigma_i))$ (one of the consequences of combining the environment and store into one type);
- the use of $Update_{Store}$ in place of the \boxplus infix function; and
- the use of the function *apply* (from the functions as relations theory, in Appendix B) to apply the partial function *map*.

The proof of the lemma begins with the following sequent, which is the lemma as stated above¹:

Assignment_Lemma :

$$\begin{aligned}
& \{1\} \forall (\rho_i : Env_I, \phi : Label, \vartheta : Cont, \epsilon : Expr, \rho_t : Env_T, \xi : Name, \\
& \quad \sigma_i : \{s : State_I \mid Step(s) > 0\}, \sigma : State) : \\
& \quad Inits(\sigma, \sigma_i) \\
& \quad \wedge Init_E(StoreOf(\sigma), Env(StoreOf_I(\sigma_i)), \rho_t) \\
& \quad \wedge CheckOk?(TC[assign(\xi, \epsilon)](\rho_t)) \Rightarrow \\
& \quad \quad \mathcal{M}_{I^*}[PROJ_2(\mathcal{O}_C \langle assign(\xi, \epsilon) \rangle (Env(StoreOf_I(\sigma_i))) \\
& \quad \quad \quad (SP(StoreOf_I(\sigma_i))(\phi)))](\rho_i)(\vartheta)(\sigma_i) = \\
& \quad \quad \vartheta(Update_{Store}(Update_A(\sigma_i)(\mathcal{M}_E[\epsilon](\sigma))) \\
& \quad \quad \quad (apply(map(Env(StoreOf_I(\sigma_i)), (\xi))) \\
& \quad \quad \quad (\mathcal{M}_E[\epsilon](\sigma)))
\end{aligned}$$

We begin by setting up automatic rewriting for some of the abbreviation functions in the definition ($Update_{Store}$ and $Update_A$) and then skolemize the universal quantifier. Skolemized variables are represented with a prime symbol. For example, a universally quantified variable n would become n' when skolemized.

¹We recommend reading this proof whilst making reference to the hand proof presented in Section 5.7.

Installing automatic rewrites from:

$Update_{Store}$

$Update_A$

...

Repeatedly Skolemizing and flattening,
this simplifies to:

Assignment_Lemma :

$$\begin{array}{l}
 \{-1\} \text{ functional}(\rho'_i) \\
 \{-2\} \phi' \geq 0 \\
 \{-3\} \text{ functional}(\rho'_i) \\
 \{-4\} \text{ Step}(\sigma'_i) > 0 \\
 \{-5\} \text{ Init}_S(\sigma', \sigma'_i) \\
 \{-6\} \text{ Init}_E(\text{StoreOf}(\sigma'), \text{Env}(\text{StoreOf}_I(\sigma'_i)), \rho'_i) \\
 \{-7\} \text{ CheckOk?}(\mathcal{T}_C \llbracket \text{assign}(\xi', \epsilon') \rrbracket(\rho'_i)) \\
 \hline
 \{1\} \mathcal{M}_{I^*} \llbracket \text{PROJ}_2(\mathcal{O}_C \llbracket \text{assign}(\xi', \epsilon') \rrbracket(\text{Env}(\text{StoreOf}_I(\sigma'_i))) \\
 \quad \quad \quad (\text{SP}(\text{StoreOf}_I(\sigma'_i))(\phi')) \rrbracket(\rho'_i)(\vartheta')(\sigma'_i) = \\
 \quad \vartheta'(\text{Update}_{Store}(\text{Update}_A(\sigma'_i)(\mathcal{M}_E \llbracket \epsilon' \rrbracket(\sigma')))) \\
 \quad \quad (\text{apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi'))(\mathcal{M}_E \llbracket \epsilon' \rrbracket(\sigma')) \rrbracket
 \end{array}$$

We can now follow the hand proof quite closely, by expanding the definition of the compiler $(\mathcal{O}_C)^2$.

²From hereon in the proof, to save trees, we have stripped the sequent and present only the key parts. The omitted sections are marked with an ellipsis.

Expanding the definition of \mathcal{O}_C ,
 this simplifies to:
 Assignment_Lemma :

```

...
{1} IF  $\xi' \in \text{domain}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))))$ 
       $\wedge \text{ExprOk}?(e')(\text{Env}(\text{StoreOf}_I(\sigma'_i)))$  THEN
       $\mathcal{M}_{I\star}[\text{append}(\mathcal{O}_E \downarrow e' \uparrow)(\text{Env}(\text{StoreOf}_I(\sigma'_i)))(\text{SP}(\text{StoreOf}_I(\sigma'_i))),$ 
         $\text{cons}(\text{store}(\text{apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi')),$ 
           $\text{null})](\rho'_i)(\vartheta')(\sigma'_i)$ 
      ELSE
       $\mathcal{M}_{I\star}[\text{PROJ}_2(\text{DeadCode})](\rho'_i)(\vartheta')(\sigma'_i)$ 
    ENDF =
     $\vartheta'(\text{Update}_{\text{Store}}(\text{Update}_A(\sigma'_i)(\mathcal{M}_E[\epsilon'](\sigma')))$ 
       $(\text{apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi'))(\mathcal{M}_E[\epsilon'](\sigma')))$ 

```

The definition of \mathcal{O}_C has expanded to a conditional. The ‘then’ part is the operational semantics for assignment, which are only valid if the given variable is defined (i.e. is a member of the domain of map) and the expression is well defined. Otherwise, \mathcal{O}_C gives a bottom (DeadCode) as described in the previous chapter.

We now rewrite the ‘then’ part of Formula 1 using the definition of $\mathcal{M}_{I\star}$ (the Aida dynamic semantics) for appended lists, which yields four subgoals. We present the first subgoal here, which represents the main branch of the proof, and defer consideration of the others until later.

Rewriting using `ML_Star_Lem`,
 this yields 4 subgoals:
 Assignment_Lemma.1 :

<p style="margin: 0;">...</p> <hr style="width: 20%; margin-left: 0;"/> <p style="margin: 0;"> $\{1\}$ <i>IF</i> $\xi' \in \text{domain}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))))$ $\wedge \text{ExprOk}?(e')(\text{Env}(\text{StoreOf}_I(\sigma'_i)))$ <i>THEN</i> $\mathcal{M}_{I^*}[\mathcal{O}_E \langle e' \rangle (\text{Env}(\text{StoreOf}_I(\sigma'_i)))(\text{SP}(\text{StoreOf}_I(\sigma'_i)))](\rho'_i)$ $(\mathcal{M}_{I^*}[\text{cons}(\text{store}(\text{apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi')),$ $\text{null}]](\rho'_i)(\vartheta'))(\sigma'_i)$ <i>ELSE</i> $\mathcal{M}_{I^*}[\text{PROJ}_2(\text{DeadCode})](\rho'_i)(\vartheta')(\sigma'_i)$ <i>ENDIF</i> = $\vartheta'(\text{Update}_{\text{Store}}(\text{Update}_A(\sigma'_i)(\mathcal{M}_E[\langle e' \rangle](\sigma'))))$ $(\text{apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi'))(\mathcal{M}_E[\langle e' \rangle](\sigma'))$ </p>

The ‘then’ part of Formula 1 now tells us that assignment is performed by first evaluating the expression (compiled using \mathcal{O}_E) and then executing a continuation which stores the value of the expression at the location of variable ξ . We now need to introduce the (previously proven) lemma defining the correct compilation of expressions.

Using lemma `Expression_Correct`,
 this yields 3 subgoals:
`Assignment_Lemma.1.1` :

$$\begin{aligned}
 & \{-1\} \text{Init}_S(\sigma', \sigma'_i) \\
 & \wedge \text{Init}_E(\text{StoreOf}(\sigma'), \text{Env}(\text{StoreOf}_I(\sigma'_i)), \rho'_i) \\
 & \wedge \neg \text{TypeWrong}?(T_E[\epsilon'](\rho'_i)) \Rightarrow \\
 & \mathcal{M}_{I^*}[\mathcal{O}_E \langle \epsilon' \rangle (\text{Env}(\text{StoreOf}_I(\sigma'_i)))(SP(\text{StoreOf}_I(\sigma'_i)))](\rho'_i) \\
 & \quad (\mathcal{M}_{I^*}[\text{cons}(\text{store}(\text{apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi')), \\
 & \quad \quad \text{null}]](\rho'_i)(\vartheta'))(\sigma'_i) = \\
 & \mathcal{M}_{I^*}[\text{cons}(\text{store}(\text{apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi')), \text{null}]] \\
 & \quad (\rho'_i)(\vartheta')(Update_A(\sigma'_i)(\mathcal{M}_E[\epsilon'](\sigma')))) \\
 & \dots \\
 & \hline
 & [1] \text{IF } \xi' \in \text{domain}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i)))) \\
 & \quad \wedge \text{ExprOk}?(e')(\text{Env}(\text{StoreOf}_I(\sigma'_i))) \text{ THEN} \\
 & \quad \mathcal{M}_{I^*}[\mathcal{O}_E \langle \epsilon' \rangle (\text{Env}(\text{StoreOf}_I(\sigma'_i)))(SP(\text{StoreOf}_I(\sigma'_i)))](\rho'_i) \\
 & \quad \quad (\mathcal{M}_{I^*}[\text{cons}(\text{store}(\text{apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi')), \\
 & \quad \quad \quad \text{null}]](\rho'_i)(\vartheta'))(\sigma'_i) \\
 & \quad \text{ELSE} \\
 & \quad \mathcal{M}_{I^*}[\text{PROJ}_2(\text{DeadCode})](\rho'_i)(\vartheta')(\sigma'_i) \\
 & \quad \text{ENDIF} = \\
 & \vartheta'(Update_{\text{Store}}(Update_A(\sigma'_i)(\mathcal{M}_E[\epsilon'](\sigma')))) \\
 & \quad (\text{apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi'))(\mathcal{M}_E[\epsilon'](\sigma')))
 \end{aligned}$$

We now apply (`ground`) to perform propositional simplification, rewriting (in accordance with the auto-rewrite rules we specified) and simplification with decision procedures. This simplifies the lemma we have just introduced, and generates two subgoals. The first is the main branch of the proof, presented next. The other subgoal will be an obligation to discharge the left hand side of the implication of the expression lemma, which we will defer until later in the proof.

Applying propositional simplification and decision procedures,
this yields 2 subgoals:

Assignment_Lemma.1.1.1 :

$$\begin{aligned}
& \{-1\} \mathcal{M}_{I^*} \llbracket \mathcal{O}_E \llbracket \epsilon' \rrbracket (Env(StoreOf_I(\sigma'_i)))(SP(StoreOf_I(\sigma'_i))) \rrbracket (\rho'_i) \\
& \quad (\mathcal{M}_{I^*} \llbracket cons(store(apply(map(Env(StoreOf_I(\sigma'_i))), \xi')), \\
& \quad \quad null) \rrbracket (\rho'_i)(\vartheta'))(\sigma'_i) = \\
& \quad \mathcal{M}_{I^*} \llbracket cons(store(apply(map(Env(StoreOf_I(\sigma'_i))), \xi')), null) \rrbracket \\
& \quad (\rho'_i)(\vartheta')(\sigma'_i \text{ WITH } [(StoreOf_I) := StoreOf_I(\sigma'_i) \\
& \quad \quad \text{WITH } [(A) := \mathcal{M}_E \llbracket \epsilon' \rrbracket (\sigma')]]) \\
& \dots \\
& \{1\} \text{ IF } \xi' \in domain(map(Env(StoreOf_I(\sigma'_i)))) \\
& \quad \wedge ExprOk?(\epsilon')(Env(StoreOf_I(\sigma'_i))) \text{ THEN} \\
& \quad \quad \mathcal{M}_{I^*} \llbracket \mathcal{O}_E \llbracket \epsilon' \rrbracket (Env(StoreOf_I(\sigma'_i)))(SP(StoreOf_I(\sigma'_i))) \rrbracket (\rho'_i) \\
& \quad \quad (\mathcal{M}_{I^*} \llbracket cons(store(apply(map(Env(StoreOf_I(\sigma'_i))), \xi')), \\
& \quad \quad \quad null) \rrbracket (\rho'_i)(\vartheta'))(\sigma'_i) \\
& \quad \text{ELSE} \\
& \quad \quad \mathcal{M}_{I^*} \llbracket PROJ_2(DeadCode) \rrbracket (\rho'_i)(\vartheta')(\sigma'_i) \\
& \quad \text{ENDIF} = \\
& \quad \vartheta'(\sigma'_i \text{ WITH } [(StoreOf_I) := StoreOf_I(\sigma'_i) \\
& \quad \quad \text{WITH } [(A) := \mathcal{M}_E \llbracket \epsilon' \rrbracket (\sigma'), \\
& \quad \quad \quad (Mem) := Mem(StoreOf_I(\sigma'_i)) \\
& \quad \quad \quad \text{WITH } [(apply(map(Env(StoreOf_I(\sigma'_i))), \xi')) := \mathcal{M}_E \llbracket \epsilon' \rrbracket (\sigma')]])])
\end{aligned}$$

Now we replace the left hand side of Formula 1 with Formula -1 , thereby rewriting our current goal with respect to the expression correctness lemma.

Replacing using formula -1,
 this simplifies to:
 Assignment_Lemma.1.1.1 :

```

...
{1} IF  $\xi' \in \text{domain}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))))$ 
     $\wedge \text{ExprOk}?(e')(\text{Env}(\text{StoreOf}_I(\sigma'_i)))$  THEN
     $\mathcal{M}_{I^*}[\text{cons}(\text{store}(\text{apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi')),$ 
         $\text{null})](\rho'_i)(\vartheta')(\sigma'_i \text{ WITH}[(\text{StoreOf}_I) := \text{StoreOf}_I(\sigma'_i)$ 
         $\text{ WITH}[(A) := \mathcal{M}_E[e'](\sigma')])]$ 
    ELSE
     $\mathcal{M}_{I^*}[\text{PROJ}_2(\text{DeadCode})](\rho'_i)(\vartheta')(\sigma'_i)$ 
  ENDIF =
 $\vartheta'(\sigma'_i \text{ WITH}[(\text{StoreOf}_I) := \text{StoreOf}_I(\sigma'_i)$ 
     $\text{ WITH}[(A) := \mathcal{M}_E[e'](\sigma'),$ 
     $(\text{Mem}) := \text{Mem}(\text{StoreOf}_I(\sigma'_i))$ 
     $\text{ WITH}[(\text{apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi')) := \mathcal{M}_E[e'](\sigma')])]$ 

```

To simplify the goal, we use `(smash)`. This allows us to consider the ‘then’ and the ‘else’ case in isolation. We first consider the ‘then’ case, where the assignment statement is well-formed.

Repeatedly simplifying with BDDs, decision procedures, rewriting,
and if-lifting,
this yields 3 subgoals:
Assignment_Lemma.1.1.1.1 :

$\begin{array}{l} \dots \\ \{-8\} \xi' \in \text{domain}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i)))) \\ \{-9\} \text{ExprOk?}(\epsilon')(\text{Env}(\text{StoreOf}_I(\sigma'_i))) \end{array}$ <hr style="width: 80%; margin: 10px auto;"/> $\begin{array}{l} \{1\} \mathcal{M}_{I\star}[\text{cons}(\text{store}(\text{apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi')), \\ \quad \text{null})](\rho'_i)(\vartheta')(\sigma'_i \text{ WITH}[(\text{StoreOf}_I := \text{StoreOf}_I(\sigma'_i) \\ \quad \quad \quad \text{WITH}[(A := \mathcal{M}_E[\epsilon'](\sigma'))]) = \\ \quad \vartheta'(\sigma'_i \text{ WITH}[(\text{StoreOf}_I := \text{StoreOf}_I(\sigma'_i) \\ \quad \quad \quad \text{WITH}[(A := \mathcal{M}_E[\epsilon'](\sigma'), \\ \quad \quad \quad \quad (\text{Mem}) := \text{Mem}(\text{StoreOf}_I(\sigma'_i)) \\ \quad \quad \quad \text{WITH}[(\text{apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi')] := \mathcal{M}_E[\epsilon'](\sigma'))])])]) \end{array}$

The left hand side of Formula 1 now expresses the Aida semantics ($\mathcal{M}_{I\star}$) over a singleton list, generated by *cons*. We repeatedly expand the definition of $\mathcal{M}_{I\star}$, to reduce this to the Aida semantics for single instructions, \mathcal{M}_I .

Applying
 (REPEAT (EXPAND " \mathcal{M}_{I^*} ")),
 this simplifies to:
 Assignment_Lemma.1.1.1.1 :

...

$$\{1\} \mathcal{M}_I \llbracket store(apply(map(Env(StoreOf_I(\sigma'_i))), \xi')) \rrbracket(\rho'_i)$$

$$(\lambda(\sigma_i : State_I) : IF Step(\sigma_i) = 0 THEN$$

$$\quad \sigma_i$$

$$\quad ELSE$$

$$\quad \vartheta'(\sigma_i)$$

$$\quad ENDIF)(\sigma'_i WITH[(StoreOf_I) := StoreOf_I(\sigma'_i)$$

$$\quad WITH[(A) := \mathcal{M}_E \llbracket \epsilon' \rrbracket(\sigma')]) =$$

$$\vartheta'(\sigma'_i WITH[(StoreOf_I) := StoreOf_I(\sigma'_i)$$

$$\quad WITH[(A) := \mathcal{M}_E \llbracket \epsilon' \rrbracket(\sigma'),$$

$$\quad (Mem) := Mem(StoreOf_I(\sigma'_i))$$

$$\quad WITH[(apply(map(Env(StoreOf_I(\sigma'_i))), \xi')) := \mathcal{M}_E \llbracket \epsilon' \rrbracket(\sigma')])])$$

Having reduced Formula 1 to the application of \mathcal{M}_I , we now expand the definition of \mathcal{M}_I for the *store* instruction. This yields a rather complex goal as there are many conditions in the low-level semantics to ensure the instruction is applicable from the current state, and that the *Step* counter has not reached 0.

Expanding the definition of \mathcal{M}_I ,
 this simplifies to:
 Assignment_Lemma.1.1.1.1 :

```

...
{1} IF  $\neg \text{stateok?}(\sigma'_i \text{ WITH}[(\text{StoreOf}_I) := \text{StoreOf}_I(\sigma'_i)]$ 
       $\text{WITH}[(A) := \mathcal{M}_E[\epsilon'](\sigma')]$ ,
       $\text{store}(\text{apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi'), \rho'_i)$  THEN
   $\sigma'_i \text{ WITH}[(\text{StoreOf}_I) := \text{StoreOf}_I(\sigma'_i)]$ 
       $\text{WITH}[(A) := \mathcal{M}_E[\epsilon'](\sigma')]$ 
ELSE
  IF  $\text{Step}(\text{Update}_{\text{Store}}(\sigma'_i \text{ WITH}[(\text{StoreOf}_I) := \text{StoreOf}_I(\sigma'_i)]$ 
           $\text{WITH}[(A) := \mathcal{M}_E[\epsilon'](\sigma')]$ )
       $(\text{apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi')$ 
       $(\mathcal{M}_E[\epsilon'](\sigma'))) = 0$  THEN
     $\text{Update}_{\text{Store}}(\sigma'_i \text{ WITH}[(\text{StoreOf}_I) := \text{StoreOf}_I(\sigma'_i)]$ 
           $\text{WITH}[(A) := \mathcal{M}_E[\epsilon'](\sigma')]$ )
       $(\text{apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi'))(\mathcal{M}_E[\epsilon'](\sigma'))$ 
  ELSE
     $\vartheta'(\text{Update}_{\text{Store}}(\sigma'_i \text{ WITH}[(\text{StoreOf}_I) := \text{StoreOf}_I(\sigma'_i)]$ 
           $\text{WITH}[(A) := \mathcal{M}_E[\epsilon'](\sigma')]$ )
       $(\text{apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi'))$ 
       $(\mathcal{M}_E[\epsilon'](\sigma'))$ 
  ENDIF
ENDIF =
 $\vartheta'(\sigma'_i \text{ WITH}[(\text{StoreOf}_I) := \text{StoreOf}_I(\sigma'_i)]$ 
       $\text{WITH}[(A) := \mathcal{M}_E[\epsilon'](\sigma')$ ,
       $(\text{Mem}) := \text{Mem}(\text{StoreOf}_I(\sigma'_i))$ 
       $\text{WITH}[(\text{apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi')) := \mathcal{M}_E[\epsilon'](\sigma')]$ )

```

This sequent simplifies easily using (smash). There is no need to concern ourselves with the *Step* counter (as assignment does not alter it) and these conditions automatically simplify away. The equality is then true, on the condition that the *stateok?* condition is true.

Repeatedly simplifying with BDDs, decision procedures, rewriting,
and if-lifting,
this simplifies to:
Assignment_Lemma.1.1.1.1 :

...	$\{1\} \text{stateok?}(\sigma'_i \text{ WITH}[(\text{StoreOf}_I) := \text{StoreOf}_I(\sigma'_i) \\ \text{WITH}[(A) := \mathcal{M}_E[\epsilon'](\sigma')]], \\ \text{store}(\text{apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi'), \rho'_i)$
...	

This is where the hand proof ends, as it does not concern itself with the well formedness of low-level programs. We are not done yet in the PVS presentation, as we have are left to prove the obligation *stateok?*, the well formedness condition for \mathcal{M}_I . We begin by expanding the definition of *stateok?*.

Expanding the definition of *stateok?*,
this simplifies to:
Assignment_Lemma.1.1.1.1 :

...	$\{1\} \text{apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi') < SP(\text{StoreOf}_I(\sigma'_i))$
...	

The domain condition is that the application of the *map* function (which maps from variable names to store locations) must yield a location for the variable ξ which is below the Aida stack pointer in order for it to be a valid destination address for an assignment. Recalling how we defined the type of *map* in Section 6.2.1, this is a direct consequence of the predicate defining the function's range. We make this explicit to the theorem prover by asking for the type constraints for *map* to be added to the antecedent.

Adding type constraints for $map(Env(StoreOf_I(\sigma'_i)))$,
 this simplifies to:

Assignment_Lemma.1.1.1.1 :

$$\begin{array}{|l}
 \{-1\} \quad \forall(l : Locn) : l \in range(map(Env(StoreOf_I(\sigma'_i)))) \Rightarrow \\
 \qquad \qquad \qquad l < top(Env(StoreOf_I(\sigma'_i))) \\
 \{-2\} \quad part_{inj}(map(Env(StoreOf_I(\sigma'_i)))) \\
 \dots \\
 \hline
 [1] \quad apply(map(Env(StoreOf_I(\sigma'_i))), \xi') < SP(StoreOf_I(\sigma'_i)) \\
 \dots
 \end{array}$$

This goal is now proved by instantiating the quantifier in -1 with $apply(map(Env(StoreOf_I(\sigma'_i))), \xi')$ and then simple reasoning that top is less than SP from their definition in Section 6.2.1.

Instantiating the top quantifier in $-$ with the terms:

$apply(map(Env(StoreOf_I(\sigma'_i))), \xi')$,

this yields 3 subgoals:

Assignment_Lemma.1.1.1.1.1 :

$$\begin{array}{|l}
 \{-1\} \quad apply(map(Env(StoreOf_I(\sigma'_i))), \xi') \in range(map(Env(StoreOf_I(\sigma'_i)))) \Rightarrow \\
 \qquad \qquad \qquad apply(map(Env(StoreOf_I(\sigma'_i))), \xi') < top(Env(StoreOf_I(\sigma'_i))) \\
 \dots \\
 \hline
 [1] \quad apply(map(Env(StoreOf_I(\sigma'_i))), \xi') < SP(StoreOf_I(\sigma'_i)) \\
 \dots
 \end{array}$$

Now we apply propositional reasoning and the decision procedures, which reduces this goal to proving that applying the function $map(Env(StoreOf_I(\sigma'_i)))$ to the variable ξ yields a location which is in the range of the function.

Applying propositional simplification and decision procedures,
 this simplifies to:
 Assignment_Lemma.1.1.1.1.1 :

\dots <hr style="width: 80%; margin-left: 0;"/> $\{1\} \text{ apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi') \in \text{range}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))))$ \dots
--

The type of *apply* guarantees it will yield a member of the range type, if the argument is a member of the domain type. This goal can therefore be trivially proved by expanding the definition of \in .

Expanding the definition of \in ,
 this simplifies to:
 Assignment_Lemma.1.1.1.1.1 :

\dots <hr style="width: 80%; margin-left: 0;"/> $\{1\} \text{ TRUE}$ \dots
--

This completes the proof of Assignment_Lemma.1.1.1.1.1.

Having now proven the subgoals representing the main branch of the proof, we are presented with the postponed subgoals from earlier. The first is a TCC, generated as we instantiated the type constraint introduced earlier with $\text{apply}(\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i))), \xi')$. For this to be valid, ξ' must be in the domain of the function $\text{map}(\text{Env}(\text{StoreOf}_I(\sigma'_i)))$.

Assignment_Lemma.1.1.1.1.2 (TCC):

- | | |
|--|--|
| [-1] | $part_{inj}(map(Env(StoreOf_I(\sigma'_i))))$ |
| [-2] | $functional(\rho'_i)$ |
| [-3] | $\phi' \geq 0$ |
| [-4] | $functional(\rho'_t)$ |
| [-5] | $Step(\sigma'_i) > 0$ |
| [-6] | $Init_S(\sigma', \sigma'_i)$ |
| [-7] | $Init_E(StoreOf(\sigma'), Env(StoreOf_I(\sigma'_i)), \rho'_t)$ |
| [-8] | $CheckOk?(TC[assign(\xi', \epsilon')](\rho'_t))$ |
| [-9] | $\xi' \in domain(map(Env(StoreOf_I(\sigma'_i))))$ |
| [-10] | $ExprOk?(\epsilon')(Env(StoreOf_I(\sigma'_i)))$ |
| {1} $domain(map(Env(StoreOf_I(\sigma'_i))))(\xi')$ | |
| | ... |

We already know it is in the domain, by Formula -9, so we expand the definition of \in .

Expanding the definition of \in ,
 this simplifies to:
 Assignment_Lemma.1.1.1.1.2 :

- | | |
|--|--|
| | ... |
| {-9} | $domain(map(Env(StoreOf_I(\sigma'_i))))(\xi')$ |
| | ... |
| {1} $domain(map(Env(StoreOf_I(\sigma'_i))))(\xi')$ | |
| | ... |

which is trivially true.

This completes the proof of Assignment_Lemma.1.1.1.1.2.

A further TCC was generated from the same instantiation. This obliges us to show that the function $map(Env(StoreOf_I(\sigma'_i)))$ is *functional*, i.e. obeys the type constraints specified in the functions as relations specification (given in Appendix B).

Assignment_Lemma.1.1.1.1.3 (TCC):

- | | |
|--|--|
| [-1] | $part_{inj}(map(Env(StoreOf_I(\sigma'_i))))$ |
| [-2] | $functional(\rho'_i)$ |
| [-3] | $\phi' \geq 0$ |
| [-4] | $functional(\rho'_t)$ |
| [-5] | $Step(\sigma'_i) > 0$ |
| [-6] | $Init_S(\sigma', \sigma'_i)$ |
| [-7] | $Init_E(StoreOf(\sigma'), Env(StoreOf_I(\sigma'_i)), \rho'_t)$ |
| [-8] | $CheckOk?(TC[assign(\xi', \epsilon')](\rho'_t))$ |
| [-9] | $\xi' \in domain(map(Env(StoreOf_I(\sigma'_i))))$ |
| [-10] | $ExprOk?(\epsilon')(Env(StoreOf_I(\sigma'_i)))$ |
| {1} $functional(map(Env(StoreOf_I(\sigma'_i))))$ | |
| | ... |

We already know from Formula -1 that it is a partial injection ($part_{inj}$), which is defined as $functional \wedge injective$. Therefore we expand the definition of $part_{inj}$:

Expanding the definition of $part_{inj}$,
this simplifies to:

Assignment_Lemma.1.1.1.1.3 :

- | | |
|--|--|
| {-1} | $functional(map(Env(StoreOf_I(\sigma'_i))))$
$\wedge injective(map(Env(StoreOf_I(\sigma'_i))))$ |
| | ... |
| [1] $functional(map(Env(StoreOf_I(\sigma'_i)))) \dots$ | |

Applying propositional simplification and decision procedures,

This completes the proof of Assignment_Lemma.1.1.1.1.3.

This completes the proof of Assignment_Lemma.1.1.1.1.

For the next postponed subgoal, we return quite a way back up the proof 'tree' to where we used the lemma about expression translation. This has a precondition $ExprOk?$, which we are now obliged to discharge.

Assignment_Lemma.1.1.1.2 :

- | |
|--|
| <p>[−1] $functional(\rho'_i)$
 [−2] $\phi' \geq 0$
 [−3] $functional(\rho'_i)$
 [−4] $Step(\sigma'_i) > 0$
 [−5] $Init_S(\sigma', \sigma'_i)$
 [−6] $Init_E(StoreOf(\sigma'), Env(StoreOf_I(\sigma'_i)), \rho'_t)$
 [−7] $CheckOk?(\mathcal{T}_C[assign(\xi', \epsilon')](\rho'_i))$</p> <hr style="border: 0.5px solid black;"/> <p>{1} $ExprOk?(\epsilon')(Env(StoreOf_I(\sigma'_i)))$
 ...</p> |
|--|

ExprOk? ensures that all variables referenced in an expression are valid, i.e. are in the domain of the store. As our assignment statement has passed the static semantic checks, this will be true. The proof of this condition is therefore by simple expansion of the static semantic function \mathcal{T}_C , the initial environment hypothesis and propositional reasoning.

Expanding the definition of \mathcal{T}_C and $Init_E$,
 this simplifies to: Assignment_Lemma.1.1.1.2 :

```

...
{-6} Env(StoreOf( $\sigma'$ )) = Env(StoreOfI( $\sigma'_i$ ))
       $\wedge$  domain(map(Env(StoreOfI( $\sigma'_i$ )))) = domain( $\rho'_t$ )
       $\wedge$  ( $\forall(\xi : Name) : domain(\rho'_t)(\xi) \Rightarrow$ 
            $\neg$  TypeWrong?(apply( $\rho'_t, \xi$ )))
{-7} CheckOk?(IF  $\xi' \in domain(\rho'_t)$ 
               $\wedge$  apply( $\rho'_t, \xi'$ ) =  $\mathcal{T}_E[\epsilon'](\rho'_t)$ 
               $\wedge$   $\neg$  TypeWrong?(apply( $\rho'_t, \xi'$ )) THEN
              CheckOk
              ELSE
              CheckWrong
              ENDIF)
-----
[1] ExprOk?( $\epsilon'$ )(Env(StoreOfI( $\sigma'_i$ )))
...

```

Repeatedly simplifying with BDDs, decision procedures, rewriting,
 and if-lifting,
 This completes the proof of Assignment_Lemma.1.1.1.2.

The second domain condition from the application of the expression lemma
 is that ξ is in the domain of function $map(Env(StoreOf_I(σ'_i)))$. We have seen
 the proof of this before, so we will take this as read and move onto the next
 subgoal.

Assignment_Lemma.1.1.1.3 :

- [-1] $functional(\rho'_i)$
- [-2] $\phi' \geq 0$
- [-3] $functional(\rho'_t)$
- [-4] $Step(\sigma'_i) > 0$
- [-5] $Init_S(\sigma', \sigma'_i)$
- [-6] $Init_E(StoreOf(\sigma'), Env(StoreOf_I(\sigma'_i)), \rho'_t)$
- [-7] $CheckOk?(\mathcal{T}_C[assign(\xi', \epsilon')](\rho'_t))$

{1} $\xi' \in domain(map(Env(StoreOf_I(\sigma'_i))))$

...

We have seen a proof of this before, so we will take this as read and move onto the next subgoal. This next subgoal, also a condition from the expression lemma, is that the result of type checking the expression ϵ' is not *TypeWrong?*. Note that as this is a negative goal, it appears in the antecedent.

Assignment_Lemma.1.1.2 :

- {-1} $TypeWrong?(\mathcal{T}_E[\epsilon'])(\rho'_t)$
- [-2] $functional(\rho'_i)$
- [-3] $\phi' \geq 0$
- [-4] $functional(\rho'_t)$
- [-5] $Step(\sigma'_i) > 0$
- [-6] $Init_S(\sigma', \sigma'_i)$
- [-7] $Init_E(StoreOf(\sigma'), Env(StoreOf_I(\sigma'_i)), \rho'_t)$
- [-8] $CheckOk?(\mathcal{T}_C[assign(\xi', \epsilon')](\rho'_t))$

...

As the assignment statement has passed type checking, then the expression must also have done so. Hence, this subgoal is proved by appealing to the static semantics again.

Expanding the definition of \mathcal{T}_C ,
 this simplifies to:
 Assignment_Lemma.1.1.2 :

[-1] <i>Type Wrong?</i> ($\mathcal{T}_E[\epsilon'](\rho'_t)$) ... {-8} <i>CheckOk?</i> (<i>IF</i> $\xi' \in \text{domain}(\rho'_t)$ $\wedge \text{apply}(\rho'_t, \xi') = \mathcal{T}_E[\epsilon'](\rho'_t)$ $\wedge \neg \text{Type Wrong?}(\text{apply}(\rho'_t, \xi'))$ <i>THEN</i> <i>CheckOk</i> <i>ELSE</i> <i>CheckWrong</i> <i>ENDIF</i>)	
...	

Repeatedly simplifying with BDDs, decision procedures, rewriting,
 and if-lifting,
 This completes the proof of Assignment_Lemma.1.1.2.
 This completes the proof of Assignment_Lemma.1.1.

All of the remaining subgoals in this proof are familiar. We now present
 them in turn, without details of their proofs.

Assignment_Lemma.1.2 (TCC):

[-1] <i>functional</i> (ρ'_i) [-2] $\phi' \geq 0$ [-3] <i>functional</i> (ρ'_t) [-4] <i>Step</i> (σ'_i) > 0 [-5] <i>Init_S</i> (σ', σ'_i) [-6] <i>Init_E</i> (<i>StoreOf</i> (σ'), <i>Env</i> (<i>StoreOf_I</i> (σ'_i)), ρ'_t) [-7] <i>CheckOk?</i> ($\mathcal{T}_C[\text{assign}(\xi', \epsilon')](\rho'_t)$)	
{1} <i>domain</i> (<i>map</i> (<i>Env</i> (<i>StoreOf_I</i> (σ'_i))))(ξ') ...	

Assignment_Lemma.1.3 (TCC):

- [−1] $functional(\rho'_i)$
 - [−2] $\phi' \geq 0$
 - [−3] $functional(\rho'_i)$
 - [−4] $Step(\sigma'_i) > 0$
 - [−5] $Init_S(\sigma', \sigma'_i)$
 - [−6] $Init_E(StoreOf(\sigma'), Env(StoreOf_I(\sigma'_i)), \rho'_i)$
 - [−7] $CheckOk?(TC[assign(\xi', \epsilon')](\rho'_i))$
-
- {1} $functional(map(Env(StoreOf_I(\sigma'_i))))$
 - ...

Assignment_Lemma.2 :

- [−1] $functional(\rho'_i)$
 - [−2] $\phi' \geq 0$
 - [−3] $functional(\rho'_i)$
 - [−4] $Step(\sigma'_i) > 0$
 - [−5] $Init_S(\sigma', \sigma'_i)$
 - [−6] $Init_E(StoreOf(\sigma'), Env(StoreOf_I(\sigma'_i)), \rho'_i)$
 - [−7] $CheckOk?(TC[assign(\xi', \epsilon')](\rho'_i))$
-
- {1} $domain(map(Env(StoreOf_I(\sigma'_i))))(\xi')$
 - ...

Assignment_Lemma.3 :

- [−1] *functional*(ρ'_i)
 - [−2] $\phi' \geq 0$
 - [−3] *functional*(ρ'_t)
 - [−4] *Step*(σ'_i) > 0
 - [−5] *Init_S*(σ', σ'_i)
 - [−6] *Init_E*(*StoreOf*(σ'), *Env*(*StoreOf_I*(σ'_i)), ρ'_t)
 - [−7] *CheckOk?*(\mathcal{T}_C [*assign*(ξ', ϵ')](ρ'_t))
-
- {1} *functional*(*map*(*Env*(*StoreOf_I*(σ'_i))))

Assignment_Lemma.4 :

- [−1] *functional*(ρ'_i)
 - [−2] $\phi' \geq 0$
 - [−3] *functional*(ρ'_t)
 - [−4] *Step*(σ'_i) > 0
 - [−5] *Init_S*(σ', σ'_i)
 - [−6] *Init_E*(*StoreOf*(σ'), *Env*(*StoreOf_I*(σ'_i)), ρ'_t)
 - [−7] *CheckOk?*(\mathcal{T}_C [*assign*(ξ', ϵ')](ρ'_t))
-
- {1} *ExprOk?*(ϵ')(*Env*(*StoreOf_I*(σ'_i)))
- ...

Q.E.D.

Run time = 36.87 secs.

Real time = 51.81 secs.

It is difficult to comprehend this style of presentation, so to allow the reader to better understand the overall structure of the proof of this lemma, we have reproduced the PVS proof tree in Figure 7.1³. In this display, the

³The diagram is merely intended to allow comparison with the later, simpler, proof tree and as such it is not necessary to be able to read the labels representing the proof steps.

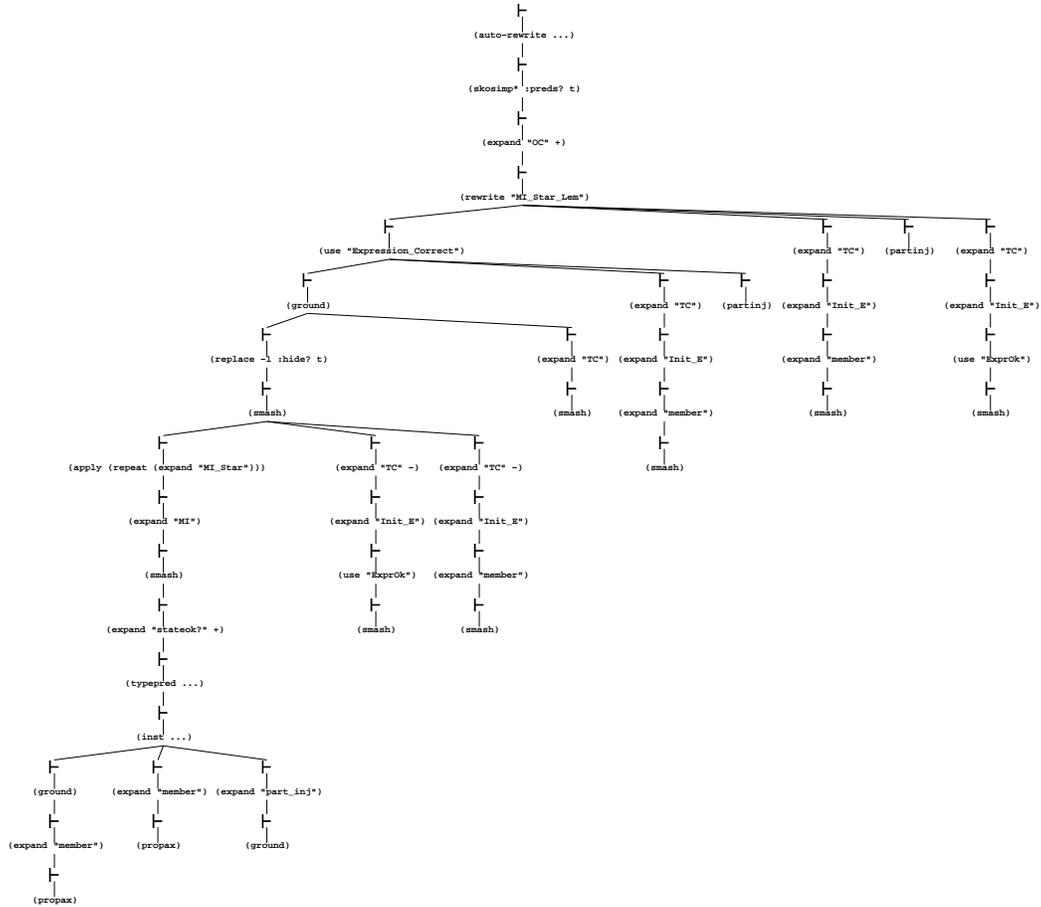


Figure 7.1: Structure of Assignment Lemma Proof

turnstiles represent proof states, with the commands used shown in the state transitions.

7.3 Initial Strategy Development

The left hand branch at each point in the proof display (Figure 7.1) shows the main branch of the proof. We can see that the PVS proof follows quite closely the hand development of the DCC proof shown in section 5.7. In this example, the divergence from the hand proof is at points where our domain

conditions must be discharged. None of these require particularly complex proofs, and their proofs are either very similar, or identical. This is evidence of the particular suitability of the tool chosen to keep the proof at a reasonable level of abstraction — automating mundane tasks like simple rewriting, and propositional simplification, and allowing the user to concentrate on directing the key steps of the proof

As we noted each particular instance of a domain condition in the proofs, we wrote a strategy to automatically discharge it. These began life as proof ‘macros’ — i.e. shorthand for the particular steps required so that the repetitive nature of their proof was reduced to recognizing which type of domain condition we were required to discharge and using the appropriate strategy.

For example, in the proof of the assignment lemma we have just walked through, there were three particular types of domain condition to discharge:

1. that *stateok?* (the domain condition for \mathcal{M}_I) holds;
2. that *ExprOk?* (the domain condition for \mathcal{O}_E) holds; and
3. that *apply* can be used for *part_{inj}* relations as well as *functional* relations.

These are all proofs about cases which follow ‘directly’ from our preconditions in the assignment lemma, in that they are generated on instantiation, rewriting and the expansion of the definitions appearing in the original lemma, so it is unsurprising that they are straightforward.

Based on the proof of these conditions in the assignment lemma, we generated three strategies for the domain conditions, as shown below. They are in essence just macros, but we will see in the next section how as the development of the DCC compiler proofs in PVS continued, they evolved into more intelligent strategies which were able to tackle a wider range of domain conditions.

```

(defstep stateok ()
  (then (expand "stateok?" +
    (typepred "map(Env(StoreOf_I(sigma_i!1)))")
    (inst - "apply(map(Env(StoreOf_I(sigma_i!1))), xi!1)")
    (grind))
    "Prove domain condition for MI holds"
    "~%Proving domain condition for MI")

(defstep exprok ()
  (then (expand* "TC" "Init_E" "member")
    (use "Exprok")
    (smash))
    "Prove the domain condition for OE"
    "~%Proving the domain condition for OE")

(defstep partinj ()
  (then (typepred "map(Env(StoreOf_I(sigma_i!1)))")
    (expand "part_inj")
    (ground))
    "Prove that a partial injection is functional"
    "~%Proving that part_inj implies functional")

```

Using these strategies, the proof tree shown in Figure 7.1 becomes the simpler one shown in Figure 7.2.

7.4 Limitation of the Strategy Approach

The previous section demonstrated how we could use the proof strategy mechanism of PVS to produce what are essentially proof macros that simplified that *particular* proof. But of what *general* use do these macros contribute? That is the issue addressed in this section.

The approach used was to find all instances of related domain conditions in the original compiler proofs. The strategies were applied to each of these in turn and where they failed modifications were made so as they could prove the original class of provable goals, and also the new case.

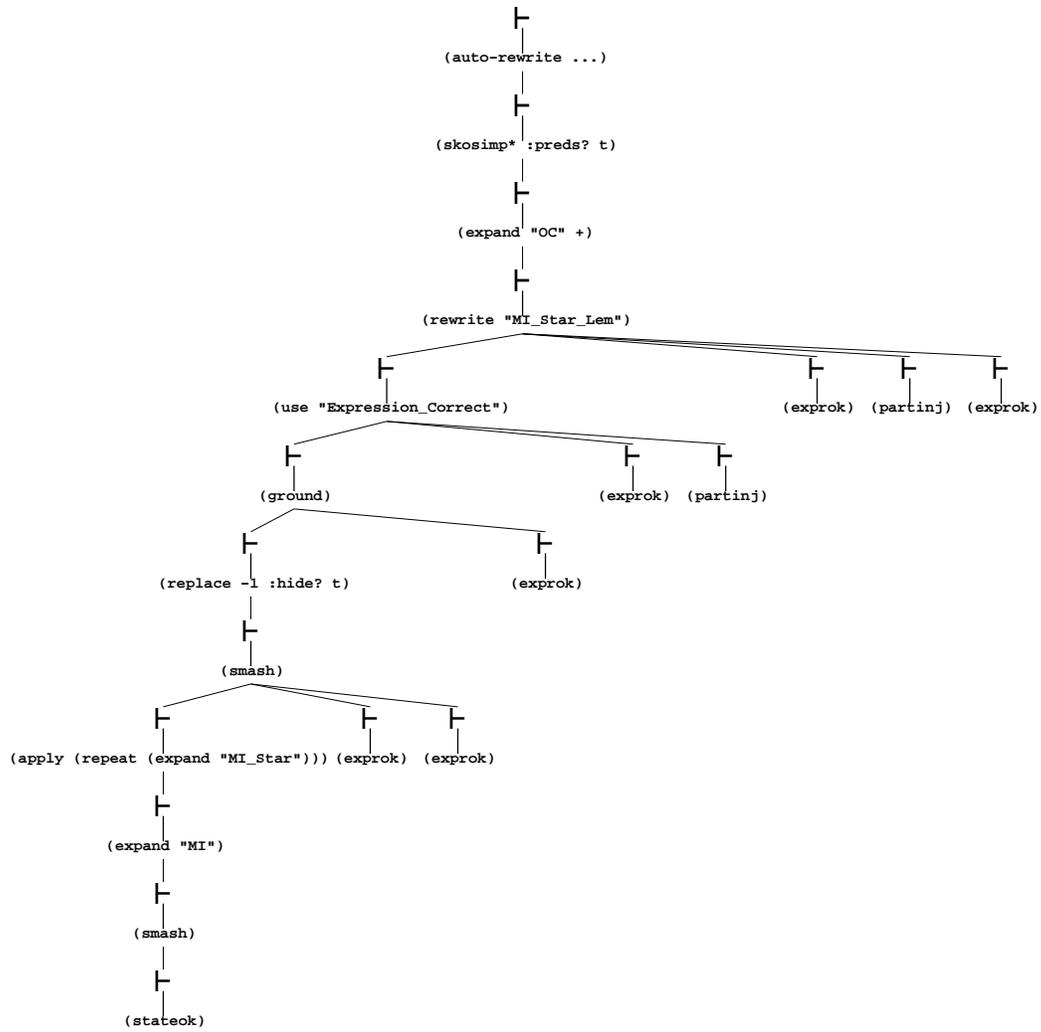


Figure 7.2: Structure of Assignment Lemma Proof Using Strategies

We failed to develop effective strategies for tackling the main branches of the correctness proofs for several reasons:

- the semantic functions make bad rewrite rules;
- automatic rewriting tended to be over eager; and
- the heuristic instantiation in PVS could not generate the required matches.

The first two points are related, and not specific to the PVS system. Automatic rewriting performs very well on specifications where you can provide rewrite rules that converge to a ‘normal form’. In the DCC specification, we were more concerned with ensuring traceability back to the original semantics as specified in Z. Hence, we did not perform too many alterations to the semantics in order to make them into ‘good’ rewrite rules. The third point is specific to the PVS system, and is discussed further in chapter 11.

These are *not* fundamental limitations of the approach, but specific to the style of specification used here. Evidence from other application domains[128] has shown that careful structuring of rewrite rules can allow rewriting to complete large proofs, in co-operation with decision procedures.

7.5 Are Strategies Necessary?

In this chapter we have shown how we have used the proof strategy mechanism of PVS to provide what are essentially re-usable proof macros for the domain conditions and other simple conditions generated in our proofs. We have shown how they were developed from the collapsing of the proof steps involved in a particular goal, and then generalized so they could be used on a class of similar goals.

There is another approach to this problem — that of using lemmata. For example, we have proof strategies which generate the necessary proofs of domain conditions given certain assumptions in the hypothesis. They could easily be stated using a lemma, i.e. *assumptions* \Rightarrow *domain condition*.

Using lemmata can be just as powerful as proof strategies. Moore was able to re-target his verification of the Piton assembler (part of the CLI stack, reported in section 3.4.2) to a different processor and reprove the

entire assembler in less than two weeks, due to the use of libraries of useful general-purpose lemmata that allowed the proof to be fully-automatically ‘discovered’ by NQTHM⁴.

This approach would (in our example) not work as well as strategies, as the instantiation in PVS does not generate the correct matches in many cases.

7.6 Summary

In this chapter we have presented a worked example of the proof of Tosca’s assignment statement within the framework developed in the previous chapter. PVS aided us by automating mundane steps (such as simple rewriting), and the resultant mechanical proof was quite similar to the hand proof. The exception to this is where domain conditions required proof, for example on instantiation or rewriting. We have shown how PVS strategies were initially used as macros to allow easy proof of these constructs, and how by application to other occurrences, they have been modified and augmented so they can cope with other types of condition.

In the next chapter, we take our armory of strategies and our exposition of what is ‘good’ proof style and tackle extensions to the original Tosca language.

7.7 An Aside — Proof Presentation

As an aside, the presentation of mechanical proofs is still very much a research problem of itself. Abstracting the detail to just the right level for a journal style presentation is currently a human art (exhibited in this chapter for instance).

Proof sequents tend to contain a lot of formulas which are not relevant to the goal under consideration. Here we stripped them out, leaving just the important formulas needed in the current proof step. This process is challenging enough for human users of proof tools, so it is not surprising that it is difficult for the tool to provide an effective solution.

⁴Noted by Young, in private communication, 1998.

Current research into this area (such as that presented in [129, 130, 131, 132, 133, 134, 135]) concentrates on this problem from the starting point of low level proof assistants, so that they must combine mundane steps in the mechanical proof in order to present them to the reader in a higher-level ‘intuitive’ style. This is a different problem to that which is experienced in a system like PVS where there is significant automation, as the system may automatically prove a goal which in a journal style would be presented in several steps.

Part III

Extending Tosca

Chapter 8

Language Extension — Local Scope

This part of the thesis describes the direct addition of extra constructs to the high-level language Tosca. The purpose of this exercise was three-fold:

1. to examine how resilient the existing proofs were to augmentation and changes in the semantics;
2. to examine the performance of the proof strategies for domain conditions we had developed on new examples; and
3. to examine the relationship between new constructs in Tosca and the increase in specification and proof work required.

The constructs chosen were local scope, parameterless procedures and call-by-value parameters for procedures. These are useful structures for any programming language, but their inclusion here was motivated by the following concerns:

1. modest scale.
2. the addition of local scope would create interesting problems in the constraints in the specification, for example, that the environment mapping from variable names to locations was static (as we had only global allocation).

3. that procedures would give experience in adding an extra type to Tosca, and that it would also result in complications about allowing an effective control ‘jump’ in the high-level language that was not originally available.

These particular constructs were chosen over any others as we felt they were the most important features of ‘practical’ programming languages that were missing from the original Tosca language.

The remainder of this chapter describes the augmentations in turn, relating:

- the modifications required to the specification;
- the changes required to proofs of existing constructs; and
- an outline of the correct translation proof for the new construct.

All of the constructs require modification of the Tosca syntax data type, Tosca’s semantic functions, and the operational semantics (the compiler specification).

Local scope

The first augmentation to the source language was to add local scope, similar to the `declare` block of Ada[16]. Local scope is a useful software engineering technique to restrict the use of variables to their intended purpose, and to allow for the use of temporary variables in procedures and functions.

The syntax of Tosca was updated to include a new construct `local`, which allows any Tosca construct to be executed in the presence of local variables¹:

```

Cmd : DATATYPE
BEGIN
  skip : skip?
  block(cs : (cons?[Cmd])) : block?
  assign(i : Name, e : Expr) : assign?
  choice(c : Expr, t, e : Cmd) : choice?
  loop(e : Expr, c : Cmd) : loop?
  input(i : Name) : input?
  output(e : Expr) : output?
  local(d : Decl, c : Cmd) : local?
END Cmd

```

The main change involved as a result of this is to the specification of the memory allocation routines. A language with local scope is, functionally, equivalent to a language with merely global (or no) scope — this is apparent from the ability to translate scoped high-level language programs into unscoped low-level assembly languages and to attain a functionally equivalent result from the low-level implementation.

This section concentrates on the implementation of local scope in the usual way, by assigning local variables space on the stack at the low-level.

Note that the addition of local scope still preserves the criteria for a safety critical language[7] that the memory utilization be statically decidable. In this case, the memory required is the sum of that required by the global allocations, and the ‘largest’ block. The largest block in the memory utilization sense can only be calculated by examining each in turn, and summing the memory required for temporary variables on the stack during expression evaluation (one per binary operator) and the number of local variables. Of

¹A useful point for those extending developments in PVS using abstract data types is always to add new constructs *last* in the ADT. Subgoals generated from structural induction on the datatype are generated in the order in which they appear in the type definition. Thus, by adding the new construct last, when replaying old proofs the original subproofs will be matched to the correct subgoal. Perhaps this is an obvious comment, but one that is important in proof reusability.

course, this calculation becomes a little more complex in the case where different types of Tosca variables have differing storage requirements, but this is currently not the case.

8.1 Tosca Semantics

8.1.1 Declaration Before Use Semantics

The declaration before use semantics of local scope are trivial. The embedded command is checked in an environment which is extended using the given declarations. This requires identifiers for variables in δ that are unique, not only within δ but also relative to the existing declarations in scope from global declarations, and enclosing local scope constructs.

```

 $\mathcal{D}_C \llbracket \gamma : Cmd \rrbracket (\rho_d : Env_D) : RECURSIVE Check =$ 
  CASES  $\gamma$  OF
    ...
    local( $\delta, \gamma_1$ ) :
       $\mathcal{D}_C \llbracket \gamma_1 \rrbracket (\mathcal{D}_{D^*} \llbracket \delta \rrbracket (\rho_d))$ 
  ENDCASES
  MEASURE sizeof( $\gamma$ )

```

8.1.2 Type Checking Semantics

The type checking semantics involve the use of an auxiliary function *DeclOk?* to check that the declarations are unique. This is implied by it having passed the declaration before use semantics, but by using the auxiliary function we do not have to pass the declaration before use environment (Env_D) as an argument to \mathcal{T}_C .

```
DeclOk?( $\delta : Decl$ )( $\rho_t : Env_T$ ) : bool =  
   $\forall (b : BasicDecl) : b \in \delta \Rightarrow$   
     $\neg vi(b) \in domain(\rho_t)$   
  
 $\mathcal{T}_C[\gamma : Cmd](\rho_t : Env_T) : RECURSIVE\ Check =$   
  CASES  $\gamma$  OF  
    ...  
    local( $\delta, \gamma_1$ ) :  
      IF DeclOk?( $\delta$ )( $\rho_t$ ) THEN  
         $\mathcal{T}_C[\gamma_1](\mathcal{T}_{D^*}[\delta](\rho_t))$   
      ELSE  
        CheckWrong  
      ENDIF  
  ENDCASES  
MEASURE sizeof( $\gamma$ )
```

8.1.3 Dynamic Semantics

The major change involved in adding local scope to Tosca was in the allocation of variables to store locations. In original Tosca, this was quite simple — the global allocation of variables meant that the environment was static throughout the execution of the program. Here, we wished to keep as close to the original allocation scheme as possible, to continue to take advantage of the work we had already done on specifying the constraints between environment and store in the PVS datatypes.

At the Tosca level, we made no changes to the datatypes involved in the representation of environment and store. We defined the dynamic semantics of local scope as follows:

```

 $\mathcal{M}_C[\gamma : Cmd](\sigma : State) : RECURSIVE State =$ 
  IF Step( $\sigma$ ) = 0 THEN  $\sigma$  ELSE
    CASES  $\gamma$  OF
      ...
      local( $\delta, \gamma_1$ ) :
        LET  $\sigma_2 : State = \mathcal{M}_C[\gamma_1](\sigma$  WITH [(StoreOf) :=
           $\mathcal{M}_{D^*}[\delta](StoreOf(\sigma))$ ) IN
          IF Step( $\sigma_2$ )  $\neq$  0 THEN
            DeAlloc( $\sigma_2$ )(StoreOf( $\sigma$ ))
          ELSE
             $\sigma_2$ 
          ENDIF
        ENDCASES
      ENDIF
    MEASURE sizeof( $\gamma$ ) + Step( $\sigma$ )

```

The above fragment shows that local scope is implemented by a recursive call to the dynamic semantics (\mathcal{M}_C), but within an environment containing the declarations from δ .

After the recursive call to \mathcal{M}_C has been made, the *Step* value is checked and if zero, the machine is effectively halted (as described in the previous chapter) and so we return just the state given by the recursive call to \mathcal{M}_C . If the *Step* is non zero, we must remove the locally allocated variables from the environment, here by the use of *DeAlloc*:

$$DeAlloc(\sigma : State)(\rho : Env_{Store}) : State = \\ \sigma \text{ WITH } [(StoreOf) := StoreOf(\sigma) \text{ WITH } [(Env) := Env(\rho)]]$$

DeAlloc returns the environment of the Tosca program to exactly what it was before the local block was entered. Note that due to the Tosca memory function being dependent on the environment, this automatically ‘shrinks’ the memory area under consideration to what it was before.

This approach is appealing due to its simplicity. Standard techniques for implementing scope rules use elaborate mechanisms such as scope stacks to determine which variable is being referenced in any given instance, but our assumption that the parser will give variables unique names allows for brevity (and thereby clarity) in the semantics.

8.2 Compiler

To compile local scope, we need to generate Aida code which implements the given Tosca command, within an environment extended with the local variable declarations. Intuitively, this involves a recursive call to the operational semantics of commands (the compilation function, \mathcal{O}_C) within an environment generated by the operational semantics of declarations (the compilation function $\mathcal{O}_{D\star}$). The new instructions (*local* and *unlocal*) are described in the next section.

```

 $\mathcal{O}_C \langle \gamma \rangle (\rho_o : \text{Inj}_{Env})(SP : \{l : \text{Locn} \mid l \geq \text{top}(\rho_o)\})$ 
  ( $\phi : \text{Label}$ ) : RECURSIVE [Label, list[Instr]] =
  CASES  $\gamma$  OF
    ...
    local( $\delta$ ,  $\gamma_1$ ) :
      IF  $\forall (d : \text{BasicDecl}) : d \in \delta \Rightarrow \neg vi(d) \in \text{domain}(\text{map}(\rho_o))$  THEN
        LET  $\rho_{o2} : \text{Inj}_{Env} = \mathcal{O}_{D^*} \langle \delta \rangle (\rho_o)$ ,
           $code_1 : \text{Code} = \mathcal{O}_C \langle \gamma_1 \rangle (\rho_{o2})(\text{top}(\rho_{o2}))(\phi)$  IN
            (proj1(code1), cons(local( $\rho_{o2}$ ),
              append(proj2(code1),
                cons(unlocal( $\rho_o$ ),
                  null))))))
      ELSE
        DeadCode
      ENDIF
    ENDCASES
  MEASURE sizeof( $\gamma$ )

```

The recursive call to \mathcal{O}_C is made within the extended environment as described above, but there is an interesting use of $\text{top}(\rho_{o2})$ (the highest allocated location in the new environment) as the new stack pointer.

In Aida code generated from the Tosca compiler, the only use of the stack is for the storage of temporary values during binary expression translation. When we are at the beginning of a code segment generated from \mathcal{O}_C , we can guarantee that top will be equal to SP , as commands are never embedded in expressions. Thus, we do not need to worry about the previous value of the stack pointer and can simply move it to after the newly allocated local variables without any problems.

8.3 Aida Semantics

A useful side effect of implementing the storage allocation in the low level machine this way is that we do not break any of the constraints previously specified on the combined environment and store datatypes:

$$\begin{aligned} \text{Inj}_{Env} : \text{TYPE} = & \\ & [\# \text{top} : \text{Locn}, \\ & \text{map} : \{m : (\text{part}_{inj}[\text{Name}, \text{Locn}]) \mid \\ & \quad \forall (l : \text{Locn}) : (l \in \text{range}(m)) \Rightarrow l < \text{top}\} \#] \\ \\ \text{Env_Store}_I : \text{TYPE} = & [\# \text{Env} : \text{Inj}_{Env}, \\ & \text{A} : \text{Value}, \\ & \text{SP} : \{l : \text{Locn} \mid l \geq \text{top}(\text{Env})\}, \\ & \text{Mem} : [\{l : \text{Locn} \mid l < \text{SP}\} \rightarrow \text{Value}] \#] \end{aligned}$$

However, by running the proofs we discovered we have with this approach the same problem as with the introduction of the stack pointer in that we need pseudo-instructions in the low-level machine to allocate and deallocate the variables as the environment and store are combined in the state. We take the same approach to its solution, and add new instructions into the low-level semantics that create and destroy the mappings necessary for the local variables. The unlocal instruction assigns its argument to *top*, and constrains the environment and memory functions appropriately so that the local variables are thereby deallocated.

As with *spinc*, *spdec*, and *stepper* these instructions are merely a specification device and need not be reflected in the underlying hardware (and should be removed from the Aida code by the assembler). Their definition is as follows:

```

 $\mathcal{M}_I[\gamma : Instr](\rho_i : Env_I)(\vartheta : Cont) : Cont =$ 
   $\lambda(\sigma_i : State_I) :$ 
    IF  $Step(\sigma_i) = 0 \vee \neg stateok?(\sigma_i, \gamma, \rho_i)$  THEN
       $\sigma_i$ 
    ELSE
      CASES  $\gamma$  OF
        ...
        local( $\rho$ ) :
           $\vartheta(Alloc(\sigma_i)(\rho))$ ,
        unlocal( $\rho$ ) :
           $\vartheta(DeAlloc(\sigma_i)(\rho))$ 
      ENDCASES
    ENDIF

```

The actual allocation and de-allocation of the variables is performed by the functions *Alloc* and *DeAlloc* that have the following definition:

```

 $Alloc(\sigma_i : State_I)(\rho : Inj_{Env}) : State_I =$ 
   $\sigma_i$  WITH [(StoreOf $_I$ ) := (# Env :=  $\rho$ ,
     $A := A(StoreOf_I(\sigma_i))$ ,
     $SP := top(\rho)$ ,
     $Mem := \lambda(loc : \{l : Locn \mid l < top(\rho)\}) :$ 
      IF  $loc < SP(StoreOf_I(\sigma_i))$  THEN
        ( $Mem(StoreOf_I(\sigma_i))(loc)$ )
      ELSE
         $IntV(0)$ 
      ENDIF#)]

```

$$\begin{aligned}
& DeAlloc(\sigma_i : State_I) \\
& (\rho : \{e : Inj_{Env} \mid top(e) \leq top(Env(StoreOf_I(\sigma_i)))\}) : State_I = \\
& \sigma_i \text{ WITH } [(StoreOf_I) := (\# Env := \rho, \\
& \qquad A := A(StoreOf_I(\sigma_i)), \\
& \qquad SP := top(\rho), \\
& \qquad Mem := restrict(Mem(StoreOf_I(\sigma_i))) \#)]
\end{aligned}$$

The function *Alloc* uses the given environment ρ and extends the domain of the memory to include the newly allocated variables. They are all initialized to the integer value 0. The particular value is not significant as we perform initialization before use checks in the static semantics, but as we lack a don't care value, 0 is used.

DeAlloc uses the approach described earlier of setting the stack pointer *SP* to the old value of *top*, and use the PVS *restrict* operator to shrink the domain of *Mem* appropriately to locations below *top*.

8.4 Proof

The augmentations to the specification in order to add local scope did not affect any of the basic types (other than the *Cmd* ADT). Hence, the previous correctness proofs were unaffected and were re-run without difficulty.

The only proof ‘juggling’ required was in the generated TCCs from the augmented specifications — they are named according the order in which they are generated, so some previously valid proofs became attached to the wrong proof obligation.

The proof of local scope did not present any complications, and follows directly from the proof of declarations and induction hypothesis for the embedded command. The proof strategies for domain conditions did not require alteration.

8.5 Summary

This chapter has presented the extension of Tosca to include the local scope of variables. We have presented the static and dynamic semantics of local scope and described the modifications required to the existing data types and functions.

Chapter 9

Language Extension — Parameterless Procedures

Procedures are a fundamental part of all programming languages. They allow us to re-use sections of code without the need to replicate them, and also provide a natural method for expressing the algorithmic solution to many problems.

In the semantics of Tosca, we need to add constructs for the definition of procedures (which may well occur within a local scope block), and constructs to allow procedures to be called. In order to simplify the task, we start with parameterless procedures.

9.1 Tosca Semantics

Firstly, we add a new type *TypeProc*:

$$ToscaType : TYPE = \{TypeInteger, TypeBoolean, TypeProc, TypeWrong\}$$

Adding *TypeProc* to the *ToscaType* definition means that we can declare procedures wherever we could declare any other Tosca variable. Because of the constraints on declarations, it has the side effect of prohibiting overloading — two procedures cannot have the same name, and a procedure cannot have the same name as a variable. This is not an overly restrictive burden, and some would argue is preferable in a safety critical programming language.

The previous semantics for declarations do not allow for the association of an object with a declaration. To remove this problem, we could alter the definition of *ToscaType* to be:

```

ToscaType : DATATYPE
BEGIN
  TypeInteger : TypeInteger?
  TypeBoolean : TypeBoolean?
  TypeProc(c : Cmd) : TypeProc?
  TypeWrong : TypeWrong?
END ToscaType

```

However, this leads to the *ToscaType* declaration referencing *Cmd*, and vice-versa. Such mutual recursion between data types is not supported in PVS, but is emulated by providing ADTs with subtypes. This removes the need for mutual recursion (in our case), by giving us a datatype which is the combination of *Decl* and *Cmd*. We thus use the following combination of types:

```

ToscaType : TYPE = {TypeInteger, TypeBoolean, TypeProc, TypeWrong}

DeclCmd : DATATYPE WITH SUBTYPES Cmd, BasicDecl, Decl
BEGIN
  declvar(vi : Name, t : ToscaType) : declvar? : BasicDecl
  declproc(pi : Name, c : Cmd) : declproc? : BasicDecl
  decls(d : list[BasicDecl]) : decls? : Decl
  skip : skip? : Cmd
  block(cs : (cons?[Cmd])) : block? : Cmd
  assign(ai : Name, e : Expr) : assign? : Cmd
  choice(c : Expr, t, e : Cmd) : choice? : Cmd
  loop(e : Expr, c : Cmd) : loop? : Cmd
  input(ii : Name) : input? : Cmd
  output(e : Expr) : output? : Cmd
  local(d : Decl, c : Cmd) : local? : Cmd
  callproc(n : Name) : callproc? : Cmd
END DeclCmd

```

This will be familiar from previous PVS datatype declarations presented in Chapter 4, with the exception of a subtype membership note for each member of the datatype.

For declarations, we now have a type *Decl*, whose constructor generates a list of *BasicDecl*. *BasicDecl* are either *declvar* or *declproc*, distinguishing between the declaration of variables and procedures. Variable declarations, as before, relate a name to a type which is constrained in the static semantics not to be *TypeWrong* or *Typeproc*. Procedure declarations relate a name to a Tosca command, most usefully a *block* or *local* command.

The commands are much as before, but with the addition of *callproc*, which invokes the command which has been related (by declaration) to the given procedure name.

9.1.1 Declaration Before Use Semantics

The declaration before use semantics of procedure declarations provide a check to ensure that the procedure is not called recursively¹. The semantic function \mathcal{D}_C returns a tuple $[Env_D, Check]$, where Env_D is a mapping from variable names to $Check$, and the $Check$ part of the tuple is the overall check state of commands.

```

 $\mathcal{D}_C[\gamma : DeclCmd](\rho_d : Env_D) : RECURSIVE [Env_D, Check] =$ 
  CASES  $\gamma$  OF
    ...
    declproc( $\xi, \gamma_1$ ) :
      LET  $\rho_{d1} : Env_D =$  IF  $\xi \in domain(\rho_d)$  THEN
        replace( $\rho_d, \xi, CheckWrong$ )
      ELSE
        add( $(\xi, CheckOk), \rho_d$ )
      ENDIF IN
      ( $\rho_{d1}, proj_2(\mathcal{D}_C[\gamma_1](replace(\rho_{d1}, \xi, CheckWrong)))$ ),
    ...
    callproc( $\xi$ ) :
      ( $\rho_d$ , IF  $\xi \in domain(\rho_d)$  THEN
        CheckOk
      ELSE
        CheckWrong
      ENDIF)
  ENDCASES
  MEASURE sizeof( $\gamma$ )

```

The body of the procedure is checked *at the point of declaration*, and as our variable names are all unique, it is therefore safe to execute the procedure in the dynamic environment *at the point of call* in the knowledge that it will not interfere with variables that were not in scope at its declaration.

The check for non-recursion is implemented by ensuring that during the checking of the procedure body, the environment Env_D maps the procedure name to $CheckWrong$, and so any attempt in the procedure body to call itself will result in the body failing the declaration before use check.

¹Recursion is often prohibited in safety critical developments[7].

The inclusion of this check for recursion is not an issue in the correctness of the translation, as we have no limits to our store usage in the proof. If recursion was to be admitted, this check could be removed without affecting the validity of the translation proof, by substituting ρ_{d1} for the second occurrence of $replace(\rho_{d1}, \xi, CheckWrong)$ in the above definition.

The declaration before use semantics are simple and just check to see if the identifier has been declared. It may not be the identifier of a procedure, but this is caught by the type checking semantics.

9.1.2 Type Checking Semantics

Procedure declarations are well typed if the identifier has not previously been declared, and the body of the procedure is well typed (according to the type environment at the point of declaration). The same comment about declaration time versus call time checking applies as for declaration before use checking.

```

 $\mathcal{T}_C[\gamma : DeclCmd](\rho_t : Env_T) : RECURSIVE [Env_T, Check] =$ 
  CASES  $\gamma$  OF
    ...
    declproc( $\xi, \gamma_1$ ) :
      IF  $\xi \in domain(\rho_t)$  THEN
        ( $\rho_t, CheckWrong$ )
      ELSE
        LET  $\rho_{t1} : Env_T = add((\xi, TypeProc), \rho_t)$  IN
          ( $\rho_{t1}, proj_2(\mathcal{T}_C[\gamma_1](\rho_{t1}))$ )
        ENDIF,
    ...
  ENDCASES
  MEASURE sizeof( $\gamma$ )

```

A procedure call is well typed if the variable name given references something of *TypeProc*.

```

 $\mathcal{T}_C[\gamma : DeclCmd](\rho_t : Env_T) : RECURSIVE [Env_T, Check] =$ 
  CASES  $\gamma$  OF
    ...
    callproc( $\xi$ ) :
      ( $\rho_t, IF \xi \in domain(\rho_t) \wedge TypeProc?(apply(\rho_t, \xi)) THEN$ 
        CheckOk
      ELSE
        CheckWrong
      ENDIF
    ENDCASES
  MEASURE sizeof( $\gamma$ )

```

9.1.3 Dynamic Semantics

To describe the semantics of procedures at the Tosca level, we first require a new environment function which maps from procedure names to procedure bodies. This environment is setup at the point the procedure is declared, either globally or inside a local block.

```

 $Env_P : TYPE = (functional[Name, Cmd])$ 

 $Initial\_Env_P : Env_P =$ 
  emptyset

```

Env_P is a partial function from procedure names to commands that is built up during the elaboration of procedure declarations. We first check that the identifier is not already declared as a procedure (i.e. is in the domain of ρ_p) or a variable (i.e. is in the domain of $map(Env(StoreOf(\sigma)))$).

```

 $\mathcal{M}_C[\gamma : DeclCmd](\sigma : State) : RECURSIVE State =$ 
  LET  $\rho_p : Env_P = \rho_p(\sigma)$  IN
  CASES  $\gamma$  OF
    ...
    declproc( $\xi, \gamma_1$ ) :
      IF  $\xi \in domain(\rho_p) \vee$ 
         $\xi \in domain(map(Env(StoreOf(\sigma))))$  THEN
         $\sigma$ 
      ELSE
         $\sigma$  WITH  $[(\rho_p) := add((\xi, \gamma_1), \rho_p)]$ 
      ENDIF,
    ...
  ENDCASES
  MEASURE sizeof( $\gamma$ ) + Step( $\sigma$ )

```

Procedure calls therefore apply the function ρ_p to yield the appropriate procedure body, and execute that in the current state.

```

 $\mathcal{M}_C[\gamma : DeclCmd](\sigma : State) : RECURSIVE State =$ 
  LET  $\rho_p : Env_P = \rho_p(\sigma)$  IN
  CASES  $\gamma$  OF
    ...
    callproc( $\xi$ ) :
      IF  $\xi \in domain(\rho_p) \wedge \neg Step(\sigma) = 0$  THEN
         $\mathcal{M}_C[apply(\rho_p, \xi)](\sigma)$ 
      ELSE
         $\sigma$ 
      ENDIF
    ...
  ENDCASES
  MEASURE sizeof( $\gamma$ ) + Step( $\sigma$ )

```

9.2 Compiler

In a similar manner to the Tosca semantics, the compiler needs a mapping between procedure names and the label which indicates the starting location of the Aida code that implements the procedure body. This role is taken by Env_{OP} :

$$Env_{OP} : TYPE = (part_{inj}[Name, Label])$$
$$Initial_Env_{OP} : Env_{OP} = \\ null_{inj}$$

This is an injection as we do not admit aliasing of procedures, in the same manner as we do not admit aliasing of stored variables.

As we are only considering parameterless procedures, the only information we need to pass to the called procedure is the appropriate return address. As procedures may themselves in turn call procedures we cannot use a static location to store the return address and so we place it on the top of the stack.

The compiled code for each procedure needs to be prefixed with a program label, to allow the procedure to be jumped to and some code to store the return label on the stack (it is stored in the accumulator at call time). A suffix is required to jump back to the stored label, and reset the stack. This cleanup operation is performed by a new low level machine instruction, *rts*, which is described in the next section.

```

 $\mathcal{O}_C \langle \gamma : DeclCmd \rangle (\rho_o : Inj_{Env}) (SP : \{l : Locn \mid l \geq top(\rho_o)\}) (\phi : Label)$ 
 $(\rho_{op} : Env_{OP}) : RECURSIVE Code =$ 
CASES  $\gamma$  OF
...
declproc( $\xi, \gamma_1$ ) :
  IF  $\xi \in domain(\rho_{op}) \vee \xi \in domain(map(\rho_o))$  THEN
    DeadCode
  ELSE
    LET  $code : Code = \mathcal{O}_C \langle \gamma_1 \rangle (\rho_o) (SP + 1) (\phi + 1)$ 
       $(add((\xi, \phi), \rho_{op}))$  IN
       $(proj_1(code), cons(label(\phi),$ 
         $cons(spinc,$ 
           $cons(store(SP),$ 
             $append(proj_2(code),$ 
               $cons(rts, null))))),$ 
         $\rho_o, add((\xi, \phi), \rho_{op}))$ 
    ENDIF,
...
callproc( $\xi$ ) :
  IF  $\xi \in domain(\rho_{op})$  THEN
     $(\phi + 1, cons(loadConst(LabelV(\phi)),$ 
       $cons(goto(apply(\rho_{op}, \xi)),$ 
         $cons(label(\phi), null))),$ 
       $\rho_o, \rho_{op})$ 
  ELSE
    DeadCode
  ENDIF
ENDCASES
MEASURE  $sizeof(\gamma)$ 

```

To allow program labels to be stored in the accumulator and in memory (on the stack), it was necessary to change the definition of *Value* — the storable values of the low-level machine.

```

Value : DATATYPE
BEGIN
  IntV(i : ToscaInteger) : IntV?
  BoolV(b : boolean) : BoolV?
  LabelV(l : Label) : LabelV?
ENDValue

```

A consequence of this change is that the semantics of expressions need to check that arithmetic and logical operators are not applied to labels in addition to their current checking.

9.3 Aida Semantics

We noted in the previous section that to perform the return from procedure operation, we have introduced a new low-level instruction, *rts*. This performs a jump to the program label on the top of the stack, using the current mapping from labels to continuations (ρ_i). It also decrements the stack pointer in the current state.

```

 $\mathcal{M}_I \llbracket \gamma : Instr \rrbracket (\rho_i : Env_I)(\vartheta : Cont) : Cont =$ 
 $\lambda(\sigma_i : State_I) :$ 
  IF Step( $\sigma_i$ ) = 0  $\vee$   $\neg$  stateok?( $\sigma_i, \gamma, \rho_i$ ) THEN
     $\sigma_i$ 
  ELSE
    CASES  $\gamma$  OF
      ...
      rts :
        apply( $\rho_i, l((Mem(StoreOf_I(\sigma_i)))(SP(StoreOf_I(\sigma_i))))$ )
        ( $\sigma_i$  WITH[(StoreOf_I) := SP_DEC(StoreOf_I( $\sigma_i$ ))])
    ENDCASES
  ENDIF

```

9.4 Proof

Existing proofs required a certain amount of modification after the addition of procedures. This was due to the combination of *Decl* and *Cmd* into one ADT, thus combining the proofs of declaration translation and command translation into one (as cases in the structural induction over *DeclCmd*). This did not create much difficulty and in some cases simplified the proofs, as the correct translation of declarations was available in the induction hypothesis instead of requiring a lemma.

The proof of procedure translation is performed in two parts — declaration and call. The procedure call proof is performed under the assumption (from the static semantics) that the procedure has been well declared and the environment maps the procedure identifier to the correct body at high and low-level. The only complex part is the manipulation of the return label and jumping to and from the code, but the proof was otherwise simple.

9.5 Summary

This chapter has presented the addition of parameterless procedures to Tosca. We have presented the static and dynamic semantics for the declaration and calling of the procedures and described the modifications required to the existing data types and functions.

Incidentally, the addition of procedures to Tosca has led to the augmentation of the specification language of PVS itself in order to support the mutual recursion between datatypes.

Chapter 10

Language Extension — Procedure Parameters

This chapter describes the addition of procedures with parameters to Tosca. The approach taken here is a different one to that of the previous two chapters, in that we already have in Tosca all of the necessary constructs to *emulate* procedures with parameters, so the semantics of parameters are described in terms of existing Tosca constructs. This lessens the proof work required, as the transformation is merely a syntactic transformation and can reuse previous proofs as lemmata¹.

The approach taken here assumes a call by value semantics: arguments to a procedure call are expressions which are evaluated before calling the procedure, and passed as values in local variables. This is emulated by enclosing a call to a parameterless procedure with a local scope block which defines a local variable for each of the procedure's formal parameters, and an assignment statement to evaluate the actual parameter and assign it to the defined local variable. To illustrate this, consider the following procedure declaration of a procedure ξ , whose body is γ and has one argument α of type *ToscaInteger*:

```
declproc( $\xi$ ,  $\gamma$ , cons( $(\alpha$ , ToscaInteger), null))
```

¹A similar approach is used by Low-Ada[136] — a proposal to separate concerns of syntactic sugar from the truly difficult parts of compilation.

To call this procedure with argument 12²:

```
callproc( $\xi$ , cons(Constant(12), null))
```

This call is equivalent to the following Tosca code, which can be implemented without adding any new features to the compiler above those described in the previous chapter:

```
local(cons(declvar( $\alpha$ , ToscaInteger), null),  
      block(cons(assign( $\alpha$ , Constant(12)),  
            npproc( $\xi$ ))))
```

This is the approach taken in this chapter. We now describe the additions to the datatypes defining the abstract syntax, and then the additions to the semantics and the compiler.

10.1 Tosca Semantics

The formal arguments to procedures form part of their type, so we change the definition of *ToscaType* from the previous enumeration to the following abstract datatype:

```
ToscaType : DATATYPE  
BEGIN  
  TypeInteger : TypeInteger?  
  TypeBoolean : TypeBoolean?  
  TypeProc(f : list[ToscaType]) : TypeProc?  
  TypeWrong : TypeWrong?  
END ToscaType
```

This allows us to associate a list of parameter types with each procedure. As we are assuming that we require procedure parameter actuals to occur in the same order as the formals, there is no need for us to store the parameter names.

²We assume that the actual parameters must occur in the same order as the formal parameters. Alternative arrangements could be simply dealt with in the parser.

The abstract syntax of commands is augmented with constructs for the declaration and calling of procedures with parameters:

```

DeclCmd : DATATYPE WITH SUBTYPES Cmd, BasicDecl, Decl
BEGIN
  declvar(vi : Name, t : ToscaType) : declvar? : BasicDecl
  declproc(pi : Name, c : Cmd,
           f : list[(declvar?)]) : declproc? : BasicDecl
  decls(d : list[BasicDecl]) : decls? : Decl
  skip : skip? : Cmd
  block(cs : (cons?[Cmd])) : block? : Cmd
  assign(ai : Name, e : Expr) : assign? : Cmd
  choice(c : Expr, t, e : Cmd) : choice? : Cmd
  loop(e : Expr, c : Cmd) : loop? : Cmd
  input(ii : Name) : input? : Cmd
  output(e : Expr) : output? : Cmd
  local(d : Decl, c : Cmd) : local? : Cmd
  npproc(n : Name) : npproc? : Cmd
  callproc(n : Name, a : list[Expr]) : callproc? : Cmd
ENDDeclCmd

```

Note that we have removed the ability to directly declare parameterless procedures, other than by having an empty list of formal parameters in `declproc`. However, we retain the ability to call parameterless procedures with the command `npproc(n)`. This is not intended to be part of the public syntax of Tosca at this point, merely a target for the compilation of parameterless procedures.

10.1.1 Declaration Before Use Semantics

The declaration before use semantics are very similar to those of procedures without parameters. The difference is the need to check the formal and actual parameters to the procedure.

```

 $\mathcal{D}_C[\gamma : DeclCmd](\rho_d : Env_D) : RECURSIVE [Env_D, Check] =$ 
  CASES  $\gamma$  OF
    ...
    declproc( $\xi, \gamma_1, formals$ ) :
      LET  $\rho_{d1} : Env_D =$  IF  $\xi \in domain(\rho_d)$  THEN
        replace( $\rho_d, \xi, CheckWrong$ )
      ELSE
        add( $(\xi, CheckOk), \rho_d$ )
      ENDIF IN
      ( $\rho_{d1}, PROJ_2(\mathcal{D}_C[\gamma_1](proj_1(\mathcal{D}_C[decls(formals)](\rho_{d1}))))$ ),
    ...
    callproc( $\xi, E$ ) :
      ( $\rho_d,$  IF  $\xi \in domain(\rho_d) \wedge$ 
        ( $\forall(\epsilon : Expr) : \epsilon \in E \Rightarrow CheckOk?(\mathcal{D}_E[\epsilon](\rho_d))$ ) THEN
        CheckOk
      ELSE
        CheckWrong
      ENDIF)
  ENDCASES
  MEASURE sizeof( $\gamma$ )

```

10.1.2 Type Checking Semantics

The main complication introduced into the Tosca semantics is in type checking. Procedure declarations involve the checking of the procedure body, within an environment including the type definitions of its formal parameters. The function *GetTypes* is used to extract a list of types from the list of variable declarations (defining the formal parameters of the procedure).

```

GetTypes(formals : list[(declvar?)]) : RECURSIVE list[ToscaType] =
  IF null?(formals) THEN
    null
  ELSE
    cons(t(car(formals)), GetTypes(cdr(formals)))
  ENDIF
MEASURE length

 $\mathcal{T}_C[\gamma : DeclCmd](\rho_t : Env_T) : RECURSIVE [Env_T, Check] =$ 
  CASES  $\gamma$  OF
    ...
    declproc( $\xi$ ,  $\gamma_1$ , formals) :
      IF  $\xi \in domain(\rho_t)$  THEN
        ( $\rho_t$ , CheckWrong)
      ELSE
        LET  $\rho_{t1} : [Env_T, Check] = \mathcal{T}_C[\llbracket decls(formals) \rrbracket](\rho_t)$  IN
          IF CheckOk?(proj2( $\rho_{t1}$ )) THEN
            (add(( $\xi$ , TypeProc(GetTypes(formals))),  $\rho_t$ ),
            proj2( $\mathcal{T}_C[\llbracket \gamma_1 \rrbracket](\text{proj}_1(\rho_{t1}))$ ))
          ELSE
            ( $\rho_t$ , CheckWrong)
          ENDIF
      ENDIF,
    ...
  ENDCASES
MEASURE sizeof( $\gamma$ )

```

Checking of procedure calls merely requires that the given name is of *TypeProc* and that the types of the actual parameters match those of the

formal parameters. The actual matching of types is performed by simple recursion over the lists by *ArgsOk?*.

```

ArgsOk?(E : list[Expr])(T : list[ToscaType])( $\rho_t$  : EnvT) : RECURSIVE bool =
  IFnull?(E) THEN
    null?(T)
  ELSE
    car(T) = TE[[car(E)]]( $\rho_t$ )  $\wedge$  ArgsOk?(cdr(E))(cdr(T))( $\rho_t$ )
  ENDIF
MEASURE length(E)

TC[[ $\gamma$  : DeclCmd]]( $\rho_t$  : EnvT) : RECURSIVE [EnvT, Check] =
  CASES  $\gamma$  OF
    ...
    callproc( $\xi$ , E) :
      ( $\rho_t$ , IF  $\xi \in \text{domain}(\rho_t) \wedge$ 
        TypeProc?(apply( $\rho_t$ ,  $\xi$ ))  $\wedge$ 
        ArgsOk?(E)(f(apply( $\rho_t$ ,  $\xi$ )))( $\rho_t$ ) THEN
          CheckOk
        ELSE
          CheckWrong
        ENDIF)
    ...
  ENDCASES
MEASURE sizeof( $\gamma$ )

```

10.1.3 Dynamic Semantics

The dynamic semantics of procedure declarations requires the modification of the environment Env_P (mapping from procedure names to commands) to include the formal parameters.

```

EnvP : TYPE = (functional[Name, [list[(declvar?)], Cmd]])

 $\mathcal{M}_C[\gamma : DeclCmd](\sigma : State) : RECURSIVE State =$ 
  LET  $\rho_p : Env_P = \rho_p(\sigma)$  IN
  CASES  $\gamma$  OF
    ...
    declproc( $\xi, \gamma_1, formals$ ) :
      IF  $\xi \in domain(\rho_p) \vee$ 
         $\xi \in domain(map(Env(StoreOf(\sigma))))$  THEN
         $\sigma$ 
      ELSE
         $\sigma$  WITH [( $\rho_p := add((\xi, (formals, \gamma_1)), \rho_p)$ )]
      ENDIF,
    ...
  ENDCASES
  MEASURE sizeof( $\gamma$ ) + Step( $\sigma$ )

```

Procedure calls are made by translating them into other Tosca commands as described previously. The function *AsPar* is used to generate the list of assignments necessary to associate the formal and actual parameters.

```

AsPar(formals : list[(declvar?)], actuals : list[Expr]) : RECURSIVE list[Cmd] =
  IF null?(formals)  $\vee$  null?(actuals) THEN
    null
  ELSE
    cons(assign(vi(car(formals)), car(actuals)),
          AsPar(cdr(formals), cdr(actuals)))
  ENDIF
MEASURE length(formals)

 $\mathcal{M}_C[\gamma : \text{DeclCmd}](\sigma : \text{State}) : \text{RECURSIVE State} =$ 
  LET  $\rho_p : \text{Env}_P = \rho_p(\sigma)$  IN
  CASES  $\gamma$  OF
    ...
    npproc( $\xi$ ) :
      IF  $\xi \in \text{domain}(\rho_p) \wedge \neg \text{Step}(\sigma) = 0$  THEN
         $\mathcal{M}_C[\text{proj}_2(\text{apply}(\rho_p, \xi))](\sigma)$ 
      ELSE
         $\sigma$ 
      ENDIF,

    callproc( $\xi$ , actuals) :
      IF  $\xi \in \text{domain}(\rho_p) \wedge \neg \text{Step}(\sigma) = 0$  THEN
         $\mathcal{M}_C[\text{local}(\text{decls}(\text{proj}_1(\text{apply}(\rho_p, \xi))),$ 
              block(append(AsPar(proj1(apply( $\rho_p$ ,  $\xi$ )), actuals),
                    cons(npproc( $\xi$ ), null)))]( $\sigma$ )
      ELSE
         $\sigma$ 
      ENDIF
  ENDCASES
MEASURE sizeof( $\gamma$ ) + Step( $\sigma$ )

```

10.2 Compiler

The compilation of procedure parameters is performed in the same manner as the dynamic semantics of procedure calls, i.e. by syntactic transformation of the call into other Tosca instructions and compiling those.

```

 $\mathcal{O}_C \Downarrow \gamma : DeclCmd \Downarrow (\rho_o : Inj_{Env})(SP : \{l : Locn \mid l \geq top(\rho_o)\})$ 
       $(\phi : Label)(\rho_{op} : Env_{OP}) : RECURSIVE Code =$ 
  CASES  $\gamma$  OF
    ...
    callproc( $\xi$ , actuals) :
      IF  $\xi \in domain(\rho_{op})$  THEN
         $\mathcal{O}_C \Downarrow local(decls(proj_1(apply(\rho_{op}, \xi))),$ 
          block(append(AsPar(proj_1(apply(\rho_{op}, \xi)), actuals),
            cons(npproc(\xi), null))))  $\Downarrow (\rho_o)(SP)(\phi)(\rho_{op})$ 
      ELSE
        DeadCode
      ENDIF
    ...
  ENDCASES
  MEASURE sizeof( $\gamma$ )

```

The compilation of procedure declarations, creating the environment Env_{OP} is slightly complicated from that given in the previous chapter to cope with the declaration of the formal parameters to the procedure. The definition of the environment is changed to the following, in a similar manner to Env_P in the previous section:

```

 $Env_{OP} : TYPE = (part_{inj}[Name, [list[(declvar?)], Label]])$ 

```

Thus the new definition of procedure declarations becomes:

```

 $\mathcal{O}_C \langle \gamma : DeclCmd \rangle (\rho_o : Env) (SP : \{l : Locn \mid l \geq top(\rho_o)\})$ 
  ( $\phi : Label$ ) ( $\rho_{op} : Env_{OP}$ ): RECURSIVE Code =
  CASES  $\gamma$  OF
    ...
    declproc( $\xi, \gamma_1, formals$ ):
      IF  $\xi \in domain(\rho_{op}) \vee \xi \in domain(map(\rho_o))$  THEN
        DeadCode
      ELSE
        LET  $code : Code = \mathcal{O}_C \langle \gamma_1 \rangle (\rho_o)(SP + 1)(\phi + 1)$ 
          ( $add((\xi, (formals, \phi)), \rho_{op})$ ) IN
          ( $proj_1(code), cons(label(\phi),$ 
             $cons(spinc,$ 
               $cons(store(SP),$ 
                 $append(proj_2(code),$ 
                   $cons(rts, null))))),$ 
             $\rho_o, add((\xi, (formals, \phi)), \rho_{op})$ )
        ENDIF,
    ...
  ENDCASES
  MEASURE sizeof( $\gamma$ )

```

10.3 Aida Semantics

Because we have implemented procedure parameters by syntactic transformation of the source language, there are no augmentations required to the low-level semantics.

10.4 Proof

The only existing proofs that required modification after the addition of procedure parameters were those related to the declaration of parameterless procedures, which is not now permitted (although you can declare a procedure with an empty list of parameters).

As we have already proved the correct compilation of the constructs used in the implementation of procedure parameters, the proof obligation is easily discharged by using the correct compilation of these constructs as lemmata. This is, however, not the true proof obligation here — what we really wish to prove is that the chosen method of emulating procedure parameters is consistent with the intuitive view of call by value procedure parameters.

We performed such a proof by writing semantics for a direct implementation of call by value procedures and showing this was equivalent to the implementation described in this chapter. The details of the direct implementation follow any standard text on compilers, and are not reported here. The proof was simple due to the small ‘semantic gap’ between the two languages (direct and emulated).

10.5 Summary

The chosen method of implementing procedure parameters by syntactic transformation resulted in a very quick implementation and proof. The code generated from this is no less efficient than it would have been by direct implementation — we would still have required the allocation of stack space for the parameters, evaluation and assignment of the actuals, and execution of the procedure body.

Part IV

Discussion and Conclusions

Chapter 11

Discussion

This chapter presents a discussion of the work presented in this thesis, the contribution we have made, and ways in which this work could be further developed.

11.1 Formal Analysis of Compilers

We noted in earlier chapters that computer systems tend to be very complex, and that we can use ‘formal methods’ in order to increase our assurance that they will behave as intended. This is not, however, the exclusive domain of formal methods — traditional methods of software construction, developed in the short history of programming, have also contributed much to the reliability of software systems. By this, we mean techniques such as structured development methodologies, testing, and so on.

Most computer systems are engineered using these traditional techniques, and perform their task adequately. They may fail from time to time due to errors in their (informal) specification, their design and coding, but these errors are tolerated due to a cost/integrity tradeoff — the software is designed to ‘best practice’ for the money available.

The development of formal methods has striven to increase the safety and reliability of systems by providing the opportunity to *calculate* properties of those systems from their specifications and to examine *all* behaviours, a situation which is difficult (or impossible) to achieve using traditional software engineering techniques.

However, this extra confidence comes with a price — the cost/integrity tradeoff referred to earlier. It is widely believed that the application of formal methods requires specially trained staff, and is a costly process. Thus, its application is restricted to systems where the highest levels of integrity are essential, for example in safety critical systems such as aircraft control and nuclear power stations.

However, experience has shown[33] that this cost/integrity tradeoff is not a simple matter. The true payback from formal methods seems to be from application to problems which are inherently difficult and complex — for example algorithms for concurrency and fault tolerance. Application of formal methods to areas late in the software development cycle have showed little payback for the effort invested — traditional techniques such as testing do actually catch most errors and problems *introduced* at that stage.

When conducting an initial survey of trusted compilation for this thesis, we found no examples of systems where compilers had been known to introduce errors into delivered systems¹. Testing is very effective at unearthing such errors as it is, after all, the object code produced by the compiler which is the subject of testing rather than the source code produced by the programmer.

This then could be used as an argument as to why we did not unearth any errors in the DCC work (although we did unearth some assumptions) — we were starting from a development which had already been through a certain level of formal development and proof, but not one where errors are traditionally found. The processes used in the DCC method (formal specification, hand proof, and human review) were sufficient to give a high level of confidence in that compiler.

However, we would argue that this is not an invalidation of performing research in the area of trusted compilation. The arguments presented in the introduction still apply — the compiler is a weak link in the production of object code, and just because no disasters have so far been caused by compiler bugs there is no reason to assume that this will remain the case. Hence, UK Def-Stan 00-55[7] mandates compilers which are developed to the same integrity level as the application code.

¹The nearest ‘miss’ would be the errors located in the Sizewell B Primary Protection System reported in Section 3.2.3. Although these errors were caught before delivery or operation of the system, it would be unfair to refer to the analysis they performed as mere ‘testing’.

What does follow from this is that application of formal methods in this area will give little payback for the effort involved until methods can be developed which reduce the human effort involved. This has been one of the themes of this thesis, and the level of its success is discussed further in later sections of this chapter.

11.2 Tool Support

The major goal of this thesis has been to explore to what extent mechanical verification can provide assistance in the process of performing analytical compiler verifications. In this section, we present a discussion of our experiences with PVS, and its utility as a tool for supporting compiler verification. We also discuss how specific our work is to PVS, and the prospects for performing similar work in other verification systems. We start by evaluating the transition from a Z specification with a hand proof to a specification and proof in PVS.

11.2.1 Z in PVS

A large proportion of the research effort was spent in translating the Z specification into the logic of the PVS system. PVS uses an entirely different paradigm for specifications, being a logic of total functions compared to Z's heavy use of partial functions and its basis in (ZF) set theory. The general translation of Z into a logic of total functions is not an easy task (cf. work on Z/HOL) and neither was our specific case of translating a particular Z specification, rather than trying to embed the Z language in its entirety.

To an extent, this was a distraction from the main goal of this work, i.e. to ease the task of performing verifications of compilers. However, our experimentation in various methods of expressing the compiler semantics in PVS has yielded some interesting lessons for those involved in providing proof support for Z.

Our need to introduce rather complex types for functions to ensure they are not applied outside their domain has been echoed by Saaltink in the development of Z/EVES[137]. Z/EVES contains a domain checker, which is used to ensure functions are applied to arguments within their domain, thus separating the issue from the main specification. We are unsure however if

Z/EVES would have enabled the near automatic proof of domain conditions as we achieved with PVS through explicit augmentation of types.

As an interesting aside, Saaltink notes in a paper at ZUM '97[137] that *every* specification he has applied Z/EVES domain checking to has failed, i.e. contained out of domain applications. Thus, the specifications contain undefined statements. This may not, however, affect the ability to reason about those specifications as the current interpretation[138] of the Z standard approach to ‘undefinedness’ retains the axiom of reflection, i.e. all predicates of the form $\epsilon = \epsilon$ where the two expressions ϵ are textually identical are unconditionally true.

In summary, translating Z to PVS was the wrong approach. The choice of starting point for this work added significant complications to the work, and we would commend starting from scratch in any new verification, writing directly in the native logic of your chosen verification system.

As an aside, we note that during the course of the work presented here, there has been much work in the area of verification support for Z. The interested reader is referred to a review by Martin[103] for details. Our feeling is that this does not invalidate our decision to use PVS, even in a present day context, as the automation available in current Z proof tools is still limited.

11.2.2 Use of PVS

The results achieved here are not specific to PVS, even though a lot of the detailed issues reported here are PVS specific. Most of those issues arose as a result of our need to translate the compiler specification from Z to PVS. Earlier in this chapter we stated that we now consider this to have been the wrong approach to take, as it resulted in significant distraction from the main objective of the thesis.

In the chapter on PVS, we described it as a mature, general purpose specification and verification system. The maturity of the system gives us confidence that the system is reliable, and gives us confidence in the correctness of its results². Its status as a general purpose system, implies it is suitable for a wide range of applications, but more than that — its results should be reproducible in other general purpose verification systems³.

²A verified verification system is still a holy grail.

³One may wish to reproduce results in another system to gain increased confidence.

There is a certain amount of evidence to support this claim that verifications performed in PVS are not specific to that system. Firstly, a simple introductory example of verification in PVS, due to Butler[139] was given a more sophisticated treatment in a report by Rushby and Stringer-Calvert[2]. This treatment, which relied heavily on the more automatic theorem proving strategies in PVS was translated into the Mizar[140] system by Rudnicki[141].

Another example is the verification of the interactive consistency algorithm, which was ‘ported’ from PVS to ACL2⁴ by Young[147]. This problem was proposed by Rushby as a benchmark problem for specification and verification systems, and Young reports how the elegant treatment of the problem in PVS was reproduced in ACL2, which lacks support for some of the features Rushby considered most useful (strong typing and higher-order functions) in the development of his improved formulation.

The existence of such examples of verifications performed in PVS being repeated in other systems lends strength to our argument that the work presented in this thesis is not tool dependent.

11.2.3 Necessary Tool Features

In this section, we discuss the necessary features of a specification and verification system for performing the type of work we have presented here. We therefore take as read that any system under consideration should support the basic necessities such as induction, rewriting, and a tactic or strategy language. We will not discuss the general issues of theorem proving support, instead the interested reader is referred to [33, Section 2.6 and Section 3.5.1].

In our development, we found predicate subtypes to be a useful tool to encode domain conditions and other constraints for functions. They allowed for the encoding of specification information in the types of objects, making it available to the ground prover and thereby increasing the efficiency of the automation.

The advantages in automation gained by encoding constraints in predicate subtypes is specific to the way that the decision procedures operate in PVS,

⁴The problem was originally formulated by Lamport, Shostak and Pease[142, 143], and then mechanically checked by Bevier and Young[144] in NQTHM. Rushby then provided a more elegant solution[145] in EHDM and then later in PVS[146] which was used as the basis for the ACL2 formulation described here.

and will probably give little gain in other verification environments. However, we would label this as a useful feature for our work — the use of predicate subtypes enabled many errors in the specification to be caught through type-checking, and thereby caught earlier. This is the same argument as the use of strong type systems for programming languages — the earlier a defect is located, the cheaper it is to fix — so the language should provide enough discipline in the application of functions and so on to facilitate this⁵.

The major stumbling block in our work which led to most of our proof strategies breaking at some point is instantiation. PVS provides a heuristic instantiation mechanism for existential strength quantifiers, for example in instantiating a lemma or induction schema. This has various options to control the heuristics, but none of them are quite right for our purposes, requiring us to provide the correct instantiation by hand.

This acts against our goal of providing useful automation in the mechanical presentation of the DCC method. The more *specific* and *low-level* the steps in each individual proof are, the less general the proof will be and it is likely that it will not re-run without modification when the specification is changed.

The designers of PVS agree⁶ that instantiation is a weak point of their system, and to provide it with more ‘intelligence’ would require the implementation of a better form of matching, for example unification.

The underlying automation in the PVS theorem prover, specifically the tight integration between the decision procedures and rewriting, is the main reason we were able to achieve a level of abstraction in our mechanical proofs that closely resembles the hand proofs. This issue (which is only of interest to those building theorem provers) is discussed further in [2].

We used PVS strategies to perform the proofs of obligations generated due to domain membership conditions. These could have been stated as lemmata, and proven once, but the instantiation of these lemmata would themselves have generated type correctness conditions, which would require discharging. The strategies also allowed us to reuse sections of proof without tedious interaction with the proof tool.

⁵However, the issue of whether a specification language should be typed *at all* is a point of debate, and covered at length in a report by Lammport and Paulson[148].

⁶Rushby and Shankar, in private communication, 1995.

11.3 Comparison with DCC Development

It is interesting to note that we did not find any errors in the by hand development of the original Tosca compiler. There are several ways in which this can be explained (their use of Prolog for validation, human review of specifications and proof, etc.), but it does enable us to conclude that effective verifications of small compilers may be undertaken without mechanical support.

The effort required to perform the original DCC proofs with PVS has been large (over a person year, with the usual interruptions — considerably more than that to perform the proof by hand). It has been noted by the authors of PVS⁷ that this is one of the largest and more complex theorems passed through their system, and the theorems we are required to discharge here are of a very different nature from the theorems that have been specified in PVS previously. We have been stretching the limits of the type system, for example in the use of doubly-dependent types and abstract datatypes with subtypes.

However, the introduction of new constructs has built on our familiarity with the proof system and the specification, allowing augmentation at little cost. The new constructs introduced took approximately a month each. The following table summarizes the growth in the size of the specification (in terms of lines).

Specification	Original Tosca	Local Scope	Procedures	Parameters
Common Parts	231	243	267	284
Aida	261	279	286	286
Tosca	501	523	598	669
Compiler	147	159	183	200

11.4 Scalability

It would be inappropriate to draw any conclusions about the limits of the mechanized DCC approach as we have studied only a small number of extensions to Tosca, and not encountered any major difficulties in doing so. The

⁷By Rushby, in private communication, 1997.

previous section reported how the size of the specifications increased in a linear manner on the addition of the new constructs considered. As the DCC compiler has been extended (by hand) to the much larger compiler reported in Section 5.3 we would contend that our method is equally scalable.

However, due to the constraints we have placed in the specification and the constraints of PVS, there are constructs which would be difficult to implement. One example is functions: the inclusion of functions requires the expression semantics to reference the command semantics (the reverse is already true). Because of the lack of recursion in PVS, this would require (in our implementation) the expression ADT to be combined with the command ADT.

In Chapter 10 we used syntactic transformation to implement procedure parameters. This allowed us to provide a simple, quick implementation with the minimum of extra effort, but did not introduce any significant overhead into the executable code. It also provided a method of structuring the proof — we did not need to redo the proof of parameterless command translation, but instead verified the correspondence between true procedures with parameters and our implementation. As the two languages involved in this proof were very similar, the proof effort required was less than would have been required for direct implementation.

This structuring (which is similar to ‘stacking’ as used in the CLI approach, described in Section 3.4.2) could be used to implement other features such as different data sizes at Tosca and Aida levels, for loops, and case statements.

The hand DCC extension has also introduced separate compilation, which is an entirely different problem, that of verifying a linker. The ‘meaning’ of a program module is not intuitive and neither is the correctness condition for linking. Type safety is a good starting point but other considerations may be needed, for example memory utilization, timing calculations and so on.

11.5 Relation to Previous Work

The body of work reported in this thesis is of a smaller scale than most of the analytical compiler verifications reported in Chapter 3. The key point which differentiates our work is the presentation of a *mechanized* method for compiler verification, at a level of abstraction similar to hand proofs. Thus,

we have shown that it is feasible to use mechanical support tools for such verifications at an attractive cost/integrity tradeoff level.

We have used a combination of methods reported in previous work in the implementation of extensions to the original Tosca compiler. The use of the loop counter is due to Yu's M68020 object code verifier, and also to the CLI stack. The use of syntactic transformations to implement procedure parameters could be considered as stacking, as in the CLI approach: the extension with procedure parameters is effectively compiled into the version of Tosca with procedures as an intermediate language.

11.6 Summary

This chapter has presented a discussion of the key aspects of the thesis. These are summarized in the next chapter, together with some suggestions for future work.

Chapter 12

Conclusions and Future Work

This thesis has the theme of mechanical verification for compiler correctness, with the following characteristics:

- the use of a fully formal and rigorous framework;
- the degree of automation achieved in the proof process, and the resilience of that automation to changes in the specification; and
- the combination of two methods of adding new features — direct augmentation of the source language, and the use of syntactic transformation.

As well as a contribution to the area of compiler correctness, we have also made a contribution to the area of proof tool support for Z specifications. These contributions are now briefly summarized — for further detailed discussion the reader is referred to Chapter 11.

12.1 Mechanical Formal Methods

Tool support is not an essential ingredient in the verification of compiler correctness, at least for compilers of a small scale. This has been shown in previous work, and its validity checked by our mechanization of an existing by-hand development not finding any significant errors.

However, we have demonstrated in this thesis that introducing tool support does not overly complicate the proof process. Once we had performed the initial translation of the Z specification into PVS, the addition of new features was a simple task of comparable complexity to the addition of new constructs in the by-hand work (where they performed only proof ‘sketches’).

The major gain in using tool support has been the checking for consistency and incompleteness in the specification, and the requirement to make all proof steps explicit.

12.2 Scalability and Complexity

We cannot draw any conclusions as to limits of the scalability of the mechanized DCC method from this work. We have seen, in the examples of augmentations, that the complexity of the specification and proof increases in a linear manner. We believe that this linear increase will continue as such, although any new construct which requires major changes to the underlying types in the specification will result in a larger amount of work, especially in the re-running of the proofs of existing constructs.

12.3 Automation

We have provided effective automation for the proof of ‘domain conditions’, which has resulted in the interaction with the proof tool being at a similar level of abstraction to that involved in the by-hand (journal style) presentation of the proofs.

We were not able to provide much automation for the discovery of the main branch of the proofs. This is due to:

- the inadequacy of the PVS heuristic instantiations; and
- the convoluted nature of the specification, since they had been translated from Z.

12.4 Resilience

We have demonstrated that our mechanical proofs require little or no modification upon the addition of new constructs to the source and target languages. This has been achieved through:

- the use of strategies for automating the discovery of proofs; and
- ‘good proof style’, by which we mean using more general proof steps, rather than specific, detailed, proof steps.

12.5 Mechanizing a Hand Development

The main issue in the translation from Z to PVS was the diverse basis of the specification languages concerned (Z -F set theory and higher-order logic). This created difficulties in the translation, for example total functions.

Due to this, we do not believe that starting from a by-hand development made the process easier. A better approach would be to start from scratch in the native logic of the tool you intend to use.

12.6 PVS for Z

Despite the comments in the above section, we have shown that (with some work) PVS can be used to prove conjectures cast in Z . We believe that our approach should scale to providing a full support tool for Z specifications, but doubt that this is a useful approach to take. To ensure exact adherence to the (emerging) semantics of Z , the PVS terms would be quite unnatural.

12.7 Suggestions for Future Work

12.7.1 Compiler Verification

Despite the positive results of this work, we would not recommend anyone to use the specifications and proofs of our work as a starting point for future work. We have demonstrated that the DCC method is mechanizable, but to

achieve real benefits in terms of automation it will be necessary to recast the problem directly into the native logic of a verification system.

A limiting factor of this work at present is the use of axioms describing the correct construction of the Env_I environment, mapping from program labels to continuations. Elimination of these will require modification of the operational semantics to generate the environment during the execution of the compiler.

One criticism that has been presented of the DCC approach is that it does not tackle any optimizations. Here, we dismissed optimization as an unsafe phase of compilation but in practice many production compilers generate safer code with optimization turned on¹. Extending the DCC work to cover peephole optimizations would be relatively easy, by taking the output of the current compiler and using a separate optimization pass which could be verified independently of the main compiler.

Another interesting avenue of work would be the application of the DCC method to a different source language, possibly based on an entirely different programming paradigm, for example functional programming. This would lead to the discovery of any part of the approach which is inherently specific to the Tosca/Aida compiler, and aid the scaling of the method to be generally applicable.

The current state of the art in compiler verification is the Verifix project, reported in Section 3.4.4. Their approach (which was developed concurrently with ours) builds from a complete embedding of various semantic frameworks within the logic of PVS. Initial results are encouraging the belief that their work will scale to realistic industrial compilers.

12.7.2 Semantic Descriptions

There is little reported in the literature on the use of mechanical support for the development of semantic descriptions of programming languages. We found the support offered by the PVS type checker in this regard to be very useful. The construction of a tool for this purpose could be generated using PVS by building on the semantic embeddings written in the Verifix project, described above.

¹This is probably due to better testing. An optimizing compiler is likely to have more code compiled with optimization on than off, resulting in a better coverage of the compiler.

Bibliography

- [1] David W.J. Stringer-Calvert, Susan Stepney, and Ian Wand. Using PVS to prove a Z refinement: A case study. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME 97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, pages 573–588, Grätz, Austria, September 1997. Springer-Verlag.
- [2] J.M. Rushby and D.W.J. Stringer-Calvert. A less elementary tutorial for the PVS specification and verification system. Technical Report CSL-95-10, Computer Science Laboratory, SRI International, August 1996.
- [3] Susan Stepney. *High Integrity Compilation: A Case Study*. Prentice Hall International, 1993.
- [4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.
- [5] Chalres N. Fischer and Richard J. LeBlanc Jr. *Crafting a Compiler*. Benjamin Cummings, 1988.
- [6] K. Thompson. Reflections on trusting trust (deliberate software bugs). *Communications of the ACM*, 27(8):761–763, August 1984.
- [7] MoD. The procurement of safety critical software in defence equipment. Interim Defence Standard 00-55 part 1, Ministry of Defence, 1989.
- [8] DoD. Trusted computer system evaluation criteria. Department of Defense Standard 5200.28-STD (The Orange Book), United States Department of Defense, December 1985.

- [9] Paul Curzon. Of what use is a verified compiler specification? Technical Report 274, University of Cambridge Computer Laboratory, 1992.
- [10] Paul Curzon. A verified vista implementation final report. Technical Report 311, University of Cambridge Computer Laboratory, September 1993.
- [11] Paul Curzon. The verified compilation of vista programs. In *1st ProCos Working Group Meeting*, Gentofte, Denmark, January 1994.
- [12] Paul Curzon. A verified compiler for a structured assembly language. In *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*. IEEE Computer Society Press, 1992.
- [13] Paul Curzon. A programming logic for a verified structured assembly language. In A. Voronkov, editor, *Logic Programming and Automated Reasoning*, volume 624 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1992.
- [14] J. Strother Moore. PITON: A verified assembly language. Technical Report 22, Computational Logic Inc., September 1988.
- [15] Susan Stepney, Dave Whitley, David Cooper, and Colin Grant. A demonstrably correct compiler. *Formal Aspects of Computing*, 3:58–101, 1991.
- [16] J D Ichbiah et al. Reference manual for the Ada programming language. ANSI MIL-STD 1815A, 1983.
- [17] K A Nyberg (Editor). The annotated Ada reference manual. ANSI MIL-STD 1815A (Annotated), Grebyn Corporation, 1989.
- [18] Praxis Critical Systems, Bath, UK. *SPARK — The SPADE Ada Kernel*, 3.2 edition, October 1996.
- [19] William Marsh. Formal semantics of SPARK - static semantics. Report PVL/SPARK_DEFN/STATIC/V1.3, Program Validation Ltd., October 1994.

- [20] Ian O’Neill. Formal semantics of SPARK - dynamic semantics. Report PVL/SPARK_DEFN/DYNAMIC/V1.4, Program Validation Ltd., October 1994.
- [21] Charles Forsyth, David Jordan, and Ian Wand. Trusted Ada compilation. Ministry of Defence Contractor Report SLS31c/73-3-D, York Software Engineering Ltd., February 1993.
- [22] IEC. Programmable controllers - Part 3: Programming languages. International Standard IEC 1131-3, International Electrotechnical Commission, 1993.
- [23] M.E. Lesk. Lex — a lexical analyzer generator. Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ, 1975.
- [24] A.V. Aho and S.C. Johnson. Lr parsing. *Computing Surveys*, 6(2):99–124, 1974.
- [25] Axel Dold, F.W. von Henke, H. Pfeifer, and H. Rueß. A generic specification for verifying peephole optimizations. Ulmer Informatik-Berichte 95-14, Universität Ulm, Fakultät für Informatik, 1995.
- [26] Axel Dold, F.W. von Henke, H. Pfeifer, and H. Rueß. Formal verification of transformations for peephole optimizations. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME 97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, pages 459–472, Grätz, Austria, September 1997. Springer-Verlag.
- [27] D. Scott and C. Strachey. Towards a mathematical semantics for computer language. Technical Report PRG-6, Programming Language Research Group, University of Oxford, 1971.
- [28] David A. Watt. *Programming Language Syntax and Semantics*. Prentice Hall International, 1991.
- [29] C.A.R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2:335–355, 1973.
- [30] C.A.R. Hoare. Consistent and complementary formal theories of the semantics of programming languages. *Acta Informatica*, 3:135–153, 1974.

- [31] Peter D. Mosses. *Action Semantics*. Number 26 in Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
- [32] P.D. Mosses. Unified algebras and action semantics. In *Symposium on the Theory of Automata and Computer Science*. Springer-Verlag, 1989.
- [33] John Rushby. Formal methods and digital systems validation for airborne systems. Technical Report CSL-93-07, Computer Science Laboratory, SRI International, November 1993.
- [34] Brian Randell, editor. *The Origins of Digital Computers: Selected Papers*. Springer-Verlag, second edition, 1975.
- [35] A.M. Turing. Checking a large routine. In D.C. Ince, editor, *Collected Works of A.M. Turing: Mechanical Intelligence*, pages 129–131. North-Holland, Amsterdam, The Netherlands, 1992. Originally presented at EDSAC Inaugural Conference on High Speed Automatic Calculating Machines, 24 June 1949.
- [36] Yuan Yu. *Automated Proofs of Object Code for a Widely Used Microprocessor*. PhD thesis, University of Texas at Austin, 1992.
- [37] Yuan Yu. Automated proofs of object code for a widely used microprocessor. Technical Report 114, Digital Systems Research Center, 5 October 1993.
- [38] Robert S. Boyer and J. Strother Moore. *A Computational Logic Handbook*. Academic Press, 1988.
- [39] Hanan Samet. Proving the correctness of heuristically optimized code. *Communications of the ACM*, 21(7):570–582, July 1978.
- [40] H. Samet. *Automatically proving the correctness of translations involving optimized code*. PhD thesis, Computer Science Department, Stanford University, Stanford, California, 1975.
- [41] D.J. Pavey and L.A. Winsborrow. Demonstrating equivalence of source code and PROM contents. *The Computer Journal*, 36(7):654–667, 1993.

- [42] Lesley A. Winsborrow and Deryk J. Pavey. Assuring correctness in a safety critical software application. *High Integrity Systems*, 1(5):453–459, 1996.
- [43] D.J. Pavey and L.A. Winsborrow. Formal demonstration of equivalence of source code and PROM contents: an industrial example. In Chris Mitchell and Victoria Stavridou, editors, *Mathematics of Dependable Systems*, pages 225–248, Royal Holloway, University of London, September 1993. Institute of Mathematics and its Applications, Clarendon Press.
- [44] Rex, Thompson and Partners Ltd. (TA Consultancy Services Ltd.). *MALPAS User Guide release 5.1*, rtp/9039/01 edition, April 1991.
- [45] J. Hannaford, D.M. Hunns, M.R. Sayers, N. Wainwright, and R.L. Yates. The Sizewell B protection system: Status report on NII's assessment of the primary protection system software. 1 July 1993.
- [46] Gunnar Stålmarmark and M Säflund. Modelling and verifying systems and software in propositional logic. In *Proceedings of IFAC SAFE-COMP'90*, London, UK, 1990.
- [47] D.L. Buttle. Compilation verification. Technical report in preparation, Department of Computer Science, University of York, 1998.
- [48] Peter D. Mosses. SIS – semantics implementation system. Technical Report Daimi MD-30, Computer Science Department, Aarhus University, 1979.
- [49] Lawrence Paulson. A semantics-directed compiler generator. In *Ninth Symposium on Principles of Programming Languages*, pages 224–233. ACM Press, January 1982.
- [50] Mitchell Wand. A semantic prototyping system. In *Proceedings ACM SIGPLAN'84 Symposium on Compiler Construction*, pages 213–221. SIGPLAN Notices, 1984.
- [51] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

- [52] Martin Raskovsky. Generating a real compiler from a denotational semantics. Technical Report CSM-41, Department of Computer Science, University of Essex, May 1981.
- [53] Martin Raskovsky. Step by step generation of a compiler for flow diagram language with jumps. Technical Report CSM-42, Department of Computer Science, University of Essex, June 1981.
- [54] Uwe F. Pleban and Peter Lee. An automatically generated, realistic compiler for an imperative programming language. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 222–232, Atlanta, Georgia, 22–24 June 1988.
- [55] Uwe F. Pleban and Peter Lee. A realistic compiler generator based on high-level semantics. In *Proceedings of the 14th annual SIGACT/SIGPLAN Conference on Principles of Programming Languages*, pages 284–295, Munich, West Germany, January 1987.
- [56] U. F. Pleban and P. Lee. On the use of LISP in implementing denotational semantics. In *Proceedings ACM Conference on LISP and Functional Programming*, pages 233–248, August 1986.
- [57] Jens Palsberg. An automatically generated and provably correct compiler for a subset of Ada. In *Proceedings of the fourth IEEE International Conference on Computer Languages*, San Francisco, CA, 20–23 April 1992.
- [58] Jens Palsberg. A provably correct compiler generator. In *Proceedings of ESOP '92, European Symposium on Programming*, 1992.
- [59] Jens Palsberg. *Provably Correct Compiler Generation*. PhD thesis, Computer Science Department, Aarhus University, January 1992.
- [60] A. Agrawal and R.B. Garner. SPARC: A scalable processor architecture. In *1990 International Conference on Information Technology (InfoJapan'90)*, volume 7 (2–3) of *Future Generation Computer Systems*, pages 303–309, April 1992.
- [61] Hewlett Packard. Precision architecture and instruction. Technical Report 09740-90014, June 1987.

- [62] R.M. Burstall and P.J. Landin. Programs and their proofs: an algebraic approach. *Machine Intelligence*, 4:17–43, 1969.
- [63] P.A. Collier. Simple compiler correctness - a tutorial on the algebraic approach. *Australian Computer Journal*, 18(3):128–135, August 1986.
- [64] R. Milner and R. Weyhrauch. Proving compiler correctness in a mechanized logic. *Machine Intelligence*, 7:51–70, 1972.
- [65] Laurian M. Chirica and David F. Martin. Toward compiler implementation correctness proofs. *ACM Transactions on Programming Languages and Systems*, 8(2):185–214, April 1986.
- [66] L. Morris. Advice on structuring compilers and proving them correct. In *Proceedings ACM Symposium on Principles of Programming Languages*, pages 144–152, 1973.
- [67] J.W. Thatcher, E.G. Wagner, and J.B. Wright. More advice on structuring compilers and proving them correct. In *Automata, Languages and Programming – Sixth Colloquium*, volume 71 of *Lecture Notes in Computer Science*, pages 596–615. Springer-Verlag, 1979.
- [68] Martin C. Henson. Extending advice on structuring compilers and proving them correct. Technical Report CSM-56, Department of Computer Science, University of Essex, May 1993.
- [69] Wolfgang Polak. *Compiler Specification and Verification*. Number 124 in *Lecture Notes in Computer Science*. Springer-Verlag, 1981.
- [70] Stanford Verification Group. Stanford Pascal verifier user manual. Technical Report 11, Stanford Verification Group, 1979.
- [71] William D. Young. A verified code generator for a subset of Gypsy. Technical Report 33, Computational Logic Inc., October 1988.
- [72] William D. Young. A mechanically verified code generator. Technical Report 37, Computational Logic Inc., January 1989.
- [73] William R. Bevier, Jr. Warren A. Hunt, J Strother Moore, and William D. Young. An approach to systems verification. Technical Report 41, Computational Logic Inc., April 1989.

- [74] William D. Young. System verification and the CLI stack. In Jonathan Bowen, editor, *Towards Verified Systems*, pages 225–248. Elsevier Science Publishers Series on Real-Time Safety Critical Systems, Amsterdam, 1993.
- [75] D.I. Good, R.L. Akers, and L.M. Smith. Report on Gypsy 2.05. Technical Report 1, Computational Logic Inc., October 1986.
- [76] J. Strother Moore. A mechanically verified language implementation. Technical Report 30, Computational Logic Inc., September 1988.
- [77] Jonathan Bowen, C.A.R. Hoare, Michael R. Hansen, Anders R. Ravn, Hans Rischel, Ernst-Rüdiger Olderog, Michael Schenke, Martin Fränzle, Markus Müller-Olm, Jifeng He, and Zheng Jianping. Provably correct systems - FTRTFT'94 tutorial. In *Proceedings of FTRTFT'94*, number 863 in Lecture Notes in Computer Science. Springer-Verlag, September 1994.
- [78] C.A.R. Hoare. Refinement algebra proves correctness of compiling specifications. In *3rd Refinement Workshop*, Workshops in Computing, pages 33–48. Springer-Verlag, 1991.
- [79] C.A.R. Hoare, He Jifeng, Jonathan Bowen, and Paritosh Pandya. An algebraic approach to verifiable compiling specification and prototyping of the ProCoS level 0 programming language. ProCoS Project Document OU CARH 2/1, May 1990.
- [80] He Jifeng and Jonathan Bowen. Specification, verification and prototyping of an optimized compiler. *Formal Aspects of Computing*, 6(6):643–658, 1993.
- [81] Deborah Weber-Wulff. Proven correct scanning. ProCoS internal report, August 1992.
- [82] Debora Weber-Wulff. Proof movie — a proof with the Boyer-Moore prover. *Formal Aspects of Computing*, (5):121–151, 1993.
- [83] Wolf Zimmermann and Thilo Gaul. On the construction of correct compiler back-ends: An ASM approach. *Journal of Universal Computer Science*, 3(5):504–567, 1997. Available from http://www.iicm.edu/jucs_3_5.

- [84] Markus Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*. Number 1283 in Lecture Notes in Computer Science. Springer-Verlag, 1997. University of Kiel PhD Thesis.
- [85] H. Pfeifer, A. Dold, F. W. v. Henke, and H. Rueß. Mechanized Semantics of Simple Imperative Programming Constructs. Ulmer Informatik-Berichte 96-11, Universität Ulm, Fakultät für Informatik, 1996.
- [86] IEEE. IEEE standard for the Scheme programming language. IEEE Std 1178-1990, Institute of Electrical and Electronic Engineers, New York, 1991.
- [87] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *LISP and Symbolic Computation*, 7(4):315–335, 1994.
- [88] Joshua D. Guttman, John D. Ramsdell, and Mitchell Wand. VLISP: A verified implementation of Scheme. *LISP and Symbolic Computation*, 5–32(8), 1995.
- [89] Joshua D. Guttman, John D. Ramsdell, and Vipin Swarup. The VLISP verified scheme system. *LISP and Symbolic Computation*, 8:33–110, 1995.
- [90] Dino P. Oliva, John D. Ramsdell, and Mitchell Wand. The VLISP verified PreScheme compiler. *LISP and Symbolic Computation*, 8:111–182, 1995.
- [91] Mitchell Wand. Semantics-directed machine architecture. In *Conference Record 9th ACM Symposium on the Principles of Programming Languages*, pages 234–241, 1982.
- [92] William Clinger. The Scheme 311 compiler: An exercise in denotational semantics. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 356–364, New York, 1984.
- [93] Mitchell Wand and Dino P. Oliva. Proving the correctness of storage representations. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 151–160, New York, 1992.

- [94] Dino P. Oliva. *Advice on Structuring Compiler Back Ends and Proving them Correct*. PhD thesis, College of Computer Science, Northeastern University, January 1994.
- [95] J. Kershaw. Vista user's guide. Technical Report 401-86, Royal Signals and Radar Establishment, 1986.
- [96] P. Sreeranga Rajan. Executing HOL specifications: Towards an evaluation semantics for classical higher order logic. In L. Clasen and M. Gordon, editors, *Higher Order Theorem Proving and its Applications*, number A-20 in IFIP Transactions, North Holland, 1992.
- [97] W.J. Cullyer and J. Kershaw. VIPER: a new microprocessor for safety-critical applications. In *Proceedings of MILCOMP '85 conference*, pages 269–274, London, England, October 1985.
- [98] W.J. Cullyer. Implementing safety critical systems: The VIPER microprocessor. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 1–25. Kluwer Academic Publishers, 1988.
- [99] J. M. Spivey. *The fUZZ manual*. Computing Science Consultancy, 2 Willow Close, Oxford, UK, 1988.
- [100] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, 1989.
- [101] C.B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, 1986.
- [102] Pete Steggles and Jason Hulance. Z tools survey. June 1994.
- [103] Andrew Martin. Approaches to proof in Z — or — why effective proof tool support for Z is hard. Technical Report 97-34, Software Verification Research Centre, School of Information Technology, The University of Queensland, Australia, November 1997.
- [104] W.T. Harwood. Proof rules for balzac. Technical Report WTH/P7/001, Imperial Software Technology, Cambridge, UK, 1991.
- [105] R.B. Jones. ICL ProofPower. *BCS-FACS*, 1(1):10–13, 1992.

- [106] I. Toyn. Formal reasoning in the Z notation using CADiZ. In N.A. Merriam, editor, *2nd International Workshop on User Interface Design for Theorem Proving Systems*, York, U.K., July 1996.
- [107] J.P. Bowen and M.J.C. Gordon. Z and HOL. In J.P. Bowen and J.A. Hall, editors, *Z User Workshop*, Workshops in Computing, pages 141–167, Cambridge, UK, 1994. Springer-Verlag.
- [108] J.P. Bowen and M.J.C. Gordon. A shallow embedding of Z in HOL. *Information and Software Technology*, 37(5–6):269–276, 1995.
- [109] Lawrence C. Paulson. Designing a theorem prover. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Background: Computational Structures*, volume 2 of *Handbook of Logic in Computer Science*, pages 415–475, Oxford, UK, 1992. Oxford Science Publications.
- [110] D. Cyrluk, S. Rajan, N. Shankar, and M.K. Srivas. Effective theorem proving for hardware verification. In Ramayya Jumar and Thomas Kropf, editors, *Theorem Provers in Circuit Design (TPCD '94)*, volume 910 of *Lecture Notes in Computer Science*, pages 203–222, Bad Herrenalb, Germany, September 1994. Springer-Verlag.
- [111] Patrick Lincoln and John Rushby. Formal verification of an interactive consistency algorithm for the Draper FTP architecture under a hybrid fault model. In *COMPASS '94 (Proceedings of the Ninth Annual Conference on Computer Assurance)*, pages 107–120, Gaithersburg, MD, June 1994. IEEE Washington section.
- [112] John Nicholls (Editor). Z notation (version 1.2). BSI Panel IST/5/-/19/2 (Z Notation), ISO Panel JTC1/SC22/WG19 (Rapporteur Group for Z), Z Standards Panel, September 1995.
- [113] B. Elspas, M. Green, M. Moriconi, and R. Shostak. A JOVIAL verifier. Technical report, Computer Science Laboratory, SRI International, January 1979.
- [114] L. Robinson, K.N. Levitt, and B.A. Silverberg. *The HDM Handbook*. Computer Science Laboratory, SRI International, Menlo Park, CA, June 1979. Three Volumes.

- [115] R.E. Shostak, R. Schwartz, and P.M. Melliar-Smith. STP: A mechanized logic for specification and verification. In D. Loveland, editor, *6th International Conference on Automated Deduction (CADE)*, volume 138 of *Lecture Notes in Computer Science*, New York, NY, 1982. Springer-Verlag.
- [116] John Rushby, Friedrich von Henke, and Sam Owre. An introduction to formal specification and verification using EHD. Technical Report SRI-CSL-91-2, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1991.
- [117] David Park. Finiteness is mu-ineffable. *Theoretical Computer Science*, 3:173–181, 1976.
- [118] S.B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.
- [119] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
- [120] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
- [121] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.
- [122] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. In *Proceedings of the Workshop on Industrial-Strength Formal Specification Techniques*, Baco Raton, Florida, April 1995.
- [123] S. Owre, N. Shankar, and J.M. Rushby. *The PVS Specification Language*. Computer Science Laboratory, SRI International, 1995.

- [124] S. Owre, N. Shankar, and J.M. Rushby. *User Guide for the PVS Specification and Verification System*. Computer Science Laboratory, SRI International, 1995.
- [125] Rosalind Barden, Susan Stepney, and David Cooper. *Z in Practice*. BCS Practitioners Series. Prentice Hall International, 1994.
- [126] Robert E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583–585, July 1978.
- [127] A.C. Leisenring. *Mathematical Logic and Hilbert's ϵ -symbol*. Gordon and Breach Science Publishers, New York, 1969.
- [128] Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *Proceedings of the Workshop on Industrial Strength Formal Specification Techniques (WIFT'95)*, Boca Raton, Florida, April 1995.
- [129] Xiaorong Huang and Armin Fiedler. Presenting machine-found proofs. In M.A. McRobbie and J.K. Slaney, editors, *CADE-13: 13th International Conference on Automated Deduction*, number 1104 in Lecture Notes in Computer Science, New Brunswick, NJ, 1996. Springer-Verlag.
- [130] Xiaorong Huang and Armin Fiedler. Proof presentation as an application of NLG. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI)*, Nagoya, Japan, August 1997.
- [131] Xiaorong Huang. Presenting machine-found proofs. In Philip Gray, editor, *User Interfaces for Theorem Provers (UITP '95)*, Glasgow, Scotland, July 1995.
- [132] Janet Bertot and Yves Bertot. The CtCoq experience. In N.A. Merriam, editor, *User Interfaces for Theorem Provers (UITP '96)*, pages 17–23, York, UK, July 1996.
- [133] Y. Coscoy. A natural language explanation for formal proofs. In C. Retoré, editor, *Proceedings of the International Conference on Logical Aspects of Computational Linguistics (LACL)*, volume 1328 of *Lecture Notes in Computer Science*, Nancy, September 1996. Springer-Verlag.

- [134] Y. Coscoy, G. Kahn, and L. Théry. Extracting text from proofs. In M. Denzani and G. Plotkin, editors, *Proceedings of the International Conference on Typed Lambda-Calculus and Applications (TLCA)*, volume 902 of *Lecture Notes in Computer Science*, Edinburgh, April 1995. Springer-Verlag.
- [135] Yann Coscoy, Gilles Kahn, and Laurent Théry. Extracting text from proofs. Rapport de Recherche 2459, INRIA, Sophia-Antipolis Cedex, France, January 1995.
- [136] B.A. Wichmann. Low-ada: An Ada validation tool. NPL Report CISE 144/89, National Physical Laboratory, April 1998.
- [137] Mark Saaltink. The Z/EVES system. In *ZUM '97: The Z Formal Specification Notation; 10th International Conference of Z Users*, number 1212 in *Lecture Notes in Computer Science*, pages 72–85, Reading, UK, April 1997. Springer-Verlag.
- [138] Sam Valentine. Inconsistency and undefinedness in Z — a practical guide. Draft Paper, November 1997.
- [139] Ricky W. Butler. An elementary tutorial on formal specification and verification using pvs 2. NASA Technical Memorandum 108991, NASA Langley Research Center, Hampton, VA, June 1993. Revised June 1995.
- [140] Piotr Rudnicki. An overview of the MIZAR project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, pages 311–330, Båstad, Sweden, June 1992.
- [141] Piotr Rudnicki. Seat reservation problem in MIZAR. Available from http://web.cs.ualberta.ca/~piotr/Mizar/FLT_DB.
- [142] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM TOPLAS*, 4(3):382–401, July 1982.
- [143] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.
- [144] W.R. Bevier and W.D. Young. Machine-checked proofs of a Byzantine agreement algorithm. Technical Report 55, Computational Logic Inc., Austin, TX, June 1990.

- [145] John Rushby. Formal verification of an oral messages algorithm for interactive consistency. Technical Report SRI-CSL-92-1, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1992. Also available as NASA Contractor Report 189704, October 1992.
- [146] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault Tolerant Computing Symposium 23*, pages 402–411, Toulouse, France, June 1993. IEEE Computer Society.
- [147] William D. Young. Comparing verification systems: Interactive consistency in ACL2. *IEEE Transactions on Software Engineering*, 23(4):214–223, April 1997.
- [148] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? Technical report, Digital Systems Research Center, May 1997.

Appendix A

A worked example of PVS

This appendix presents a worked example of a complete specification and proof in PVS. The purpose is to provide the reader unfamiliar with the notation and capability of PVS a basic grounding so that the fragments of specification and proof appearing in the main body of the thesis will be more readily understood.

A.1 Specification of GCD

The example we present in this section is the specification and proof of a simple algorithm to compute the greatest common divisor (GCD) of two positive natural numbers (`posnat` in PVS terminology). The algorithm is specified as:

```
i, j : VAR posnat

gcd(i, j): RECURSIVE posnat =
  IF i = j THEN i
    ELSIF i > j THEN gcd(i - j, j)
    ELSE gcd(i, j - i)
  ENDIF
MEASURE i + j
```

The GCD is computed by repeatedly subtracting the smaller argument from the larger argument until the arguments are the same. The recursion must

terminate, as the GCD function is required to be total — this is the purpose of the MEASURE function, which defines an expression which will reduce on each recursive call. Here, the sum of the arguments will always decrease.

A.2 Challenging the Specification

From our knowledge of GCD, argument order should not matter. We can prove that this is so with the following challenge:

```
gcd0: LEMMA gcd(i, j) = gcd(j, i)
```

Issuing the proof command to PVS we are presented with the following sequent:

```
gcd0 :
  |-----
  {1}   (FORALL (i: posnat, j: posnat): gcd(i, j) = gcd(j, i))
```

We need to prove this by induction, but on the sum of the arguments i and j . PVS provides a non-trivial induction schema with the command MEASURE-INDUCT. We issue this command and get:

```
Rule? (measure-induct "i+j" ("i" "j"))

Inducting on i+j,
this simplifies to:
gcd0 :
  |-----
  {1}   (FORALL (x_43: {i: nonneg_int | i > 0}),
          (x_44: {i: nonneg_int | i > 0}):
          (FORALL (y_45: {i: nonneg_int | i > 0}),
              (y_46: {i: nonneg_int | i > 0}):
              y_45 + y_46 < x_43 + x_44
              IMPLIES gcd(y_45, y_46) = gcd(y_46, y_45))
              IMPLIES gcd(x_43, x_44) = gcd(x_44, x_43))
```

Now, we need to eliminate the top level universal quantifier by skolemization. We do this with (SKOSIMP), which also performs disjunctive simplification:

```

Rule? (skosimp)

Skolemizing and flattening,
this simplifies to:
gcd0 :

{-1}   (FORALL (y_45: {i: nonneg_int | i > 0}),
        (y_46: {i: nonneg_int | i > 0})):
        y_45 + y_46 < x!1 + x!2 IMPLIES
        gcd(y_45, y_46) = gcd(y_46, y_45))
|-----
1      gcd(x!1, x!2) = gcd(x!2, x!1)

```

Now, we expand the definition of the GCD function in the consequent, to give:

```

Rule? (expand "gcd" +)

Expanding the definition of gcd,
this simplifies to:
gcd0 :

[-1]   (FORALL (y_45: {i: nonneg_int | i > 0}),
        (y_46: {i: nonneg_int | i > 0})):
        y_45 + y_46 < x!1 + x!2 IMPLIES
        gcd(y_45, y_46) = gcd(y_46, y_45))
|-----
{1}   (IF x!1 = x!2 THEN x!1
      ELSIF x!1 > x!2 THEN gcd(x!1 - x!2, x!2)
      ELSE gcd(x!1, x!2 - x!1)
      ENDIF
      = IF x!2 = x!1 THEN x!2
      ELSIF x!2 > x!1 THEN gcd(x!2 - x!1, x!1)
      ELSE gcd(x!2, x!1 - x!2)
      ENDIF)

```

This sequent contains an equality over two IF-THEN-ELSE expressions, so we ‘lift’ the conditional to the outermost level using (LIFT-IF):

Rule? (lift-if)

Lifting IF-conditions to the top level,
this simplifies to:

gcd0 :

```

[-1]   (FORALL (y_45: {i: nonneg_int | i > 0}),
        (y_46: {i: nonneg_int | i > 0})):
        y_45 + y_46 < x!1 + x!2 IMPLIES
        gcd(y_45, y_46) = gcd(y_46, y_45)
|-----
{1}   IF x!1 = x!2
      THEN
      (x!1
       = IF x!2 = x!1 THEN x!2
       ELSIF x!2 > x!1
         THEN gcd(x!2 - x!1, x!1)
       ELSE gcd(x!2, x!1 - x!2)
       ENDIF)
      ELSE IF x!1 > x!2
      THEN
      (gcd(x!1 - x!2, x!2)
       = IF x!2 = x!1 THEN x!2
       ELSIF x!2 > x!1
         THEN gcd(x!2 - x!1, x!1)
       ELSE gcd(x!2, x!1 - x!2)
       ENDIF)
      ELSE
      (gcd(x!1, x!2 - x!1)
       = IF x!2 = x!1 THEN x!2
       ELSIF x!2 > x!1 THEN gcd(x!2 - x!1, x!1)
       ELSE gcd(x!2, x!1 - x!2)
       ENDIF)
      ENDIF
      ENDIF

```

We can now see that we have three cases in the outermost conditional. When combined with the inductive case, these offer a proof of this lemma. Now

we've got to a point where we think the proof is more or less obvious, we apply one of the more 'heavyweight' proof steps, (`grind`), with rewriting turned off (we've already done enough rewriting by hand):

```
Rule? (GRIND :DEFS NIL)
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.

Run time = 6.48 secs.
Real time = 35.46 secs.
```

Having proved this 'challenge theorem' we have more confidence in our specification, in that a property we would naturally expect of the function has been proven to hold. We now turn our attention to the two fundamental properties of the GCD algorithm: that the result divides exactly into both arguments, and that it is the greatest of all such divisors.

A.3 Correctness Properties

A.3.1 GCD is divisible

The first property we require of our GCD algorithm is that its result is a divisor of both arguments. Firstly, we need to express the property that two numbers divide in such a manner:

```
divides?(i, j): bool = EXISTS (k:posnat): i*k=j
```

Using this definition we can express the divisibility condition as:

```
gcd_divides: THEOREM
  divides?(gcd(i, j), i) AND divides?(gcd(i, j), j)
```

The proof of this starts in the same manner as the previous one, by induction on i and j :

```

gcd_divides :
  |-----
{1}   (FORALL (i: posnat, j: posnat):
      divides?(gcd(i, j), i) AND divides?(gcd(i, j), j))

Rule? (MEASURE-INDUCT "i+j" ("i" "j"))
Inducting on i+j,
this simplifies to:
gcd_divides :
  |-----
{1}   (FORALL (x_53: {i: nonneg_int | i > 0}),
      (x_54: {i: nonneg_int | i > 0}):
      (FORALL (y_55: {i: nonneg_int | i > 0}),
      (y_56: {i: nonneg_int | i > 0}):
      y_55 + y_56 < x_53 + x_54
      IMPLIES divides?(gcd(y_55, y_56), y_55)
      AND divides?(gcd(y_55, y_56), y_56))
      IMPLIES divides?(gcd(x_53, x_54), x_53)
      AND divides?(gcd(x_53, x_54), x_54))

```

As before, we now skolemize away the universal quantifiers:

```

Rule? (SKOSIMP*)
Repeatedly Skolemizing and flattening,
this simplifies to:
gcd_divides :
{-1}  (FORALL (y_55: {i: nonneg_int | i > 0}),
      (y_56: {i: nonneg_int | i > 0}):
      y_55 + y_56 < x!1 + x!2
      IMPLIES divides?(gcd(y_55, y_56), y_55)
      AND divides?(gcd(y_55, y_56), y_56))
  |-----
{1}   divides?(gcd(x!1, x!2), x!1) AND
      divides?(gcd(x!1, x!2), x!2)

```

And then expand the definition of GCD in the consequent:

```
(EXPAND "gcd" +)
Expanding the definition of gcd,
this simplifies to:
gcd_divides :

[-1]   (FORALL (y_55: {i: nonneg_int | i > 0}),
        (y_56: {i: nonneg_int | i > 0}):
        y_55 + y_56 < x!1 + x!2
        IMPLIES divides?(gcd(y_55, y_56), y_55)
        AND divides?(gcd(y_55, y_56), y_56))
|-----
{1}   (divides?(IF x!1 = x!2 THEN x!1
              ELSIF x!1 > x!2 THEN gcd(x!1 - x!2, x!2)
              ELSE gcd(x!1, x!2 - x!1)
              ENDIF,
        x!1)
      AND
      divides?(IF x!1 = x!2 THEN x!1
              ELSIF x!1 > x!2 THEN gcd(x!1 - x!2, x!2)
              ELSE gcd(x!1, x!2 - x!1)
              ENDIF,
        x!2))
```

If we apply our ‘heavyweight’ hammer (`grind`) again at this point, the prover enters a rewriting loop:

```
Warning: Rewriting depth = 50; Rewriting with gcd
Warning: Rewriting depth = 100; Rewriting with gcd
Warning: Rewriting depth = 150; Rewriting with gcd
...
```

This has been caused by the heuristic instantiation giving the wrong instantiation to the inductive step. We interrupt the prover and back out of this step and then re-run (`grind`) with instantiations turned off. This gives six subgoals that we will now address in turn.

```

Rule? (grind :if-match nil)
divides? rewrites divides?(gcd(y_65, y_66), y_65)
  to EXISTS (k: posnat): gcd(y_65, y_66) * k = y_65
divides? rewrites divides?(gcd(y_65, y_66), y_66)
  to EXISTS (k: posnat): gcd(y_65, y_66) * k = y_66

divides? rewrites
  divides?(IF x!1 = x!2 THEN x!1
            ELSIF x!1 > x!2 THEN gcd(x!1 - x!2, x!2)
            ELSE gcd(x!1, x!2 - x!1)
            ENDIF,
            x!2)
  to EXISTS (k: posnat):
    IF x!1 = x!2 THEN x!1
    ELSIF x!1 > x!2 THEN gcd(x!1 - x!2, x!2)
    ELSE gcd(x!1, x!2 - x!1)
    ENDIF
    * k
    = x!2
divides? rewrites divides?(gcd(y_65, y_66), y_65)
  to EXISTS (k: posnat): gcd(y_65, y_66) * k = y_65
divides? rewrites divides?(gcd(y_65, y_66), y_66)
  to EXISTS (k: posnat): gcd(y_65, y_66) * k = y_66
divides? rewrites
  divides?(IF x!1 = x!2 THEN x!1
            ELSIF x!1 > x!2 THEN gcd(x!1 - x!2, x!2)
            ELSE gcd(x!1, x!2 - x!1)
            ENDIF,
            x!1)
  to EXISTS (k: posnat):
    IF x!1 = x!2 THEN x!1
    ELSIF x!1 > x!2 THEN gcd(x!1 - x!2, x!2)
    ELSE gcd(x!1, x!2 - x!1)
    ENDIF
    * k
    = x!1

```

Trying repeated skolemization, instantiation, and if-lifting,
 this yields 6 subgoals:

gcd_divides.1 :

```

{-1}   (FORALL (y_65: {i: nonneg_int | i > 0}),
        (y_66: {i: nonneg_int | i > 0}):
        y_65 + y_66 < 2 * x!2 IMPLIES
        (EXISTS (k: posnat): gcd(y_65, y_66) * k = y_65)
        AND EXISTS (k: posnat): gcd(y_65, y_66) * k = y_66)
{-2}   x!1 = x!2
      |-----
{1}    EXISTS (k: posnat): x!2 * k = x!2
  
```

This goal is trivial: we need to instantiate the existential quantifier in the consequent with 1.

Rule? (inst + 1)

Instantiating the top quantifier in + with the terms:

1,

this simplifies to:

gcd_divides.1 :

```

[-1]   (FORALL (y_65: {i: nonneg_int | i > 0}),
        (y_66: {i: nonneg_int | i > 0}):
        y_65 + y_66 < 2 * x!2 IMPLIES
        (EXISTS (k: posnat): gcd(y_65, y_66) * k = y_65)
        AND EXISTS (k: posnat): gcd(y_65, y_66) * k = y_66)
[-2]   x!1 = x!2
      |-----
{1}    x!2 * 1 = x!2
  
```

Multiplication by 1 is an identity, so we invoke the arithmetic decision procedures with (assert).

```

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of gcd_divides.1.

gcd_divides.2 :

{-1}   (FORALL (y_65: {i: nonneg_int | i > 0}),
        (y_66: {i: nonneg_int | i > 0})):
        y_65 + y_66 < x!1 + x!2 IMPLIES
        (EXISTS (k: posnat): gcd(y_65, y_66) * k = y_65)
        AND EXISTS (k: posnat): gcd(y_65, y_66) * k = y_66)
{-2}   x!1 > x!2
      |-----
{1}    x!1 = x!2
{2}    EXISTS (k: posnat): gcd(x!1 - x!2, x!2) * k = x!2

```

In this subgoal, clearly formula 2 follows from the inductive step in formula -1, with the instantiation $(x!1 - x!2)$, $x!2$. This is a simple enough matching that (`grind`) completes this branch automatically.

```

Rule? (GRIND)
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of gcd_divides.2.

gcd_divides.3 :

{-1}   (FORALL (y_65: {i: nonneg_int | i > 0}),
        (y_66: {i: nonneg_int | i > 0})):
        y_65 + y_66 < x!1 + x!2 IMPLIES
        (EXISTS (k: posnat): gcd(y_65, y_66) * k = y_65)
        AND EXISTS (k: posnat): gcd(y_65, y_66) * k = y_66)
      |-----
{1}    x!1 = x!2
{2}    x!1 > x!2
{3}    EXISTS (k: posnat): gcd(x!1, x!2 - x!1) * k = x!2

```

In this subgoal there is a correspondence here between the first conjunct on the right of the implication in formula -1 and the formula 3. We use (`inst?`)

to instantiate formula -1, and then (grind) (with instantiation turned off) to skolemize formula 3 and simplify formula -1.

```

Rule? (INST?)
Found substitution:
y_66 gets x!2 - x!1,
y_65 gets x!1,
Instantiating quantified variables,
this simplifies to:
gcd_divides.3 :

{-1}    x!1 + (x!2 - x!1) < x!1 + x!2 IMPLIES
        (EXISTS (k: posnat): gcd(x!1, x!2 - x!1) * k = x!1) AND
        EXISTS (k: posnat): gcd(x!1, x!2 - x!1) * k = x!2 - x!1
    |-----
[1]     x!1 = x!2
[2]     x!1 > x!2
[3]     EXISTS (k: posnat): gcd(x!1, x!2 - x!1) * k = x!2

Rule? (GRIND :IF-MATCH NIL)
Trying repeated skolemization, instantiation, and if-lifting,
this simplifies to:
gcd_divides.3 :

{-1}    k!2 > 0
{-2}    k!1 > 0
{-3}    gcd(x!1, x!2 - x!1) * k!1 = x!1
{-4}    gcd(x!1, x!2 - x!1) * k!2 = x!2 - x!1
    |-----
[1]     x!1 = x!2
[2]     x!1 > x!2
[3]     EXISTS (k: posnat): gcd(x!1, x!2 - x!1) * k = x!2

```

Now, we can see that formulas -3 and -4 match formula 3 with an instantiation of $k!1 + k!2$.

```

Rule? (INST + "k!1 + k!2")
Instantiating the top quantifier in + with the terms:
  k!1 + k!2,
this simplifies to:
gcd_divides.3 :

[-1]    k!2 > 0
[-2]    k!1 > 0
[-3]    gcd(x!1, x!2 - x!1) * k!1 = x!1
[-4]    gcd(x!1, x!2 - x!1) * k!2 = x!2 - x!1
  |-----
[1]     x!1 = x!2
[2]     x!1 > x!2
{3}     gcd(x!1, x!2 - x!1) * (k!1 + k!2) = x!2

```

Again, this is now a problem in linear arithmetic (treating the applications of gcd as uninterpreted function symbols), so we use (assert).

```

Rule? (ASSERT)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of gcd_divides.3.

gcd_divides.4 :

{-1}    (FORALL (y_65: {i: nonneg_int | i > 0}),
          (y_66: {i: nonneg_int | i > 0})):
          y_65 + y_66 < 2 * x!2 IMPLIES
          (EXISTS (k: posnat): gcd(y_65, y_66) * k = y_65)
          AND EXISTS (k: posnat): gcd(y_65, y_66) * k = y_66)
{-2}    x!1 = x!2
  |-----
{1}     EXISTS (k: posnat): x!2 * k = x!2

```

We have seen a similar goal to this earlier in the proof, and indeed the further two goals are also similar to those seen previously. Hence, we present the rest of the interaction without explanatory text.

```

Rerunning step: (INST 1 1)
Instantiating the top quantifier in 1 with the terms:
  1,
this simplifies to:
gcd_divides.4 :

[-1]   (FORALL (y_55: {i: nonneg_int | i > 0}),
        (y_56: {i: nonneg_int | i > 0})):
        y_55 + y_56 < 2 * x!2 IMPLIES
        (EXISTS (k: posnat): gcd(y_55, y_56) * k = y_55)
        AND EXISTS (k: posnat): gcd(y_55, y_56) * k = y_56)
[-2]   x!1 = x!2
      |-----
{1}    x!2 * 1 = x!2

Rerunning step: (ASSERT)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of gcd_divides.4.

```

```

gcd_divides.5 :

{-1}   (FORALL (y_55: {i: nonneg_int | i > 0}),
        (y_56: {i: nonneg_int | i > 0})):
        y_55 + y_56 < x!1 + x!2 IMPLIES
        (EXISTS (k: posnat): gcd(y_55, y_56) * k = y_55)
        AND EXISTS (k: posnat): gcd(y_55, y_56) * k = y_56)
{-2}   x!1 > x!2
      |-----
{1}    x!1 = x!2
{2}    EXISTS (k: posnat): gcd(x!1 - x!2, x!2) * k = x!1

```

```

Rerunning step: (INST?)
Found substitution:
y_56 gets x!2,
y_55 gets x!1 - x!2,
Instantiating quantified variables,
this simplifies to:
gcd_divides.5 :

{-1}    x!1 - x!2 + x!2 < x!1 + x!2 IMPLIES
        (EXISTS (k: posnat): gcd(x!1 - x!2, x!2) * k = x!1 - x!2)
        AND EXISTS (k: posnat): gcd(x!1 - x!2, x!2) * k = x!2
[-2]    x!1 > x!2
        |-----
[1]     x!1 = x!2
[2]     EXISTS (k: posnat): gcd(x!1 - x!2, x!2) * k = x!1

```

```

Rerunning step: (GRIND :IF-MATCH NIL)
Trying repeated skolemization, instantiation, and if-lifting,
this simplifies to:
gcd_divides.5 :

{-1}    k!2 > 0
{-2}    k!1 > 0
{-3}    gcd(x!1 - x!2, x!2) * k!1 = x!1 - x!2
{-4}    gcd(x!1 - x!2, x!2) * k!2 = x!2
[-5]    x!1 > x!2
        |-----
[1]     x!1 = x!2
[2]     EXISTS (k: posnat): gcd(x!1 - x!2, x!2) * k = x!1

```

```

Rerunning step: (INST + "k!1+k!2")
Instantiating the top quantifier in + with the terms:
  k!1+k!2,
this simplifies to:
gcd_divides.5 :

[-1]    k!2 > 0
[-2]    k!1 > 0
[-3]    gcd(x!1 - x!2, x!2) * k!1 = x!1 - x!2
[-4]    gcd(x!1 - x!2, x!2) * k!2 = x!2
[-5]    x!1 > x!2
  |-----
[1]     x!1 = x!2
{2}    gcd(x!1 - x!2, x!2) * (k!1 + k!2) = x!1

Rerunning step: (ASSERT)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of gcd_divides.5.

```

```

gcd_divides.6 :

{-1}   (FORALL (y_55: {i: nonneg_int | i > 0}),
        (y_56: {i: nonneg_int | i > 0})):
        y_55 + y_56 < x!1 + x!2 IMPLIES
        (EXISTS (k: posnat): gcd(y_55, y_56) * k = y_55)
        AND EXISTS (k: posnat): gcd(y_55, y_56) * k = y_56)
  |-----
{1}    x!1 = x!2
{2}    x!1 > x!2
{3}    EXISTS (k: posnat): gcd(x!1, x!2 - x!1) * k = x!1

Rerunning step: (GRIND)
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of gcd_divides.6.

Q.E.D.

Run time = 32.46 secs.
Real time = 86.14 secs.

```

A.3.2 GCD is greatest

The theorem that GCD is the greatest divisor of its arguments is stated as follows:

```

gcd_greatest: THEOREM
  FORALL (z:posnat):
    divides?(z, i) AND divides?(z, j)
    IMPLIES divides?(z, gcd(i, j))

```

This states that all positive (i.e. non zero) natural numbers who divide both arguments will also divide the GCD of the arguments. If there was a positive natural who divides both arguments, but does not divide the GCD, then it would be larger than the GCD — hence the GCD computed would not be the greatest.

The proof of this theorem begins with induction again:

```
gcd_greatest :  
  
  |-----  
{1}   FORALL (i: posnat, j: posnat), (z: posnat):  
      divides?(z, i) AND divides?(z, j) IMPLIES  
      divides?(z, gcd(i, j))  
  
Rule? (MEASURE-INDUCT "i+j" ("i" "j"))  
Inducting on i+j,  
this simplifies to:  
gcd_greatest :  
  
  |-----  
{1}   (FORALL (x_73: {i: nonneg_int | i > 0}),  
      (x_74: {i: nonneg_int | i > 0}):  
      (FORALL (y_75: {i: nonneg_int | i > 0}),  
      (y_76: {i: nonneg_int | i > 0}):  
      y_75 + y_76 < x_73 + x_74  
      IMPLIES FORALL (z: posnat):  
      divides?(z, y_75) AND divides?(z, y_76)  
      IMPLIES divides?(z, gcd(y_75, y_76)))  
      IMPLIES FORALL (z: posnat):  
      divides?(z, x_73) AND divides?(z, x_74)  
      IMPLIES divides?(z, gcd(x_73, x_74)))
```

Again, we skolemize the universal quantifier.

Rule? (skosimp*)

Repeatedly Skolemizing and flattening,

this simplifies to:

gcd_greatest :

```
{-1}  (FORALL (y_75: {i: nonneg_int | i > 0}),
        (y_76: {i: nonneg_int | i > 0})):
      y_75 + y_76 < x!1 + x!2
      IMPLIES FORALL (z: posnat):
        divides?(z, y_75) AND divides?(z, y_76)
        IMPLIES divides?(z, gcd(y_75, y_76))
{-2}  divides?(z!1, x!1)
{-3}  divides?(z!1, x!2)
      |-----
{1}   divides?(z!1, gcd(x!1, x!2))
```

Now, we may think that formulas -2 and -3 imply formula 1 by the inductive hypothesis in -1. However, if you work out the instantiation, we would get the left hand side of the implication reducing to $x!1 + x!2 < x!1 + x!2$ which is clearly false. So, we must expand the definition of GCD in the consequent, simplify, and we are presented with two subgoals.

Rule? (expand "gcd" +)

Expanding the definition of gcd,

this simplifies to:

gcd_greatest :

```
[-1]  (FORALL (y_75: {i: nonneg_int | i > 0}),
        (y_76: {i: nonneg_int | i > 0})):
      y_75 + y_76 < x!1 + x!2
      IMPLIES FORALL (z: posnat):
        divides?(z, y_75) AND divides?(z, y_76)
        IMPLIES divides?(z, gcd(y_75, y_76))
[-2]  divides?(z!1, x!1)
[-3]  divides?(z!1, x!2)
      |-----
{1}   divides?(z!1,
        IF x!1 = x!2 THEN x!1
        ELSIF x!1 > x!2 THEN gcd(x!1 - x!2, x!2)
        ELSE gcd(x!1, x!2 - x!1)
        ENDIF)
```

Now simplifying formula 1 using (smash).

```

Rule? (smash)
Repeatedly simplifying with BDDs, decision procedures, rewriting,
and if-lifting,
this yields 2 subgoals:
gcd_greatest.1 :

[-1]   (FORALL (y_75: {i: nonneg_int | i > 0}),
        (y_76: {i: nonneg_int | i > 0}):
        y_75 + y_76 < x!1 + x!2
        IMPLIES FORALL (z: posnat):
        divides?(z, y_75) AND divides?(z, y_76)
        IMPLIES divides?(z, gcd(y_75, y_76)))
[-2]   divides?(z!1, x!1)
[-3]   divides?(z!1, x!2)
{-4}   x!1 > x!2
      |-----
{1}    x!1 = x!2
{2}    divides?(z!1, gcd(x!1 - x!2, x!2))

```

From formula -1 we know that any number which divides into $x!1$ and $x!2$ will divide into their GCD. From formulas -2 and -3 we know that $z!1$ divides into both $x!1$ and $y!1$, and it is our goal (formula 2) that $z!1$ divides into the GCD of $x!1 - x!2$. But, as we also know that $x!1 > x!2$ (from formula -5) then $z!1$ will also divide into $x!1 - x!2$. We make this explicit in a lemma¹ which will prove later.

```

divides_lemma: LEMMA
  divides?(z, i) AND divides?(z, j) AND i>j
  IMPLIES divides?(z, i-j)

```

Now, using this lemma in our proof we get:

¹It is possible to introduce lemmata on the fly during a PVS proof, deferring their proof until later.

```

Rule? (use "divides_lemma")

Using lemma divides_lemma,
this simplifies to:
gcd_greatest.1 :

{-1}   divides?(z!1, x!1) AND divides?(z!1, x!2)
        AND x!1 > x!2 IMPLIES divides?(z!1, x!1 - x!2)
[-2]   (FORALL (y_12: {i: nonneg_int | i > 0}),
        (y_13: {i: nonneg_int | i > 0})):
        y_12 + y_13 < x!1 + x!2
        IMPLIES FORALL (z: posnat):
        divides?(z, y_12) AND divides?(z, y_13)
        IMPLIES divides?(z, gcd(y_12, y_13)))
[-3]   divides?(z!1, x!1)
[-4]   divides?(z!1, x!2)
[-5]   x!1 > x!2
      |-----
[1]    x!1 = x!2
[2]    divides?(z!1, gcd(x!1 - x!2, x!2))

```

This proof is now trivial, and we discharge it with (grind) with rewriting turned off.

```

Rule? (grind :defs nil)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of gcd_greatest.1.

```

The other subgoal from the (smash) is the same, but with $x!1$ less than $x!2$. It yields to the same proof.

```
gcd_greatest.2 :
```

```
[-1] (FORALL (y_12: {i: nonneg_int | i > 0}),
      (y_13: {i: nonneg_int | i > 0})):
      y_12 + y_13 < x!1 + x!2
      IMPLIES FORALL (z: posnat):
      divides?(z, y_12) AND divides?(z, y_13)
      IMPLIES divides?(z, gcd(y_12, y_13)))
[-2] divides?(z!1, x!1)
[-3] divides?(z!1, x!2)
|-----
{1} x!1 = x!2
{2} x!1 > x!2
{3} divides?(z!1, gcd(x!1, x!2 - x!1))
```

```
Rule? (use "divides_lemma")
```

```
Using lemma divides_lemma,
this simplifies to:
```

```
gcd_greatest.2 :
```

```
{-1} divides?(z!1, x!2) AND divides?(z!1, x!1)
      AND x!2 > x!1 IMPLIES divides?(z!1, x!2 - x!1)
[-2] (FORALL (y_12: {i: nonneg_int | i > 0}),
      (y_13: {i: nonneg_int | i > 0})):
      y_12 + y_13 < x!1 + x!2
      IMPLIES FORALL (z: posnat):
      divides?(z, y_12) AND divides?(z, y_13)
      IMPLIES divides?(z, gcd(y_12, y_13)))
[-3] divides?(z!1, x!1)
[-4] divides?(z!1, x!2)
|-----
[1] x!1 = x!2
[2] x!1 > x!2
[3] divides?(z!1, gcd(x!1, x!2 - x!1))
```

```
Rule? (grind :defs nil )
```

```
Trying repeated skolemization, instantiation, and if-lifting,
```

```
This completes the proof of gcd_greatest.2.
```

```
Q.E.D.
```

```
Run time = 30.34 secs.
```

```
Real time = 46.93 secs.
```

A.3.3 Divides Lemma

In the proof that the GCD specification does produce the greatest of divisors, we used the following lemma:

```
divides_lemma: LEMMA
  divides?(z, i) AND divides?(z, j) AND i>j
  IMPLIES divides?(z, i-j)
```

This lemma is a simple problem in linear arithmetic. We begin by using (`grind`), but this generates an incorrect instantiation, so we turn instantiations off.

```
divides_lemma :
  |-----
  {1}   (FORALL (i: posnat, j: posnat, z: posnat):
        divides?(z, i) AND divides?(z, j)
        AND i > j IMPLIES divides?(z, i - j))
```

```

Rule? (grind :if-match nil)

divides? rewrites divides?(z, i)
  to EXISTS (k: posnat): z * k = i
divides? rewrites divides?(z, j)
  to EXISTS (k: posnat): z * k = j
divides? rewrites divides?(z, i - j)
  to EXISTS (k: posnat): z * k = i - j

Trying repeated skolemization, instantiation, and if-lifting,
this simplifies to:
divides_lemma :

{-1}    k!2 > 0
{-2}    k!1 > 0
{-3}    i!1 > 0
{-4}    j!1 > 0
{-5}    z!1 > 0
{-6}    z!1 * k!1 = i!1
{-7}    z!1 * k!2 = j!1
{-8}    i!1 > j!1
|-----
{1}    EXISTS (k: posnat): z!1 * k = i!1 - j!1

```

Now, we perform the instantiation manually. Inspecting formulas -6, -7 and 1 we can determine the correct instantiation is $k!1 = k!2$.

```
Rule? (Inst + "k!1 - k!2")
```

```
Instantiating the top quantifier in + with the terms:
```

```
  k!1 - k!2,
```

```
this yields 2 subgoals:
```

```
divides_lemma.1 :
```

```
[-1]    k!2 > 0
```

```
[-2]    k!1 > 0
```

```
[-3]    i!1 > 0
```

```
[-4]    j!1 > 0
```

```
[-5]    z!1 > 0
```

```
[-6]    z!1 * k!1 = i!1
```

```
[-7]    z!1 * k!2 = j!1
```

```
[-8]    i!1 > j!1
```

```
  |-----
```

```
{1}    z!1 * (k!1 - k!2) = i!1 - j!1
```

We now use (assert) to apply the decision procedures, which completes the proof of this subgoal.

```
Rule? (assert)
```

```
Simplifying, rewriting, and recording with decision procedures,
```

```
This completes the proof of divides_lemma.1.
```

```
divides_lemma.2 (TCC):
```

```
[-1]    k!2 > 0
```

```
[-2]    k!1 > 0
```

```
[-3]    i!1 > 0
```

```
[-4]    j!1 > 0
```

```
[-5]    z!1 > 0
```

```
[-6]    z!1 * k!1 = i!1
```

```
[-7]    z!1 * k!2 = j!1
```

```
[-8]    i!1 > j!1
```

```
  |-----
```

```
{1}    k!1 - k!2 >= 0 AND k!1 - k!2 > 0
```

This goal requires the use of a lemma from the PVS prelude which relates the cases in which multiplication can yield negative answers (effectively what we're trying to show in formula 1).

```
pos_times_lt: LEMMA
  0 < x * y IFF (0 < x AND 0 < y) OR (x < 0 AND y < 0)
```

We now apply this lemma, and instantiate it according to formula 1.

```
Rule? (lemma "pos_times_lt")

Applying pos_times_lt
this simplifies to:
divides_lemma.2 :

{-1} (FORALL (x: real, y: real):
      0 < x * y IFF (0 < x AND 0 < y) OR (x < 0 AND y < 0))
[-2] k!2 > 0
[-3] k!1 > 0
[-4] i!1 > 0
[-5] j!1 > 0
[-6] z!1 > 0
[-7] z!1 * k!1 = i!1
[-8] z!1 * k!2 = j!1
[-9] i!1 > j!1
    |-----
[1]  k!1 - k!2 >= 0 AND k!1 - k!2 > 0
```

```
Rule? (inst - "k!2 - k!1" "z!1")
```

```
Instantiating the top quantifier in - with the terms:
```

```
k!2 - k!1, z!1,
```

```
this simplifies to:
```

```
divides_lemma.2 :
```

```
{-1}    0 < (k!2 - k!1) * z!1 IFF  
        (0 < k!2 - k!1 AND 0 < z!1)  
        OR (k!2 - k!1 < 0 AND z!1 < 0)
```

```
[-2]    k!2 > 0
```

```
[-3]    k!1 > 0
```

```
[-4]    i!1 > 0
```

```
[-5]    j!1 > 0
```

```
[-6]    z!1 > 0
```

```
[-7]    z!1 * k!1 = i!1
```

```
[-8]    z!1 * k!2 = j!1
```

```
[-9]    i!1 > j!1
```

```
|-----
```

```
[1]    k!1 - k!2 >= 0 AND k!1 - k!2 > 0
```

Now the subgoal is a problem in propositional reasoning and arithmetic, so we use (ground):

```
Rerunning step: (GROUND)
```

```
Applying propositional simplification and decision procedures,
```

```
This completes the proof of divides_lemma.2.
```

```
Q.E.D.
```

```
Run time = 3.35 secs.
```

```
Real time = 5.34 secs.
```

Appendix B

Functional Relations Specification

This specification of functions as a subtype of relations is a slightly modified version of that presented in a SRI technical report I co-authored with John Rushby — “A less elementary tutorial for the PVS specification and verification system”[2].

```
rel_as_fun[A : TYPE, B : TYPE] : THEORY
  BEGIN

  a, b : VAR A

  x, y : VAR B

  rel : TYPE = pred[[A, B]]

  R : VAR rel

  domain(R) : setof[A] = {a | ∃ x : R(a, x)}
```

```

range(R) : setof[B] = {x | ∃ a : R(a, x)}

functional(R) : bool = ∀ a, x, y : R(a, x) ∧ R(a, y) ⇒ x = y

injective(R) : bool = ∀ a, b, x : R(a, x) ∧ R(b, x) ⇒ a = b

part_inj(R) : bool = functional(R) ∧ injective(R)

null_inj : (part_inj) = ∅[[A, B]]

reldel_1(R : (part_inj), a) : (part_inj) = {(b, y) | R(b, y) ∧ a ≠ b}

reldel_2(R : (part_inj), x) : (part_inj) = {(b, y) | R(b, y) ∧ x ≠ y}

apply(R : (functional) a : (domain(R))) : (range(R)) =
  choose! (x : (range(R))) : R(a, x)

apply_lemma : LEMMA
  R(a, x) ⊃ apply(R, a) = x

invapply(R : (part_inj), x : (range(R))) : (domain(R)) =
  choose! (a : (domain(R))) : R(a, x)

override(R, S : (functional)) : (functional) =
  (S ∪ {(a, x) | ¬(a ∈ domain(S)) ∧ ((a, x) ∈ R)})

replace(R : (functional), a : (domain(R)), x : B) : (functional) =
  ({(b, y) | (R(b, y) ∧ a ≠ b)} ∪ {(a, x)})

update_ok : LEMMA
  LET newR = (R ∪ {(a, x)})
  IN part_inj(R) ∧ ¬(a ∈ domain(R)) ∧ ¬(x ∈ range(R)) ⊃
    part_inj(newR) ∧ apply(newR, a) = x ∧ invapply(newR, x) = a

END rel_as_fun

```

Appendix C

Z types, functions and domains in the DCC method

This section provides a quick reference guide to the types, functions and domains used in the Z presentation of the DCC method. It follows the structure of Stepney's book[3] which should be used as the definitive reference.

The semantic functions follow a naming convention where the first letter represents the type of function and the second letter represents the objects to which it is applied:

\mathcal{D}	Declaration before use semantics
\mathcal{T}	Type checking semantics
\mathcal{U}	Initialization before use semantics
\mathcal{M}	Dynamic semantics
\mathcal{O}	Operational semantics (compiler templates)
\mathcal{D}	Declarations
\mathcal{U}	Unary Expressions
\mathcal{B}	Binary Expressions
\mathcal{E}	Expressions
\mathcal{C}	Tosca Commands
\mathcal{I}	Aida Instructions
\mathcal{P}	Tosca Programs
\mathcal{A}	Aida Programs

C.1 Tosca — States and Environments

Check status:

$$CHECK ::= checkOK \mid checkWrong$$

$$_ \bowtie _ : CHECK \times CHECK \rightarrow CHECK$$

Memory locations:

$$Locn == \mathbb{Z}$$

Declaration before use semantics:

$$Env_D == NAME \mapsto CHECK$$

Type checking semantics:

$$TYPE ::= typeInteger \\ \mid typeBoolean \\ \mid typeWrong$$

$$Env_T == NAME \mapsto TYPE$$

Initialization before use semantics:

$$Store_U == Locn \mapsto CHECK$$

$$State_U == Store_U \times CHECK$$

$$storeOf_U == first[Store_U, CHECK]$$

$$checkOf_U == second[Store_U, CHECK]$$

$$_ \boxplus_v _ : State_U \times Store_U \rightarrow State_U$$

$$updateUse_U : CHECK \rightarrow State_U \rightarrow State_U$$

$$Env_U == NAME \mapsto Locn$$

$$worseStore : Store_U \times Store_U \rightarrow Store_U$$

$$worseState : State_U \times State_U \rightarrow State_U$$

Dynamic Semantics:

$$\textit{Store} ::= \textit{Locn} \mapsto \textit{VALUE}$$
$$\textit{Input} ::= \textit{seq Integer}$$
$$\textit{Output} ::= \textit{seq Integer}$$
$$\textit{State} ::= \textit{Store} \times \textit{Input} \times \textit{Output}$$
$$\textit{storeOf} : \textit{State} \rightarrow \textit{Store}$$
$$\textit{outOf} : \textit{State} \rightarrow \textit{Output}$$
$$_ \boxplus _ : \textit{State} \times \textit{Store} \rightarrow \textit{State}$$
$$\textit{Env} ::= \textit{NAME} \mapsto \textit{Locn}$$

C.2 Tosca — Semantics

Declarations:

$$\textit{DECL} ::= \textit{declVar} \langle \langle \textit{NAME} \times \textit{TYPE} \rangle \rangle$$
$$\mathcal{D}_D[-] : \textit{DECL} \rightarrow \textit{Env}_D \rightarrow \textit{Env}_D$$
$$\mathcal{T}_D[-] : \textit{DECL} \mapsto \textit{Env}_T \rightarrow \textit{Env}_T$$
$$\mathcal{U}_D[-] : \textit{DECL} \mapsto \textit{Env}_U \rightarrow \textit{Env}_U$$
$$\mathcal{M}_D[-] : \textit{DECL} \mapsto \textit{Env} \rightarrow \textit{Env}$$
$$\mathcal{D}_{D^*}[-] : \textit{seq DECL} \rightarrow \textit{Env}_D \rightarrow \textit{Env}_D$$
$$\mathcal{T}_{D^*}[-] : \textit{seq DECL} \mapsto \textit{Env}_T \rightarrow \textit{Env}_T$$
$$\mathcal{U}_{D^*}[-] : \textit{seq DECL} \mapsto \textit{Env}_U \rightarrow \textit{Env}_U$$
$$\mathcal{M}_{D^*}[-] : \textit{seq DECL} \mapsto \textit{Env} \rightarrow \textit{Env}$$

Operators:

$$\begin{aligned} UNY_ARITH_OP & ::= \text{negate} \\ UNY_LOGIC_OP & ::= \text{not} \\ UNY_OP & ::= \text{unyArithOp}\langle\langle UNY_ARITH_OP \rangle\rangle \\ & \quad | \text{unyLogicOp}\langle\langle UNY_LOGIC_OP \rangle\rangle \end{aligned}$$

$$\begin{aligned} BIN_ARITH_OP & ::= \text{plus} \mid \text{minus} \\ BIN_COMP_OP & ::= \text{less} \mid \text{greater} \mid \text{equal} \\ BIN_LOGIC_OP & ::= \text{or} \mid \text{and} \\ BIN_OP & ::= \text{binArithOp}\langle\langle BIN_ARITH_OP \rangle\rangle \\ & \quad | \text{binCompOp}\langle\langle BIN_COMP_OP \rangle\rangle \\ & \quad | \text{binLogicOp}\langle\langle BIN_LOGIC_OP \rangle\rangle \end{aligned}$$

$$\begin{aligned} \mathcal{T}_U[-] & : UNY_OP \rightarrow TYPE \rightarrow TYPE \\ \mathcal{T}_B[-] & : BIN_OP \rightarrow TYPE \times TYPE \rightarrow TYPE \\ \mathcal{M}_U[-] & : UNY_OP \rightarrow VALUE \rightarrow VALUE \\ \mathcal{M}_B[-] & : BIN_OP \rightarrow VALUE \times VALUE \rightarrow VALUE \end{aligned}$$

Expressions:

$$\begin{aligned} EXPR & ::= \text{const}\langle\langle VALUE \rangle\rangle \\ & \quad | \text{var}\langle\langle NAME \rangle\rangle \\ & \quad | \text{unyExpr}\langle\langle UNY_OP \times EXPR \rangle\rangle \\ & \quad | \text{binExpr}\langle\langle EXPR \times BIN_OP \times EXPR \rangle\rangle \end{aligned}$$

$$\begin{aligned} \mathcal{D}_E[-] & : EXPR \rightarrow Env_D \rightarrow CHECK \\ \mathcal{T}_E[-] & : EXPR \rightarrow Env_T \rightarrow TYPE \\ \mathcal{U}_E[-] & : EXPR \rightarrow Env_U \rightarrow State_U \rightarrow State_U \\ \mathcal{M}_E[-] & : EXPR \rightarrow Env \rightarrow State \rightarrow VALUE \end{aligned}$$

Commands:

$$\begin{aligned} CMD & ::= \text{block}\langle\langle \text{seq}_1 CMD \rangle\rangle \\ & \quad | \text{skip} \\ & \quad | \text{assign}\langle\langle NAME \times EXPR \rangle\rangle \\ & \quad | \text{choice}\langle\langle EXPR \times CMD \times CMD \rangle\rangle \\ & \quad | \text{loop}\langle\langle EXPR \times CMD \rangle\rangle \\ & \quad | \text{input}\langle\langle NAME \rangle\rangle \\ & \quad | \text{output}\langle\langle EXPR \rangle\rangle \end{aligned}$$

$$\begin{aligned}
\mathcal{D}_C[-] &: \text{CMD} \rightarrow \text{Env}_D \rightarrow \text{CHECK} \\
\mathcal{D}_{C^*}[-] &: \text{seq CMD} \rightarrow \text{Env}_D \rightarrow \text{CHECK} \\
\mathcal{T}_C[-] &: \text{CMD} \mapsto \text{Env}_T \mapsto \text{CHECK} \\
\mathcal{T}_{C^*}[-] &: \text{seq CMD} \mapsto \text{Env}_T \mapsto \text{CHECK} \\
\mathcal{U}_C[-] &: \text{CMD} \mapsto \text{Env}_U \mapsto \text{State}_U \mapsto \text{State}_U \\
\mathcal{U}_{C^*}[-] &: \text{seq CMD} \mapsto \text{Env}_U \mapsto \text{State}_U \mapsto \text{State}_U \\
\mathcal{M}_C[-] &: \text{CMD} \mapsto \text{Env} \mapsto \text{State} \mapsto \text{State} \\
\mathcal{M}_{C^*}[-] &: \text{seq CMD} \mapsto \text{Env} \mapsto \text{State} \mapsto \text{State}
\end{aligned}$$

Programs:

$$\text{PROG} ::= \text{Tosca} \langle\langle \text{seq DECL} \times \text{CMD} \rangle\rangle$$

$$\begin{aligned}
\mathcal{D}_P[-] &: \text{PROG} \rightarrow \text{CHECK} \\
\mathcal{T}_P[-] &: \text{PROG} \mapsto \text{CHECK} \\
\mathcal{U}_P[-] &: \text{PROG} \mapsto \text{CHECK} \\
\mathcal{M}_P[-] &: \text{PROG} \mapsto \text{Input} \mapsto \text{Output}
\end{aligned}$$

C.3 Aida — The Target Language

Labels and Instructions:

$$\text{Label} == \mathbb{N}$$

$$\begin{array}{l}
\text{INSTR} ::= \text{goto} \langle\langle \text{Label} \rangle\rangle \\
\quad | \text{jump} \langle\langle \text{Label} \rangle\rangle \\
\quad | \text{label} \langle\langle \text{Label} \rangle\rangle \\
\quad | \text{loadConst} \langle\langle \text{VALUE} \rangle\rangle \\
\quad | \text{loadVar} \langle\langle \text{Locn} \rangle\rangle \\
\quad | \text{store} \langle\langle \text{Locn} \rangle\rangle \\
\quad | \text{unyOp} \langle\langle \text{UNY_OP} \rangle\rangle \\
\quad | \text{binOp} \langle\langle \text{BIN_OP} \times \text{Locn} \rangle\rangle \\
\quad | \text{input} \\
\quad | \text{output}
\end{array}$$

$$\text{AIDA_PROG} ::= \text{Aida} \langle\langle \text{seq INSTR} \rangle\rangle$$

Aida's domains:

$$\begin{aligned} Store_I &== Locn \rightarrow VALUE \\ State_I &== Store_I \times Input \times Output \\ A &: Locn \\ storeOf_I &: State_I \rightarrow Store_I \\ outOf_I &: State_I \rightarrow Output \\ \\ Cont &== State_I \rightarrow State_I \\ Env_I &== Label \rightarrow Cont \end{aligned}$$

Dynamic Semantics:

$$\begin{aligned} \mathcal{M}_I[-] &: INSTR \rightarrow Env_I \rightarrow Cont \rightarrow State_I \rightarrow State_I \\ \mathcal{M}_{I^*}[-] &: \text{seq } INSTR \rightarrow Env_I \rightarrow Cont \rightarrow State_I \rightarrow State_I \\ \\ \mathcal{M}_A[-] &: AIDA_PROG \rightarrow Input \rightarrow Output \end{aligned}$$

C.4 Operational Semantics

Translation Environment:

$$\begin{aligned} Env_O &== NAME \rightarrow (Locn \setminus \{A\}) \\ top &: Locn \end{aligned}$$

Declarations:

$$\begin{aligned} \mathcal{O}_D \langle - \rangle &: DECL \rightarrow Env_O \rightarrow Env_O \\ \mathcal{O}_{D^*} \langle - \rangle &: \text{seq } DECL \rightarrow Env_O \rightarrow Env_O \end{aligned}$$

Expressions:

$$\mathcal{O}_E \langle - \rangle : EXPR \rightarrow Env_O \rightarrow Locn \rightarrow \text{seq } INSTR$$

Commands:

$$\begin{aligned} \mathcal{O}_C \langle - \rangle &: CMD \rightarrow Env_O \rightarrow Label \rightarrow (Label \times \text{seq } INSTR) \\ \mathcal{O}_{C^*} \langle - \rangle &: \text{seq } CMD \rightarrow Env_O \rightarrow Label \rightarrow (Label \times \text{seq } INSTR) \end{aligned}$$
$$\begin{aligned} labelOf &== first[Label, \text{seq } INSTR] \\ instrOf &== second[Label, \text{seq } INSTR] \end{aligned}$$

Programs:

$$\mathcal{O}_P \langle - \rangle : \text{PROG} \rightarrow \text{AIDA_PROG}$$