A Verified Vista Implementation
Final Report

Paul Curzon

Technical Report 311
September 1993

University of Cambridge
Computer Laboratory
New Museums Site
Pembroke Street
Cambridge
CB2 3QG
United Kingdom

Contents

Chapter 1

Introduction

Software is increasingly being used in safety-critical systems where correctness is of paramount importance. Traditionally, testing has been the main validation method used for software. The software is run on a selection of input values and the results obtained are checked for correctness. Unfortunately, the complexity of even small systems precludes testing on all input values. Only a small fraction of the possible values can be tested. Thus, errors can easily go undetected. Formal verification has been advocated as a more reliable way of checking that systems are correct. In this approach, mathematical techniques are used to prove that correctness properties hold of the software for *all* possible input values. Since errors are more easily made when writing programs in machine-code, higher-level languages are preferable. Consequently, much effort has been invested in using formal verification to validate high-level programs. However, high-level programs are compiled before being executed and it is the object code rather than the source program which must be correct. Proving that the source program satisfies a specification is not sufficient. We really wish to know that the object code satisfies the specification. As recent standards for safety critical software [41] have recognised, this problem must be addressed when high dependability is required.

Verified object code can be obtained in several ways as Figure 1.1 illustrates.

- The program can be written and validated in the object language. The advantages of high-level programs are lost. Mistakes are easy to make and hard to find. Formal verification of low-level code is much harder than of high-level programs.

- The program can be written in a high-level language, but validation performed on the compiled code. This is the normal procedure when testing is the validation method used. However, as noted above, it is harder to perform formal verification on object code than on a high-level program. The situation is made worse here because the object code is machine-generated. Understanding why it is supposed to be correct, a prerequisite for formal verification, is much harder. This is especially so if an optimising compiler is used.

- The source program can be formally shown to be equivalent to the

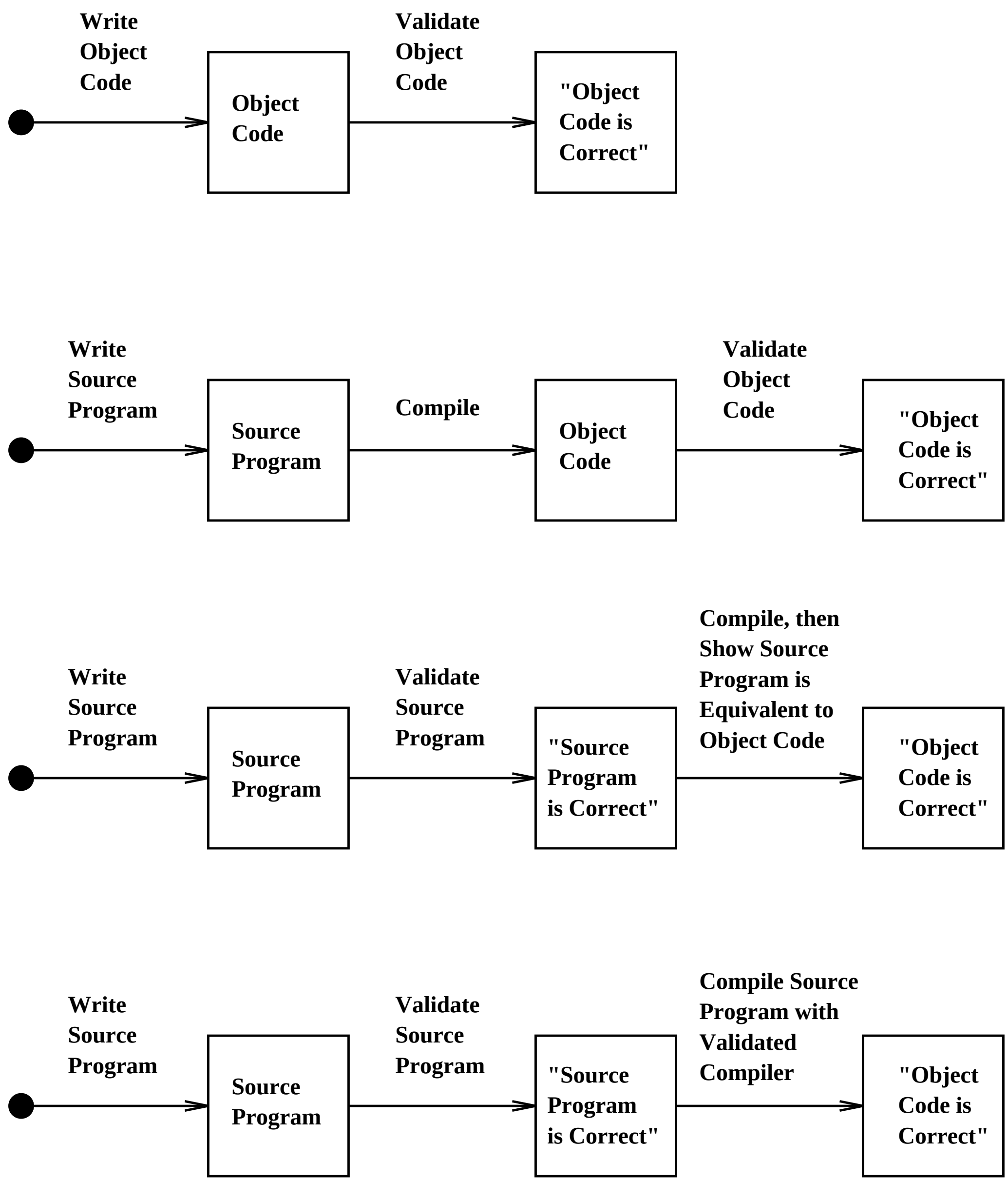

Figure 1.1: Alternate ways of Obtaining Validated Object Code

compiled code in a one-off proof. Validation can then be performed on the source program as the result is applicable to the object code. This approach has the disadvantage that, in addition to a correctness proof, an equivalence proof must be performed for each program. This proof must also be redone if the program is changed.

- The compiler can be formally verified. Source programs can be formally verified as the results apply to the object code. A single proof (that of the compiler) shows that all object programs produced using the compiler correspond to their source programs. However, formally verifying a compiler for a high-level language is difficult. Also, care must be taken that the compiler correctness theorem proved is sufficient to ensure that the results really do apply to the low-level code.

Much research has been undertaken concerning the verification of both machine code and microcode. Such systems can be divided into those that perform verification on mnemonic assembly language programs and those which use a bit-representation of the code.

The assembly language systems have typically followed Floyd's approach to program verification [25] modified to deal with low-level code. They provide a verification condition generation program which embodies the semantics of the assembly language. Maurer [37, 38] used this approach to verify IBM 370 code and code for the Litton C4000 airborne computer. Lamb also used it in his Intel 8080 Assembly Language Verifier [36]. More recently it has been embodied in the SPADE verification environment. SPADE has been used in the verification of assembly code for the Intel 8080 [11], and also of Z8002 code used in the fuel control unit of the RB211-524G jet engine [43].

Verification of bit-level code has typically been based around an operational semantics of the host machine and the use of formal symbolic simulation techniques. MCS was an early system which took this approach. It was used to verify production code for the NASA Standard Spaceborne Computer-2 [9]. A hybrid approach using verification condition generation techniques to verify bit-level microprograms has also been suggested [17].

Boyer and Yu [4] adopted the approach of writing programs in a high-level language, but verifying low-level code. They verified compiled C and Ada code for the MC68020 microprocessor. Their methodology was to compile the source code using an industrial strength compiler and verify the resulting object code. This was done by first writing a second algorithmic version of the program in the Boyer–Moore logic. The algorithm was effectively a functional version of the program. This was verified to be equivalent to the object code. The algorithm was then shown to be correct. Applying formal methods directly to object code can have advantages in that stronger properties of the program may be provable than when verifying a high-level program. For example, it is easier to reason about timing properties at this level.

Shepherd [47, 48] adopted an approach based on doing a one-off proof between a source program and compiled code. This work concerned the verification of microcode for the IMS T800 floating point Transputer. The intended methodology was to prove correct a high-level Occam implementation of a program then use the Occam transformation system to produce an equivalent microcode version. The microcode version was still an Occam program but matched the micro-machine functions. The transformation system was based on the algebraic semantics of Occam. The transformations were chosen by the user, with the system ensuring they were correctly applied. In practice, for each transformation step an implementation was proposed and then transformed backwards into the higher level version.

There has been interest in formally verifying compilers from the early days of verification technology, the first work being that by McCarthy and Painter [39] in 1966. Since then many different techniques have been used. However, there are as yet no formally verified commercially available compilers for real languages. A good overview of the compiler verification literature is given by Joyce [33]. Notable work, includes that of Polak [45] on a compiler for a Pascal-like language, and the Piton and Gypsy compilers which form part of Computational Logic's verified stack of system components [42, 52].

The majority of compiler correctness work has been concerned only with the correctness of code generators. Exceptions to this include Polak's work [45] and that of Chirica and Martin [10] where aspects of compiler front ends are also considered. There has also been isolated work on the formal verification of the front ends of compilers, notably parsers [14, 12, 26]. The need for a front end can be removed if the abstract syntax used is in a sufficiently readable form. For example, the LISP-like concrete syntax of Piton is also its abstract syntax. Programs are both written and accepted by the verified code generator in this form.

For small projects the cost of verifying a compiler might outweigh the benefits. For example, if only a few programs are safety critical, it might be better to use one of the other methods. However, in the long term, the use of formally verified compilers offers the best solution. This is therefore the approach investigated in this project. We have formally verified a compiler using the HOL theorem proving system. The language and microprocessor considered were a subset of the structured assembly language Vista, and the VIPER microprocessor, respectively. On the whole, previous work in this area has been concerned with the verification of "toy" languages or idealised microprocessors, often designed specifically for the proof of the implementation. VIPER was designed using formal methods, but was intended as a commercial product. Vista was designed to be used for writing real applications software.

1.1 Overview

This report is divided into four chapters. In the remainder of this chapter, we describe the higher-order logic notation we use and overview the compiler correctness problem. In Chapter 2, we describe the verification of the compiler. First we describe the source language of the compiler, giving its syntax and formal semantics. We then describe the formal description of the compiler. Next we discuss the compiler correctness theorem we have proved. Finally we describe how our work is applicable to a family of languages and compilers. In Chapter 3 we discuss how the correctness theorem and verified compiler fit into a wider context of ensuring that object code is correct. We first show how the compiler correctness result can be formally combined with proof systems for application programs. We then show how our verified compiler, despite not being written in a traditional programming language, can be used to produce compiled code. We also discuss how a dependable implementation might be obtained. Finally in Chapter 4 we summarise the project, draw conclusions and suggest further research. In the remainder of this section we overview the main achievements of the project, giving pointers to the sections where they are discussed.

Previous compiler correctness work has considered simple file based models of I/O if it has been considered at all. We investigated a more general model of I/O. The semantics of this model is discussed in Section 2.1.2.

For formal compiler verification to be practical, the results obtained must be repeatable for different systems with a minimum of effort. We therefore considered the verification of a generic compiler from a generic version of Vista to a generic flat assembly code. The correctness results obtained can be quickly targeted to versions of Vista for the VIPER microprocessor or for other similar machines. Previous work has considered only single compilers in isolation. We describe this work in Section 2.4.

We have proven a much simpler compiler correctness theorem than is normal. This is possible because the languages considered are deterministic. We have illustrated that our compiler correctness theorem is sufficient. We have thus drastically cut the verification work required. This is discussed in Sections 2.3 and 3.1.

We have combined our verified compiler with a derived programming logic. Thus correctness properties of source programs can be proved. From these, corresponding properties of the compiled code can be automatically derived. This is the first time that such a formal link has been made. We have also used a novel definition of total correctness which includes I/O behaviour. We have shown how I/O specifications can be naturally given. This work is described in Section 3.1.

We verified a compiling algorithm written in a logic, rather than an implementation in a programming language. We have investigated ways in which such an algorithm can be securely executed. This work illustrates how the verification task can be reduced further, since an implementation need not be verified. This is discussed in Section 3.2.

If an implementation is to be verified, a problem is that it must be compiled somehow, so it appears that a highly assured compiler is needed before a highly assured compiler can be obtained! We have suggested how this problem can be avoided. This is discussed in Section 3.3.

`T`	truth
`F`	falsity
`P x`	`x` has property `P`
$\sim$`t`	not `t`
$t_1 \vee t_2$	t_1 or t_2
$t_1 \wedge t_2$	t_1 and t_2
$t_1 \supset t_2$	t_1 implies t_2
$t_1 = t_2$	t_1 equals t_2
$\forall$`x. t[x]`	for all `x`, `t[x]` holds
$\exists$`x. t[x]`	for some `x`, `t[x]` holds
$t \Rightarrow t_1 \mid t_2$	if `t` is true then t_1 otherwise t_2
`t[a/x]`	substitute `a` for variable `x` in term `t`
`f t`	the application of function `f` to argument `t`
let x = t_1 in t_2	let declaration: $t_2[t_1/x]$
(t_1, t_2)	a pair with first element t_1 and second element t_2
`[]`	the empty list
`CONS h t`	the list with head `h` and tail `t`
$[t_1; \ldots ;t_n]$	the list with elements $t_1 \ldots t_n$
$*$`a`	the type variable `a`
`bool`	the type of booleans
`word32`	the type of 32-bit words
ty_1 # ty_2	a pair type
$ty_1 \rightarrow ty_2$	a function type
$ty = c_1 \mid \ldots \mid c_n$	`ty` is a recursive data type with elements $c_1 \ldots c_n$
`$`	an empty alternative in a type definition
$\vdash$ `th`	`th` is a higher-order logic theorem
`asm` $\vdash$ `th`	`th` is a higher-order logic theorem with assumption `asm`

Table 1.1: Higher-order Logic Notation

1.2 Notation

The work described here has been formalised within the HOL formal verification system. We thus use a higher-order logic notation. It is an extension of first-order logic in which functions may take functions as arguments and return them as results. All terms must have a well-defined type. It may be an atomic type, such as a natural number or a string. Compound types can also be constructed such as pairs, lists and functions. Type definitions similar to the data types of programming languages may also be given. The logic is polymorphic. Type variables may be used for arbitrary types. A basic understanding of typed first-order logic and a functional programming language such as ML should be sufficient to follow the main points of this paper. The notation we use is outlined in Table 1.1.

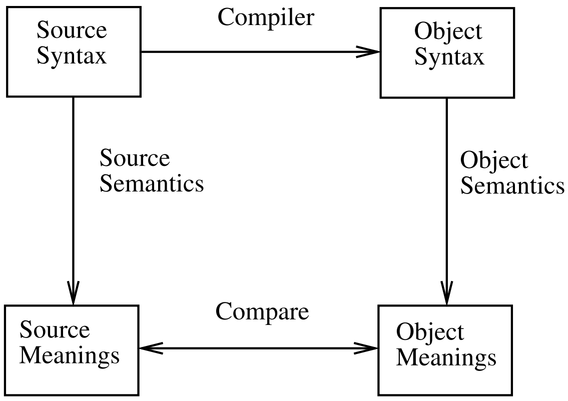

Figure 1.2: An Abstract Specification of Compiler Correctness

1.3 Compiler Specifications

A compiler (the code generation part at least) must produce object code whose meaning corresponds to that of the source program. An *abstract compiler specification* can be given in terms of the source and object language semantics. Informally, a compiler will be correct if the meaning of every source program is related to the meaning of the object code resulting from compiling it. More formally, a compiler must fulfil an abstract specification of the form below.

```
AbstractCompilerSpec compiler =
      ∀p.  Compare (SourceSemantics p)
                   (ObjectSemantics (compiler p))
```

`SourceSemantics` gives the semantics of the source language, `ObjectSemantics` gives the semantics of the target language and `Compare` relates semantics of the two forms. The argument `compiler` is a compiler from the source language to the target language. This form of specification is illustrated in Figure 1.2.

Many different object programs will be suitable as an implementation of a given source program. An *algorithmic compiler specification* is a function which specifies a particular object program for each source program. It specifies a compiling algorithm. To take a simple example, if we assume the target language has conditional branch and unconditional goto instructions, the algorithm might specify that a source language While command is translated as follows.

```
WHILE <test>                          begin:  <test>
  DO                                          BRANCH end
    <body>           ⟶                        <body>
  OD                                          GOTO begin
                                      end:
```

An algorithmic specification is normally given in a particular logic, such as the Boyer-Moore logic or higher-order logic. *Compiler specification correctness*

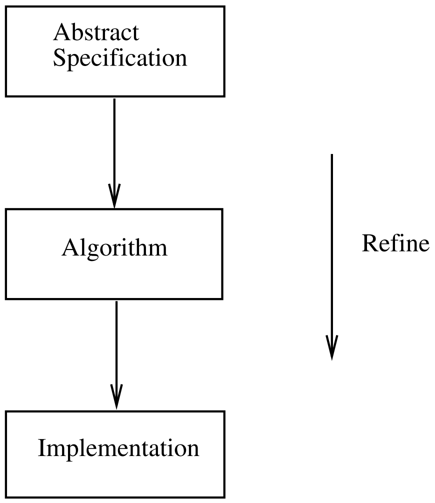

Figure 1.3: The Refinement Hierarchy

concerns whether the algorithm is correct with respect to the abstract specification, that is, whether the semantics of a source program is preserved in the code that the algorithm specifies should be produced. By far the majority of compiler correctness work described in the literature is concerned with this form of correctness.

Given the object code that a compiler must produce for a particular source program, there are many different ways it could be produced. A *compiler implementation* is a concrete program which produces the object code. It specifies not only what the object program should be, but also how it is produced. The implementation is given in a programming language, that is, an executable language. A compiler implementation can be verified against either an algorithm or an abstract specification.

Ultimately, we wish to know that an implementation preserves the semantics of the source language. This suggests we should verify it against the abstract specification. This was the approach adopted by Polak [45]. However, a simpler alternative is to use a verified algorithm as a refinement step towards obtaining a verified implementation (see Figure 1.3). The algorithm is first shown to satisfy the abstract specification. Next the implementation is shown to satisfy the algorithm. It can then be deduced that the implementation satisfies the abstract specification. This split of the problem is similar to that used by Boyer and Yu to verify machine code programs [4]. A similar split has also been used in the verification of protocols [8].

Proving that an algorithm satisfies an abstract specification is simpler than proving that the implementation does. This is because the semantics of the implementation language does not need to be considered in the reasoning. Instead we reason about the logical constructs of the algorithm. When comparing the implementation with the algorithm, the semantics of the programming language in which the compiler is implemented must be

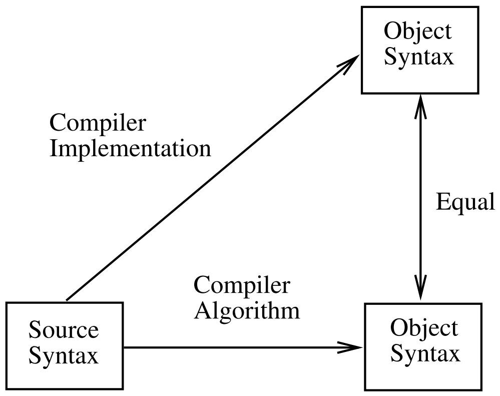

Figure 1.4: Verifying a Compiler Implementation against an Algorithm

considered. However, here the semantics of the source and target languages of the compiler do *not* need to be considered. Only their syntax is important. What is required is that the implementation produces syntactically the same program as indicated by the specification. This approach was followed by Chirica and Martin [10], Simpson [49] and Buth *et al.* [5]. It is illustrated in Figure 1.4.

In this approach, we first prove that the algorithm, `CompilerAlgorithm`, satisfies the abstract compiler specification given by `AbstractCompilerSpec`:

```
⊢ AbstractCompilerSpec CompilerAlgorithm
```

Of the implementation we prove that for all programs the code it produces, `(CompilerImpl p)`, is equal to that specified by the algorithm, `(CompilerAlgorithm p)`.

```
⊢ ∀p.  CompilerImpl p = CompilerAlgorithm p
```

Combining these we obtain the required theorem, which states that the implementation `CompilerImpl` satisfies the abstract compiler specification.

```
⊢ AbstractCompilerSpec CompilerImpl
```

This is illustrated in Figure 1.5.

Splitting the proof into two parts in this way not only simplifies the programming and verification task, but also allows proofs to be reused. If different implementations of the same specification are produced, or the verified one is modified, only the compiler implementation correctness theorem needs to be reproved. The compiler specification theorem can be reused. Of course, the new compiler will have to generate the same code as the old one to fulfil the specification. However, the compiler itself can be more efficient, or contain better error detection. Some flexibility may be left by making the algorithmic specification non-deterministic. However, leaving such choices open to the programmer may make the compiler specification proof harder. It also has

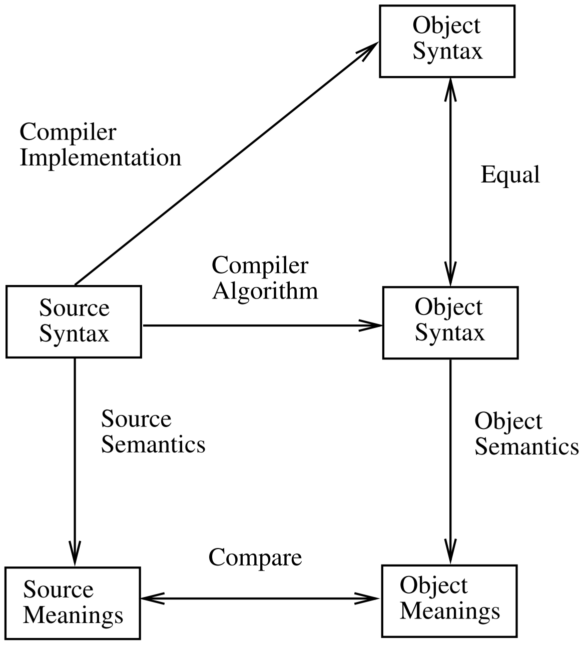

Figure 1.5: Combining Specification Correctness and Implementation Correctness

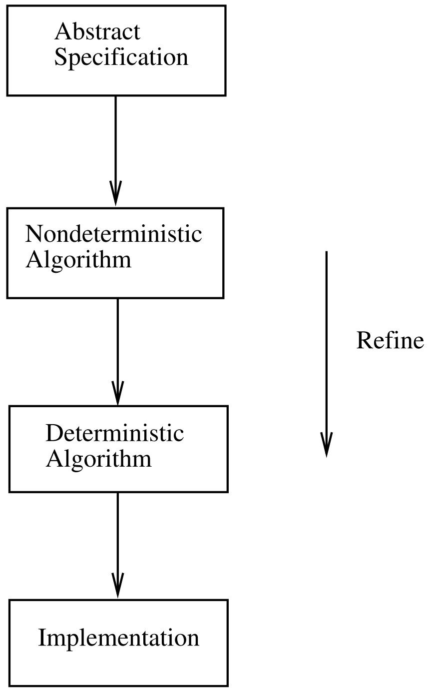

Figure 1.6: The Refinement Hierarchy with Non-deterministic Specifications

disadvantages if we wish to execute some form of the specification as discussed later. It is therefore advisable to use a deterministic specification when verifying a particular implementation. This does not preclude verifying the deterministic specification against a more general non-deterministic one. We would then have three refinement steps as shown in Figure 1.6.

Chapter 2

A Verified Vista Compiler

2.1 The Vista Subset

Our work involved the verification of a compiler for Vista. Vista is a structured assembly language designed at the Defence Research Agency (Malvern). The instructions of Vista correspond directly to the instructions available on the underlying machine. For example, the arithmetic instructions are those provided by the ALU. They operate on the registers of the underlying machine, which are visible to the Vista programmer. In this sense it is an assembly language. However, the program counter of the target machine is not directly visible to the Vista programmer. Instead, high-level constructs such as while loops and procedures are provided. There are also limited data abstraction facilities. For example, variable and channel names are used in place of addresses. Vista was intended for use with the VIPER microprocessor. VIPER is a 32-bit computer, designed for safety-critical applications. Some aspects of it have been formally verified [13].

Vista provides a good case study for investigating the compiler verification problem. It is a "real" language, rather than a "toy" designed with formal verification in mind. It thus provides a realistic case study. However, it is relatively simple and its semantics are largely straightforward. It provides a tractable problem that can be used as the basis for looking at more complex languages.

2.1.1 Syntax

We have considered a substantial subset of the Vista language. The syntax of this subset is given in Table 2.1. The A, X and Y general purpose registers may be accessed. Expressions may be word constants or variables. Conditions may be chosen from the comparison operations and the test of the B register. Commands may be chosen from skip, stop, sequencing, the machine functions, variable assignment, input, output, procedure call and the while loop. Top level variable declarations are provided. Top-level procedure declaration is also provided. A program consists of a series of variable declarations, a series of procedure declarations and then the program body.

An example program, written in the subset of Vista, which performs multiplication by repeated addition is given in Table 2.2.

Rather than use the concrete syntax shown above, we use an abstract syntax. This abstract syntax is described formally as a type within the HOL system. The HOL system contains parsing and pretty-printing generation tools which could be used to generate parsers and pretty-printers to convert between the two forms of syntax. We generated a parser for a smaller subset of Vista, though not for the complete subset. This could easily be done, as could the generation of a pretty-printer. It should be emphasized that this project has not been concerned with the verification of parsers or pretty-printers. We use the pretty-printed version of Vista commands here for clarity.

We used a generic version of Vista in which the word sizes, ALU instructions and comparison instructions are left unspecified. With a minimum of additional work, the same definitions and proofs can be reused for different machines. This is achieved using type variables. The variables `*monop`, `*aluop`, `*arithop` and `*compop` are used in place of the syntactic categories `Monop`, `Aluop`, `Arithop` and `Compop` to represent the monadic ALU operations, logical ALU operations, arithmetic and comparison operations, respectively. They can be instantiated with the operations which correspond specifically to the VIPER microprocessor as in Figure 2.1 or alternatively to those of other machines. The type used to represent a machine word is also left as a variable — `*word32`, rather than the concrete type `word32`. The name is suggestive of the word size of the VIPER microprocessor, though it is only a name: it could be instantiated to a word type of any length. We discuss this in more detail in Section 2.4.

2.1.2 Semantics

To reason formally about the source programs or compiler of a language, we need a formal semantics. Traditionally, the denotational semantics of a command is a partial function from an initial state to the state which results from executing the program from that state. Thus, the semantics of a program `p` would be given by a function `Meaning` from an initial state $\mathtt{q}_1$:

$$\texttt{Meaning p } \mathtt{q}_1 \texttt{ = } \textit{the state resulting if } \texttt{p} \textit{ is executed from state } \mathtt{q}_1$$

The function must be partial because programs which do not terminate do not have final states. All functions in the version of higher-order logic embodied in the HOL theorem prover must be total however. They cannot be used as denotations in this way. In HOL, partial functions are modelled by relations. Thus, the denotation of a program is given by a relation between the initial and final states.

$$\texttt{Meaning p } \mathtt{q}_1 \ \mathtt{q}_2 \texttt{ =} \begin{cases} \texttt{T} & \textit{if } \mathtt{q}_2 \textit{ is the final state when } \texttt{p} \textit{ is executed from state } \mathtt{q}_1 \\ \texttt{F} & \textit{otherwise} \end{cases}$$

If a program does not terminate from a particular initial state, there will be no corresponding final state for which the relation is true. For a non-deterministic program, multiple final states can be related to a single initial state. This is the form of semantics used by Gordon [28] and Joyce [32].

The languages considered by Gordon and Joyce did not include I/O constructs, so I/O was not modelled in the semantics. Many different ways have

```
Compop = < | <= | = | /= | > | >= | LT | GT

Monop = $ | NOT

Aluop = + | - | XOR | AND | NOR | AND NOT

Arithop = ADD | SUB

Variable = string

Register = A | X | Y

Expression = word32 | Variable

Condition = B |
            Register Compop Expression

Command = SKIP |
          STOP |
          Command ; Command |
          Register := Register Aluop Expression |
          Register := Register Arithop Expression |
          Register := Monop Expression |
          Variable := Register |
          Register := INPUT Variable |
          CALL Variable
          OUTPUT Register Variable |
          WHILE Condition DO Command OD

Vlist = Variable | Variable , Vlist

Declarations = DATA Vlist | $

Proc = PROC Variable BEGIN Command END

Procedures = Proc | Proc ; Procedures

Program = PROGRAM
            Declarations
            Procedures
            Command
          FINISH
```

Table 2.1: The syntax of the Vista schema subset

```
PROGRAM
      DATA arg1, arg2, ans, t

      PROC multiply
       BEGIN
       A := 0;
       WHILE X/=0 DO
        A := A + t;
        X := X - 1 OD;
       END

      X := INPUT arg1;
      Y := INPUT arg2;
      t := Y;
      CALL multiply;
      OUTPUT A ans;
      STOP
FINISH
```

Table 2.2: A multiplication program in a subset of Vista

been suggested to model I/O in the literature. On the whole the differences are in the mechanism for recording the events which have occurred or will occur. The details can vary greatly. For example, the data-structures may store the events that have happened (a *trace*), predict events that will happen (an *oracle*), or give all events; past, present and future (a *history*). We consider here just a few examples from the literature. A simple approach used in programming logics is a file based model [31]. Input and output take the form of reading and writing to files. A file is a sequential data-structure and is included as part of the state. Input files are initially full of values and are gradually emptied by the program (an oracle). Output files start empty and are filled (a trace). Only the ordering of events within individual files is recorded. There is no ordering between different files, and in particular the causality between input and output events is not recorded. Both Polak [45] and Stepney *et al* [50] have used this model in their compiler correctness work. In the hardware and microprocessor verification work performed with higher-order logic a model with an explicit notion of time is used. State variables are represented by history functions from time to data values; the unit of time depending on the level of abstraction considered [40]. In the system verification work performed at Computational Logic Inc. [42], an oracle represented by a list of tuples is used for input. On each cycle of the processor, a tuple is removed from the oracle. It represents the values on the asynchronous inputs at that clock cycle.

It should be noted that the top-level formal specification of VIPER side-stepped the issue of I/O. A single memory structure contained both peripheral and normal locations. The undefined constants `FETCH21` and `STORE21` were used to access both parts of the memory. Since these constants were not defined, properties of the memory could not be reasoned about. A property that one would expect to hold of these constants with respect to normal memory is that if a value is stored to some address and a read is later performed from that address, the original value will be obtained.

```
FETCH21 a (STORE21 a source ram) = source
```

However, it is not usually desirable for this to hold if the address is a peripheral address.

For the semantics of Vista, we require a model of I/O which does not include a notion of time, since Vista does not have a real-time semantics. We do need to determine the relative ordering of all events, however, since Vista programs can interact with external devices. This suggests that we should use one data-structure for all events, input or output and whichever channel they are related to. Vista cannot perform events in parallel, so simultaneous events are not an issue. We use an oracle model based on a suggestion by Joyce [35]. The future behaviour of a process is given by a single oracle represented as a list of events. This gives the relative orderings of all the events which occur whilst a process is active. The oracle is not part of the state. This has the advantage that the behaviour of the process is given by a single oracle rather than by the difference between the oracles in the initial and final states. Events contain information such as the data passed, the channel they are associated with and whether they are input or output events with respect to the program. The details may vary, depending on the language being described. For Vista, three kinds of I/O events are possible. Word values may be input to and from memory mapped channels and a stop light may be set. These are represented by the type `Event`.

```
Event = IN address ∗word32 | OUT address ∗word32 | DONE
```

We have extended the relational semantics model to include oracles. The semantics of each construct is defined by a relation between the initial and final states and the oracle. The behaviour of a program which does I/O may depend on the particular values input. The semantic relation of a program will model this non-determinism by relating a given initial state with many oracles and final states. The semantics is thus characterised by the set of all triples for which the relation is true.

A Vista state may be one of three kinds:

```
VistaState = ERROR | HALT Ms Rs | RUN Ms Rs
```

An *Error* state indicates that the program is invalid. A *Halt* state indicates that the program has terminated. A *Run* state is an intermediate state that indicates that the program is executing normally. The latter two kinds of state contain a memory and a register store, giving the values of memory and registers, respectively.

For a given command the Vista semantics will relate two states and an oracle if executing the command from the first state could terminate in the second state whilst exhibiting the I/O behaviour of the oracle. For example, for a Skip command the initial and final states must be identical and the oracle must be empty since Skip does not change the state and performs no I/O. The semantics of the Skip instruction is given by the relation `SemSkip`.

```
SemSkip oracle q₁ q₂ = (q₂ = q₁) ∧ (oracle = [])
```

When executing normally the semantics of the Output command is given by the relation SemOut. It outputs a word value to a given address and leaves the state unchanged. Thus, the semantics specifies that the initial and final state should be identical, and that the oracle should consist of a list containing a single output event.

```
SemOut l w oracle q₁ q₂ = (q₂ = q₁) ∧ (oracle = [OUT l w])
```

The syntax of the Output command refers to channel names and register names rather than addresses and word values. The full semantics of Output must look up the register's value in the state and look up the address corresponding to the channel name in the environment. Either of these actions may cause an error; for example, the channel name may not have been declared. If an error occurs the final state is an Error state and the oracle is empty. If the initial state is not a Run state the state is unchanged and the oracle empty. The full semantics of Output is given by the relation SemOutput defined in terms of SemOut. We do not give the definition here.

The semantic relations for each command are combined into a single relation, SemCommand, defined recursively.

```
(SemCommand SKIP rep env penv = SemSkip) ∧
(SemCommand (OUTPUT src chn) rep env penv = ...
```

SemCommand takes three arguments in addition to the command, states and oracle. The first, rep, is a representation tuple [34]. We discuss it further in Section 2.4. The second is the variable environment. It is a function which maps variable names to their memory locations. The third is a procedure environment. It is a higher-order function which maps procedure names to the semantic relation corresponding to their body.

Sequencing is defined inductively in terms of the component commands. First the semantic relations corresponding to each sub-command are determined for the given representation tuple and environments: (SemCommand c_1 rep env penv) and (SemCommand c_2 rep env penv). These relations are passed to the higher-order relation, SemSeq which combines the semantics of the two sub-commands.

```
SemCommand (c₁; c₂) rep env penv =
       SemSeq (SemCommand c₁ rep env penv) (SemCommand c₂ rep env penv)
```

For the relation SemSeq to hold, there must be a suitable intermediate state and point at which the oracle can be split. That is there must be two oracles which when appended together give the original oracle. They must be such that the first command results in the intermediate state exhibiting the I/O behaviour of the first part of the oracle. The second command must yield the final state from the intermediate state exhibiting the behaviour of the remainder of the oracle.

```
SemSeq csem₁ csem₂ oracle q₁ q₂ =
      ∃q oracle₁ oracle₂.
       (csem₁ oracle₁ q₁ q) ∧
       (csem₂ oracle₂ q q₂) ∧
       (oracle = APPEND oracle₁ oracle₂)
```

The use of oracles is easily extended to more complex constructs such as While loops and procedures.

The semantics of a program is given by SemProgram. First the variable environment is determined using SemDec applied to the empty initial environment. The Procedure environment is then determined using SemPDec again with an empty initial environment. We do not give the semantic definitions for the environments here. The semantics of the command is determined in these environments.

```
SemProgram rep (PROGRAM d p c FINISH) =
       let env = SemDec d EmptyEnv in
       let penv = SemPDec p rep env EmptyPenv
       in
        SemCommand c rep env penv
```

2.1.3 The B Register

The VIPER program counter is not visible to the Vista programmer and so does not form part of the Vista state. The general purpose VIPER registers are always visible to the Vista programmer and thus are part of the state. Both program counter and general purpose registers can be treated cleanly in the proof. The B register on the other hand is visible, but on some occasions its value is not. It can be directly assigned to by the Vista programmer and can be directly tested. However, after it is tested as part of a loop or conditional statement its value becomes "undefined" and is no longer visible to the Vista programmer. Any given Vista compiler will compile a given program so that the B register has a particular value which can be tested. However, a different compiler might give it the opposite value. Thus, a given program could have widely different behaviours when compiled with different compilers. This seems undesirable for safety-critical applications. In the formal semantics we introduced an "undefined" value that the Vista B register holds when not visible. When comparing the Vista state with the VIPER state, an undefined value was defined to match both true and false values in the VIPER B register. This complicates the proofs, since the Vista and VIPER register contents are no longer equal. An alternative would have been to assign an arbitrary and unknown value to the B register when it was undefined. The B registers of the two levels would then have the same type. However, they could not be compared by equality as the arbitrarily chosen value at the Vista level would not necessarily be the same as that chosen by any particular compiler. In the latter approach the B register can be manipulated without a run-time error occurring, as would happen with the code produced by a given compiler. The semantics would just not predict a result of the execution if it depended on the B registers value. With our semantics it is an explicit dynamic Vista error to manipulate the value in the B register when undefined. For safety-critical systems this seems more desirable, as it would be an anomaly in the program if it manipulated the B register when undefined. In retrospect, it would have perhaps been better to design the Vista language so that the B register was always visible, or abstracted away from and so never visible as with the program counter. This would have made the correctness proofs easier and also removed the possibility of a programmer introducing undesirable anomalies into the code.

2.2 The Compiler

The compiler we have verified is an algorithmic specification of a code generator, written in higher-order logic. The compiler is split into a series of levels which perform distinct compiling operations. This simplifies the verification task since it allows the distinct parts to be verified separately. A small number of concepts need be reasoned about at each stage. The syntax and semantics of intermediate languages are formally defined for each level. The Vista compiler we verified is split into two stages. The first stage converts the structured Vista code into a flat unstructured language, Visa. The store of the machine is split into separate infinite stores for each distinct use: a constant store, a code store, a data store, and a link store for procedure link addresses. Of these only the last two are part of the writable state. At this level, the address spaces are allowed to be infinite. Variable names are converted into these addresses. The second level of translation compiles to a VIPER assembly language, Kaa, which has a single finite store. The translator is concerned mainly with converting addresses to this form. A further simple assembly stage is required to obtain VIPER machine code. We have neither implemented nor verified such an assembler, however. Due to its simplicity we forsee no problems in doing so.

At each stage, both the syntax and the semantics of the target language change in a small way. The change in syntax can be seen by considering an example fragment of code. If we assume a word size of 4, the Vista command:

```
WHILE X/=0 DO
      A := A + t;
      X := X - 1 OD
```

might be translated in the first phase to:

```
Address      Visa instruction

(`Code`,1):  VisaCMP /= X (LITERAL #b000)
(`Code`,2):  VisaJNB (`Code`,6)
(`Code`,3):  VisaALU + A A (CONTENTS T (`Data`,3))
(`Code`,4):  VisaALU - X X (LITERAL #b001)
(`Code`,5):  VisaJMP (`Code`,1)
```

If data locations are compiled to high addresses and code to low ones, the above code might then translated in the second phase to:

```
Address  Kaa Instruction
```

```
#b0001:  KaaCMP /= X (LITERAL #b0000)
#b0010:  KaaJNB #b0110
#b0011:  KaaALU + A A (CONTENTS T #b1011)
#b0100:  KaaALU - X X (LITERAL #b0001)
#b0101:  KaaJMP #b0001
```

Splitting the compiler into levels in this way has an advantage in addition to making the proof more tractable. Since each level is verified independently, the definitions and their correctness proofs can be reused in compilers for other source or target languages without re-verification.

The output of the compiler is either a Compile Error or a list of blocks of code and constants, with associated base addresses. The main body of the program and each procedure is compiled to a separate entry in the list. Consider a program with a single procedure:

```
PROGRAM declarations procedure command
```

Suppose that the compiled code of the procedure is $\texttt{code}_p$, that it is compiled to base address $\texttt{code-base}_p$, that the list of compiled constants for the procedure is $\texttt{constant}_p$ and that it is compiled to the base address $\texttt{constant-base}_p$. Suppose also that those of the command are similarly $\texttt{code}_c$, $\texttt{code-base}_c$, $\texttt{constant}_c$, and $\texttt{constant-base}_c$. The result of compiling this program would have the form:

$$\texttt{[CODE(code}_c\texttt{, constant}_c\texttt{), code-base}_c\texttt{, constant-base}_c\texttt{;}$$
$$\texttt{ CODE(code}_p\texttt{, constant}_p\texttt{), code-base}_p\texttt{, constant-base}_p\texttt{]}$$

The subset of Vista considered, is actually compiled to contiguous addresses. The format used is thus more complex than strictly required. This format was chosen in anticipation of adding region declarations to the subset. They allow the separate blocks to be compiled to unrelated addresses. Region declarations were not implemented, however.

We give a formal definition of the semantics of each level. The top (Vista) level was described in Section 2.1.2. For each of the lower level languages an interpreter style description is given of the semantics. That is, a next state function is defined. Given an initial state and possibly other information such as the I/O behaviour, code store etc. as applicable, it returns the new state that results when executing the instruction pointed to by the program counter. The next state function is defined in terms of relations which give the semantics of each individual instruction. A relation between initial and final states for multiple execution steps is then defined in terms of this function. The states of a particular level consist of a tuple of values with entries for each register and store defined at that level. The I/O behaviour of the lower levels is specified by an oracle similar to that used for Vista. The main difference is that the lower level oracles include additional "non-events" which occur when no other event occurs. This gives a crude model of time with respect to the number of instructions executed. More detailed timing models could similarly be used. Since every instruction consumes exactly one event, the single instruction semantics take an event argument rather than a full oracle.

A Visa state is represented as a tuple, `(data, p, (a,x,y), b, stop)`. It consists of a data store, a program counter, a register store holding the A, X and Y registers, the B condition code and a stop flag.

The Visa Stop instruction performs a `DONE` event, setting the stop flag. All other parts of the state are left unchanged. Its semantics is defined by `VisaSemStp`.

```
VisaSemStp tevent (data, p, r, b, stop) s2 =
     (tevent = DONE) ∧ (s2 = (data, p, r, b, T))
```

A decoding relation `VisaStep` chooses the appropriate semantic relation for a given function:

```
(VisaStep VisaSTOP rep amap cst tevent s1 s2 = VisaSemStop tevent s1 s2) ∧
                                        ⋮
```

This relation is used to define a fetch-decode-execute relation giving the semantics of a single execution cycle. Provided the initial state is not erroneous and the processor has not stopped, the semantics of a single cycle is given by the semantics of the instruction at the address in the code store pointed to by the program counter. This relation is in turn used to define the semantics of executing a sequence of instructions given a full oracle.

2.3 Compiler Correctness

We suggested in Section 1.3 that the abstract compiler specification should have the following form:

```
AbstractCompilerSpec compiler =
     ∀p.  Compare (SourceSemantics p)
                  (ObjectSemantics (compiler p))
```

In its simplest form, the relation `Compare` will just be equality; the semantics of the source program should equal the semantics of the compiled program.

```
AbstractCompilerSpec compiler =
     ∀p.  SourceSemantics p = ObjectSemantics (compiler p)
```

This could be proved in two parts: firstly that the source semantics implies the semantics of the compiled code (*Soundness*), and secondly that the semantics of the compiled code implies the source semantics (*Completeness*).

```
⊢ SourceSemantics p ⊃ ObjectSemantics (compiler p)

⊢ ObjectSemantics (compiler p) ⊃ SourceSemantics p
```

Unfortunately, equality is too simplistic a relation for real compiler verification since the states of the two languages are different. Abstraction mechanisms must be introduced. Traditionally the compiler correctness statement is given as the conjunction of more complex soundness and completeness properties. However, if the languages are deterministic then only soundness need be proved.

Completeness follows as a consequence [24, 21]. Ultimately, we require a compiler correctness theorem so that if we prove correctness properties of source programs, we can deduce that similar results hold for the compiled code. We have shown that with only the soundness compiler correctness theorem we can deduce total correctness properties of compiled programs from total correctness properties of source programs. This is discussed in more detail in Section 3.1.

The `Compare` relation we use has the following form, where `source` is the semantic relation of a source program or command and `target` is that for the target code:

```
∀q1 q2 oracle s1.
       source oracle q1 q2 ∧
       CompareStates q1 s1 ⊃
              ∃s2 toracle.
               target toracle s1 s2 ∧
               CompareStates q2 s2 ∧
               CompareOracles oracle toracle
```

This states that if we assume

- the source program takes some initial state q_1 to final state q_2 using oracle `oracle`, (`source oracle` q_1 q_2), and
- q_1 corresponds to initial target state s_1, (`CompareStates` q_1 s_1)

then we can deduce that there is a target state s_2 and target oracle `toracle` such that

- the target semantics from an initial state s_1 will have I/O behaviour as specified by `toracle` and end in state s_2, (`target toracle` s_1 s_2)
- the final target state will correspond to the final source state, (`CompareStates` q_2 s_2) and
- the source oracle and target oracle will correspond, (`CompareOracles oracle toracle`).

The actual relation used is more complex because extra assumptions must be made that the compiled code and constants are loaded into their respective memory stores, that the program counter is initially loaded with the start address of the code and that no static errors are present in the source program, for example. Error states must also be considered. The actual relation we use relates the semantics of a source program with the *syntax* of a target program rather than with its semantics. The semantic relation for the target is built in. Our correctness diagram really has the form shown in Figure 2.1. A compiler correctness statement based on this relation, is sufficient to prove total correctness properties of compiled code from total correctness properties of a source program.

The compiler is split into stages. The first stage compiles Vista programs into Visa code. The second stage compiles Visa code to Kaa code. The proof

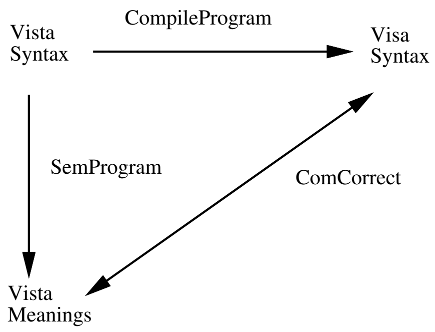

Figure 2.1: **The Compiler Correctness Problem Restated**

of correctness is similarly split. Previously, verified compilers have also been split into levels in this way [33, 42]. First we prove a correctness statement of the above form about the Vista to Visa compiler. We then prove a similar correctness statement about the Visa to Kaa compiler. Finally we combine these two theorems to give a correctness theorem about the Vista to Kaa compiler. This is illustrated in Figure 2.2. It splits the correctness proof into tractable parts. Each is concerned with distinct problems. In the first part we are concerned with flattening a heirarchically structured program and the conversion of symbolic names to addresses. We do not need to consider the finite restrictions on adresses, for example. In the second proof we do consider the finite address restrictions, but now are dealing with flat code and do not need to consider symbol tables.

The main correctness statement for the Vista to Visa compiler is proved by proving appropriate correctness statements for each of the Vista syntactic domains: declarations, commands, etc. These are each proved by structural induction on the syntax concerned. The correctness statement for declarations states that they preserve an appropriate correspondence between the symbol table and the environment. The translation of a program uses an empty initial symbol table and the semantics uses an empty environment. Since these correspond, we can use the declaration correctness theorem to deduce that the correspondence holds between the symbol table used when translating commands and the environment used in their semantics. The correctness statement for commands is proved by structural induction. For each command, we consider three cases, depending on whether the initial Vista state is a run, halt or error state. We then compare the results of executing the Vista command with those for executing the Visa instructions, showing that they correspond.

The main correctness statement for the Visa to Kaa compiler is proven by induction on the number of instructions executed. The proof requires a lemma that the execution of a single Visa instruction is mirrored by the compiled code. This is proven by cases on the different Visa instructions. Assumptions made about the initial state in this proof, must also be proved to hold of the final state. Otherwise, we could deduce nothing about executing sequences of instructions. This proof is mainly concerned with ensuring that all addresses used are within

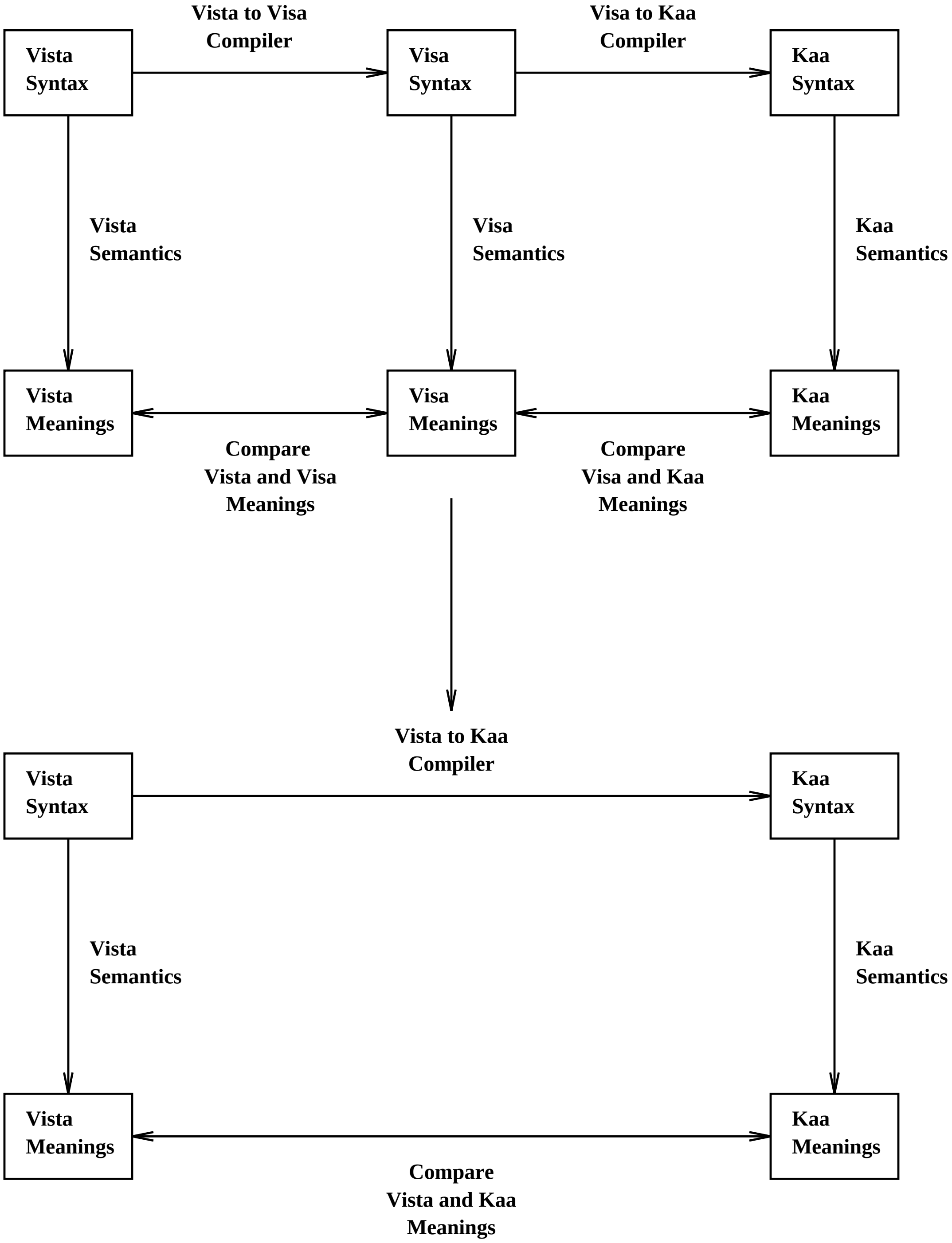

Figure 2.2: Combining Correctness Statements

the bounds of the finite regions. In Vista these bounds are given in region declaration statements. We did not implement such declarations in our subset, though this would be straightforward. Instead, the bounds are provided in a separate data structure to the source program. Various properties of this data structure are assumed, such as that the regions do not overlap and are less than the maximum address of the machine.

The correctness theorems of the two levels of the compiler can be combined on the whole straightforwardly. We essentially have correctness theorems of the following form:

```
⊢ ∀q1 q2 oracle v1.
      (VistaSemantics p) oracle q1 q2 ∧
      CompareVistaVisaStates q1 v1 ⊃
       ∃v2 voracle.
           VisaSemantics (CompileVistaVisa p) voracle v1 v2 ∧
           CompareVistaVisaStates q2 v2 ∧
           CompareVistaVisaOracle oracle voracle

⊢ ∀v1 v2 voracle k1.
      (VisaSemantics c) voracle v1 v2 ∧
      CompareVisaKaaStates v1 k1 ⊃
       ∃k2 koracle.
           KaaSemantics (CompileVisaKaa c) koracle k1 k2 ∧
           CompareVisaKaaStates v2 k2 ∧
           CompareVisaKaaOracle voracle koracle
```

The conclusion of the first theorem which gives the Visa semantics can be matched with the assumption of the second, giving:

```
⊢ ∀q1 q2 oracle v1 k1.
      VistaSemantics p oracle q1 q2 ∧
      CompareVistaVisaStates q1 v1 ∧
      CompareVisaKaaStates v1 k1 ⊃
       ∃k2 v2 koracle voracle.
           KaaSemantics (CompileVisaKaa (CompileVistaVisa p) koracle k1 k2 ∧
           CompareVistaVisaStates q2 v2 ∧
           CompareVisaKaaStates v2 k2 ∧
           CompareVistaVisaOracle oracle voracle ∧
           CompareVisaKaaOracle voracle koracle
```

Ideally we would like the correctness theorem to have the same form as the correctness theorems for the individual levels. This is desirable, because each compiler correctness theorem is intended to serve the same purpose for its particular compiler. We can then define a single `Compare` relation, and hence define a single `AbstractCompilerSpec` relation which defines "abstract compiler correctness". These relations would take as arguments, the various comparison relations, semantic definitions and compiler definition. The above theorem does not have the required form due to the references to the Visa level. We therefore make new definitions which hide the Visa references. We define the Vista to Kaa compiler by

```
CompileVistaVisa p = CompileVisaKaa (CompileVistaVisa p)
```

For the relations comparing states and oracles we use existential quantification. For example, we have separate assumptions which relate Vista and Visa states and which relate Visa and Kaa states:

```
CompareVistaVisaStates q₁ v₁ ∧
CompareVisaKaaStates v₁ k₁
```

We can combine these into a single relation which abstracts away from the Visa state:

```
CompareVistaKaaStates q k =
      ∃v.
       CompareVistaVisaStates q v ∧
       CompareVisaKaaStates v k
```

A Vista state q is related to a Kaa state k if there exists some intermediate Visa state v which compares with both. We thus obtain a correctness theorem in the desired form:

```
⊢ ∀q₁ q₂ oracle k₁.
      VistaSemantics p oracle q₁ q₂ ∧
      CompareVistaKaaStates q₁ k₁ ⊃
       ∃k₂ koracle.
           KaaSemantics (CompileVistaKaa p) koracle k₁ k₂ ∧
           CompareVistaKaaStates q₂ k₂ ∧
           CompareVistaKaaOracle oracle koracle
```

It is possible to go further than this, and remove all references to Visa concepts, even those hidden in definitions. For example, a single pass compiler could be defined which mapped Vista programs directly to Kaa programs. Similarly CompareVistaKaaStates could be defined by directly relating the elements of the Vista state to elements of the Kaa state. This is similar to a hardware verification where the first version is a structural implementation and the second a specification. We did not do this however.

We have simplified things somewhat in the above description. In particular, we must take account of the loading of the code into the Visa and Kaa stores. There are also additional assumptions. For example, in the Visa to Kaa compiler proof assumptions are also made about the form of the Visa program. For instance, it is assumed that all addresses refer to declared regions, that is, regions for which there is an entry in the data structure which holds the bounds information. When the correctness therems of the two levels are combined, such assumptions can be discharged. This involves proving that the code produced by the Vista to Visa compiler does obey these restrictions. We have also assumed that the code exactly fills the declared regions, as this simplified the proof.

2.4 Generic Definitions and Proofs

Our work uses a generic version of Vista in which the word sizes, ALU instructions and comparison instructions are left unspecified. The language schema so defined may be instantiated for different machines. We have essentially defined a general purpose structured assembly language. It is not a universal language, however. The target machine must have a similar architecture to that of VIPER. Our results suggest that much more could be done in this way to make the results applicable to a wider class of architectures.

The language definitions are made generic using type variables to stand for the unspecified features of the language. The variables ∗monop, ∗aluop, ∗arithop and ∗compop are used to represent the monadic ALU operations, binary logical ALU operations, arithmetic operations and comparison operations, respectively. They can be instantiated with the operations which correspond specifically to the VIPER microprocessor or alternatively to those of other machines. The ALU operations are split into separate classes because they take different types of arguments or return different types of results. The type used to represent a machine word is also left as a variable — ∗word32. The name is suggestive of the word size of the VIPER microprocessor, though it is only a name: it could be instantiated to a word type of any length.

The syntax for a particular machine is defined by instantiating the type variables with the syntax of the available operations and words. We have defined syntax for the VIPER1 processor. For example, the type of comparison operators is declared as:

```
VIPER1_COMPOP = VIPER1_LESS |
                VIPER1_NOT_LESS |
                VIPER1_EQUAL |
                VIPER1_NOT_EQUAL |
                VIPER1_LESS_OR_EQUAL |
                VIPER1_NOT_LESS_OR_EQUAL |
                VIPER1_BORROW |
                VIPER1_NOT_BORROW
```

This states that the simple comparisons available for a VIPER1 instantiation of Vista are: less, not less, equal, not equal, less than or equal, not less then or equal, borrow and not borrow. We instantiate the type variable compop to be this type. The above gives the abstract syntax of the comparisons rather than the concrete Vista syntax given in Figure 2.1.

Instantiating the type variables as illustrated above, only defines the syntax of the language. It does not tell us anything about the intended semantics, even if suggestive names are used. The semantic information for a given instantiation of the language is given by a representation tuple. All information specific to a target machine is contained there.

We use the representation tuple in two ways. The first is to provide information about the word types used. For example, one field in the tuple gives a function for converting from natural numbers to the generic data sized words (32-bit words for VIPER1). A second field gives a function which converts words back again. These functions are used in the semantics. In the proof certain properties of these function—and implicitly of the word types involved—are required to hold. For example when a data sized word is converted to a natural number and back again, the original word must result. The function NumToW32RepOf accesses the field holding the former function from the representation tuple, whereas the function W32ToNumRepOf accesses the latter function. Such properties occur in the theorems which depend on them as explicit assumptions about the representation tuple. For example, a theorem dependent upon the above property would contain the assumption:

```
NumToW32RepOf rep (W32ToNumRepOf rep w) = w
```

Using the representation tuple in this way means that the correctness theorems can be used for machines with any word sizes that do not contravene the explicit assumptions made.

The second use of the representation tuple is to provide information about the ALU and comparison operations. This is of use because for a structured assembly language, the semantics of the high level operations such as addition, less than, etc., are identical to that on the target machine. This means that in a compiler correctness proof, the actual details of the semantics are not important. All that is required is that they are the same at both levels. Making these operations generic has two advantages. The first is that the theory can be targeted to machines which provide different ALU operations. The second is that a single proof can be used to prove that all ALU operations of a similar type on one machine are correct. Thus the work required to produce the compiler correctness theorem in the first place is reduced as well as the time taken to re-target it to a new machine.

For example, one field in the tuple contains semantic information for the comparison operations. The semantics of each Vista comparison needs three arguments: the semantics of an expression, `esem`, the semantics of a register, `reg`, and a state, `q`. The expression and register semantics return word values which depend on the state. These are compared by the comparison returning a boolean value. Thus, for example the semantics of a "less-than" comparison might be given by:

```
LessThanSemantics esem reg q = (esem q) ≤ (reg q)
```

The semantics of the other comparison operations is obtained by replacing the $\leq$ operator with that corresponding to the comparison in question. The field in the representation tuple which holds the semantic information for comparisons has type

```
∗compop → ∗word32 → ∗word32 → bool
```

That is, given a comparison operation it returns a comparison function—a function which takes two word values and returns a boolean indicating how they compared. This field is accessed using `CompRepOf`. The semantics of all comparisons can therefore be given by the definition below, which additionally takes representation tuple and comparison operation arguments. It looks up the semantics of the particular comparison operation in the representation tuple (`CompRepOf rep`) applies it to the comparison operation giving the specific comparison function for that operation (`CompRepOf rep compop`), and applies that to the words determined by the expression and register semantics. Note that we use a prefix notation here rather than the infix notation used above.

```
VistaSemComp rep compop esem reg q = (CompRepOf rep compop) (esem q) (reg q)
```

At the lower, Kaa, level the comparisons perform the same operations on word values. Thus, the result is obtained once more by looking up the comparison function in the representation tuple. However, the result is used in a different way here to at the Vista level. Rather than being the final result, it is used to update the B register in the state tuple.

```
KaaSemComp rep compop w1 w2 (ram, p, r, b, stop) s2 =
      s2 = (ram, INC p, r, (CompRepOf rep compop) w1 w2, F)
```

Targeting the theory to a particular machine involves supplying a value for the tuple, and proving the explicit assumptions made about the tuple. If the type variable ∗compop is instantiated to the type VIPER1_COMPOP described earlier and the word type is instantiated to 32 bit words, the type of the comparison field of the tuple becomes:

```
VIPER1_COMPOP → word32 → word32 → bool
```

Thus for VIPER1 we must provide a representation tuple which gives a semantic function mapping from each of the elements in type VIPER1_COMPOP to an appropriate comparison function. For example, VIPER1_EQUAL is mapped to the function EQUAL_SEM which compares two 32-bit words for equality:

```
EQUAL_SEM (w1:word32) w2 = (w1 = w2)
```

Our compiler correctness theorem holds for all representation tuples which satisfy the assumptions made in the proof about the tuple. These assumptions are explicit in the theorem. Consider the following simplified version of the correctness theorem, which has a single assumption about the conversion of words.

```
NumToW32RepOf rep (W32ToNumRepOf rep w) = w
      ⊢ Compare rep (VistaSemantics rep p)
                    (KaaSemantics rep (CompileProgram rep p))
```

It states that provided the function of the representation tuple for converting from a natural number to a 32-bit word is an inverse to the function which converts a word to a number, then the compiler will be correct. Consider now the representation tuple, VIPER1rep for the VIPER1 microprocessor. We obtain a correctness theorem for a compiler for VIPER1, by instantiating the variable rep with VIPER1rep in the above theorem. We define the semantics of VIPER1 versions of Vista and Kaa with a corresponding compiler by:

```
VIPER1VistaSemantics = VistaSemantics VIPER1rep

VIPER1KaaSemantics = KaaSemantics VIPER1rep

VIPER1VistaCompiler = CompileProgram VIPER1rep
```

That is, the VIPER1 version of the compiler and semantics are obtained by applying the generic versions to the concrete representation tuple. Programs containing the ALU, comparison, etc. operation syntax and word sizes specified by the types used in the representation tuple are compiled and their semantics determined using these definitions.

Once we have made these definitions, we can trivially prove from our generic compiler correctness theorem a theorem specifically for the VIPER1 microprocessor.

```
NumToW32RepOf VIPER1rep (W32ToNumRepOf VIPER1rep w) = w
      ⊢ Compare VIPER1rep (VIPER1VistaSemantics p)
                          (VIPER1KaaSemantics (VIPER1VistaCompiler p))
```

This theorem has an explicit assumption stating that the conversion functions given in the VIPER1 representation tuple convert a word back to itself.

```
NumToW32RepOf VIPER1rep (W32ToNumRepOf VIPER1rep w) = w
```

If this does not hold, then the theorem tells us nothing about the correctness of the compiler. We must prove it to be true, before we accept the compiler correctness theorem. This is very easily done. Thus with very little proof effort, we obtain a correctness theorem for a VIPER1 version of the compiler. We could similarly obtain theorems for other microprocessors for which a suitable representation tuple was provided. The limited amount of proof required in this approach contrasts with that needed to verify a compiler for each new target machine from scratch if generic methods had not been used.

Chapter 3

Of What Use is a Verified Compiler Specification?

In the previous chapter we described our verified compiler. In this chapter, we consider how this work fits into a wider context of producing correct object code. In particular, we consider how correctness theorems about compiled code can be formally obtained from correctness theorems about source programs. We then consider ways in which correct compiled code can actually be obtained.

3.1 Combining a Compiler Correctness Theorem with a Programming Logic

The reason for verifying a compiler is to increase our confidence that if we compile a correct source program, the object code will also be correct. How can we be sure that the particular theorem about the compiler that we have proved tells us this? The answer is to use the compiler correctness theorem to derive an inference rule which given a correctness theorem about a source program returns a correctness theorem about the compiled code. We can then use this inference rule to obtain theorems about compiled application programs. HOL ensures that faultily programmed derived inference rules cannot produce theorems unless they are theorems. Thus we can be sure that the correctness theorem we obtain really is a theorem.

We could reason about the correctness of application programs directly using the relational semantics. However, it is more convenient to use a programming logic [30]. In this approach, axioms and inference rules are given which allow correctness properties of programs to be proved. A programming logic gives a new semantics for the language. To ensure that it is consistent with the relational semantics we derive the former from the latter [28]. This involves giving a formal definition of the correctness property embodied in the programming logic in terms of the relational semantics. Theorems are then proved about this definition. From these theorems we derive inference rules which correspond to the axioms and rules of the programming logic. The programming logic is thus a toolkit of derived rules for deducing correctness properties about the semantics of programs. These rules abstract away from the relational definitions. As the toolkit is just a series of HOL derived inference rules both the security and full power of HOL are retained in programming logic proofs.

We have demonstrated this approach by combining our verified compiler with a derived programming logic embedded in the HOL system for the Vista subset. We can prove properties of Vista programs using the programming logic. We then use our derived inference rule to obtain a theorem stating that a corresponding property holds for the compiled code. No such formal link has previously been made. Gordon's derived programming logic [28] was for a simple imperative language similar but not identical to that used by Joyce in his compiler correctness work [33]. The Gypsy Verification Environment implements a programming logic for Gypsy which complements Young's verified compiler. However, there is no formal link between the semantics used for the compiler correctness proof and that embodied in the verification environment, so the two could potentially be inconsistent [52].

The correctness of a program is often split between the properties of partial correctness and termination. A program is partially correct with respect to a precondition and postcondition if, whenever the program is executed from a state satisfying the precondition and from which it terminates, the final state satisfies the postcondition. Termination is the property that the program does cease execution from a state satisfying the initial condition. Total correctness is then the combination of partial correctness and termination. A program is totally correct with respect to a precondition and postcondition, if whenever the program is executed from a state satisfying the precondition, it terminates and the final state satisfies the postcondition.

Rather than separating the issues of partial correctness and termination, we give a single programming logic of total correctness which encompasses both. We show how partial correctness properties can be derived from statements of the logic. With traditional definitions this is trivial. However, our definitions of total correctness and partial correctness are non-standard. They are extended to deal with I/O behaviour. The full details of the programming logic and its derivation are given elsewhere [20, 22, 16].

When specifying the I/O behaviour of a program, we believe it is natural to give a single assertion which describes both the inputs and outputs and the way they are intended to interleave. This is consistent with our use of oracles in the relational semantics. It allows us to prove that a program obeys a single I/O specification. This specification can in turn be shown to satisfy more abstract properties. For example we may wish to prove that all the behaviours it represents are such that an output always immediately follows an input. The values input are provided by the environment rather than being under the control of the program. Hence, they must be supplied in the precondition where they can be represented by auxiliary variables. Furthermore, any restrictions on their values must be part of the precondition. We therefore include all the I/O specification in the precondition for our logic. The I/O behaviour of a program is specified by "predicting" the I/O events which will occur during the program's execution. A more conventional definition of partial correctness, in which only the input restrictions appear in the precondition, can be derived from the total correctness definition.

The assertion language provides some basic assertions about I/O properties, together with a way of combining them into more complex assertions. The assertion {`IN chn w`} specifies that a generic word value `w` is to be input on channel `chn`. {`OUT chn w`} specifies that the value `w` is to be output on channel `chn`. {`DONE`} indicates that the stop light will be set (and the program terminate). Conditions may be sequenced using a temporal conjunction operator (`;`). Events predicted in the first conjunct must occur prior to those in the second. For example, {`IN arg v; DONE`} is an assertion which indicates that a single Input event will occur after which the stop light will be set. Assertions specifying repetitive behaviour or behaviour that depends on the values input can similarly be defined. The assertion language can be extended by the user as required. This is an advantage of embedding the programming logic within HOL. The full definitional power of HOL is available. New definitions can be made as required, either to abbreviate assertions, give more expressive power or even to alter the style of specification.

We prove, using the compiler correctness theorem, a theorem of the following form and derive a corresponding inference rule.

```
⊢ ∀P p Q rep.
      ProgramCompiles rep p ∧
      SourceTOTAL rep P (SemProgram rep p) Q ⊃
            TargetTOTAL rep P (CompileProgram rep p) Q
```

This states that if a source program compiles and if it has been proved to be totally correct with respect to an I/O specification `P` and postcondition `Q` then we can deduce that the result of compiling it is also totally correct with respect to that specification. Note, that different definitions of total correctness are used for the source and target languages. This is because the target definition must take into account the different type of states at the lower level of abstraction and also make assumptions such as that the code is loaded into the store. The inference rule must be given a theorem stating that a program of interest compiles and also a theorem stating its total correctness. It returns a theorem stating that the compiled version is similarly totally correct.

Suppose we have proved the correctness of the multiplication program given earlier using the derived programming logic, and have obtained a theorem of the form:

```
⊢ SourceTOTAL rep
      {IN arg1 v1; IN arg2 v2; OUT ans (v1 × v2); DONE}
      (SemProgram rep <multiplication program>)
      {A = v1 × v2}
```

In brief, this specifies that the I/O behaviour of the program is to input two values (`IN arg1 v1; IN arg2 v2`), output the result of multiplying them, (`OUT ans (v1 × v2)`), and then set the stop light and terminate, (`DONE`). It also states that in the final state the accumulator will hold the result output, (`A = v1 × v2`). No assumptions about the initial state are made.

We can deduce a corresponding theorem about the code produced by the compiler, using the derived inference rule.

```
⊢ TargetTOTAL rep
      {IN arg1 v1; IN arg2 v2; OUT ans (v1 × v2); DONE}
      (CompileProgram rep <multiplication program>)
      {A = v1 × v2}
```

We only implemented such a derived inference rule for the Vista to Visa compiler. It would be simple to extend this to the Vista to Kaa compiler.

We can also obtain a correctness result which includes the text of the compiled code itself. This can be done automatically using constant folding and rewriting techniques on the definitions of the compiler. This process is described in more detail in Section 3.2.

```
⊢ TargetTOTAL rep
      {IN arg1 v1; IN arg2 v2; OUT ans (v1 × v2); DONE}
      <compiled multiplication program>
      {A = v1 × v2}
```

The programming logic is a logic for the Vista schema. It can be targeted to a particular machine by providing a suitable representation tuple. This means that generic source language proofs can be performed and instantiated for any machine which provides operations with the properties used in the program. For example, the generic multiplication program above could be verified. The resulting theorem could then be re-targeted to machines of any word size and any ALU that had addition, subtraction and non-equality operations. As with the compiler correctness theorem, any assumptions made about the operators used would be explicit in the proof. An obligation when targeting the generic program to a particular machine would be to discharge these assumptions. For example, for the multiplication program we might have an assumption that the zero constant used was an identity for the addition operator used:

```
∀a.  a + 0 = a
```

This property would need to be proved for the particular addition operator provided by the target machine.

3.2 Executing a Compiler Specification

We have verified a compiler algorithm rather than an implementation in a programming language. We ultimately require a compiler which can be executed. However, higher-order logic is not directly executable. An implementation could be developed in any suitable programming language, with the algorithm being the specification that the programmer works from. Having a formal specification of a programming problem is good in its own right. This is an area where formal methods are already proving themselves to be of use in industry. For example, Z and VDM are widely used. Producing correct formal specifications is difficult. Specification errors account for most of the bugs in code. Thus, possessing a verified specification is of great use when implementing a compiler even if the implementation is not then formally verified. It gives an unambiguous and correct description of the code that must be produced by the compiler. A verified compiler specification can help the programmer avoid introducing bugs. If a proof theory is available for the implementation language, then a standard program correctness proof can also be performed, using the verified algorithm as the specification. This is a very secure way of obtaining an implementation. However, implementation verification can be time-consuming. It may be that time constraints do not permit an implementation correctness proof to be performed. We therefore consider other approaches. An algorithm could be executed in several ways, where there is a trade off between the effort required and the security obtained. We have investigated two ways that relatively secure implementations of compilers can be obtained from a verified algorithm: execution by translation to ML and execution by proof. In this section we describe these techniques but also overview other possible methods of execution.

If the algorithm is given in an executable language, the distinction between an algorithm and an implementation is blurred. When this is so, the algorithm itself can be used as a compiler implementation. The work done at Computational Logic Inc. where the Boyer-Moore logic was used is a case in point [42, 52]. The Boyer-Moore logic is a first-order, quantifier-free logic resembling pure Lisp and hence is executable. The Boyer-Moore theorem prover contains an interpreter for the logic which can be used to execute specifications. Thus, verified compiler specifications can perform compilation. Due to the extra expressiveness of the HOL system, higher-order logic is not executable in this way. Instead an interpreter for an executable subset could be written. Algorithms specified using the subset could then be executed. A potential source of insecurity in this approach is that the interpreter or compiler for the logic may not be correct. It may give a different semantics to the logic. Thus a verified implementation of the logic is ideally required. No such interpreter currently exists for HOL. We did not pursue this approach further.

Alternatively, the specification could be rapidly prototyped in a non-executable logic by translating it into a similar but executable language. In the field of hardware behavioural specifications, Albert Camilleri showed that specifications in higher-order logic can be automatically translated into the functional programming language HOL ML and so simulated [6]. Hall and Windley [29] have adapted this approach to allow microprocessor specifications to be executed. Researchers at the University of British Columbia use similar techniques to automatically translate general deterministic higher-order logic specifications into HOL ML code [46]. In this approach, the potential sources of insecurity are in the correctness of the translator and the correctness of the implementation of the simulation language. In conjunction with the UBC researchers we have prototyped an ML version of the verified Vista compiler using their tool [23]. When this implementation was applied to a Vista program which had previously been compiled by hand, errors were found in the hand-produced target code. This appears to be a good way of quickly prototyping a compiler with fairly high confidence of its correctness. It could also be used to test the compiler definitions prior to verification. This would remove more obvious mistakes prior to verification.

If the implementation language has a close syntactic correspondence with the logic, errors in the translation process can be reduced. Also, there is a greater chance that any errors that do occur will be detected by a visual comparison of the specification and its translated form. For example, a specification written in a subset of higher-order logic can be identified with an implementation in Standard ML. This was the approach taken by Aagaard and Leeser [1]. They verified a higher-order logic specification of a logic synthesis tool using Nuprl. It was also implemented in Standard ML using corresponding definitions. In some cases the definitions required by Nuprl were in a different form to that required by Standard ML. Theorems corresponding to the Standard ML style definitions were therefore proved from the Nuprl definitions. The insecurity of this approach is that the semantics of the logic and language may not be the same, even though their syntax is. This can impart a false sense of security about the resulting implementations. For example, as noted by Aagaard, Standard ML and higher-order logic do not match exactly since the former is an eager language whilst the latter is lazy.

We illustrate the translation approach with a simple example: the definition of a list APPEND function. Definitions in HOL higher-order logic, Standard ML and HOL ML, respectively, are given below.

```
|- (!l. APPEND [] l = l) /\
   (!l1 l2 h. APPEND (CONS h l1) l2 = CONS h(APPEND l1 l2))

fun APPEND [] l = l |
    APPEND (h :: l1) l2 = h :: (APPEND l1 l2)

letrec APPEND =
    fun [] . (\l. l) |
        (h . l1) . (\l2. (h. (APPEND l1 l2)))
```

The main difference between the higher-order logic and Standard ML definitions is in the syntax of the CONS constructor which is a prefix operator in HOL and infix in Standard ML. The HOL ML syntax differs even more. Because the pattern matching mechanism is not so general, lambda expressions (\) are used for the second argument. This makes it much harder to visually confirm that it is the same function as the higher-order logic one.

Alternatively, an executable specification language can be semantically embedded in a non-executable logic. That is, the semantics of an executable language can be defined within the logic. Language terms then have the same semantics as the logic equivalent. The compiler can be specified in that language, and so be executable. Since the underlying logic is still the original logic, the theorem proving tools associated with it can still be used. Sufficient proof infrastructure, such as derived inference rules would have to be developed to allow proofs in the embedded language to be naturally performed. Such semantic embedding has been done for several specification languages in HOL, such as linear temporal logic [32], and VDM style specifications [28]. Aagaard's work, described above, essentially involved embedding Standard ML in Nuprl. Since Standard ML is so similar to higher-order logic little work was needed to define the semantics. Such semantic embedding has also been used in the field of hardware verification. Subsets of the languages ELLA, VHDL and SILAGE, for which simulators are available, have been semantically embedded in higher-order logic [2]. Also when performing a compiler correctness proof, the semantics of the source and target languages must be defined: that is they are semantically embedded in the logic. Semantic embedding removes the insecurity of translating between the logic and specification language, though the possibility of an incorrect implementation of the simulation language remains. Of course, if the language which is embedded has a complex semantics such as a programming language, the advantages of having a separate algorithmic specification are lost. We have not pursued this approach further.

A more secure approach is to use formal proof to perform the compilation [21]. This is done by taking the definitions of the compiler, specialising the appropriate variable with the program to be compiled and performing rewriting until target code is obtained. It can be done automatically using a mechanized proof assistant such as HOL. This means that the actual definitions that have been verified are executed. As a side effect a theorem is obtained stating that applying the algorithm to the source program yields the compiled code.

```
⊢ FunctionalCompilerSpec SourceProgram = CompiledCode
```

The approach can be illustrated again using the definition of `APPEND` given earlier. In the following we use the standard bracketed notation for lists. For example, `[1; 2]` is an abbreviation for `CONS 1 (CONS 2 [])`.

Suppose we wish to execute `APPEND` applied to the lists `[1; 2]` and `[3; 4]`. Initially, the variables `l1`, `l2` and `h` in the second clause of the definition of `APPEND` are specialised with `[2]`, `[3;4]` and `1`, respectively. This gives the theorem:

```
⊢ APPEND [1; 2] [3; 4] = CONS 1 (APPEND [2] [3; 4])
```

In a similar way we can also obtain the theorem:

```
⊢ APPEND [2] [3; 4] = CONS 2 (APPEND [] [3; 4])
```

We can use this to rewrite the first theorem giving:

```
⊢ APPEND [1; 2] [3; 4] = CONS 1 (CONS 2 (APPEND [] [3; 4]))
```

Next, we specialise the first clause of the definition of `APPEND` with the list `[3;4]` to give the theorem

```
⊢ APPEND [] [3; 4] = [3;4]
```

Rewriting the previous theorem with this we obtain the desired theorem:

```
⊢ APPEND [1; 2] [3; 4] = [1; 2; 3; 4]
```

This tells us that the result of executing `APPEND` with these values is the list `[1; 2; 3; 4]`.

We can use the same tool to perform symbolic execution. For example, we can obtain a theorem containing variables in place of the numbers. The theorem holds for all values of the variables:

```
⊢ APPEND [m; n] [p; q] = [m; n; p; q]
```

Similar tools can be built for any HOL definition and in particular for those of a compiler algorithm. The tools used to perform execution of definitions in this way are conversions [44]. Given a term in the logic they return a theorem expressing an equality between that term and another. Various tools are available in HOL for creating rewriting conversions for a particular definition and for combining conversions. Thus tools for executing compiler definitions are straightforward to build. Further, they can be built compositionally. If conversions are written to execute definitions that are used in a later definition, an execution conversion for the later definition can be obtained by combining the original conversions. For example, a tool to compile programs by proof can be built from previously written tools to compile declarations and commands by proof. Such tools are of more use than just executing the verified compiler algorithm. They could be used to test the definitions prior to verification, though translation to ML would probably be a more successful approach to do large amounts of testing of this form. They could also be used to generate theorems which will be of use when verifying the algorithm.

Juanito Camilleri has used this technique very successfully to simulate the definitions of a compiler for an Occam subset [7]. Valuable feedback was obtained to help ensure the definitions were correct before verification was attempted. Goossens [27] has also used execution by proof, though to simulate hardware designs and in combination with semantic embedding. The hardware description language picoELLA was semantically embedded in higher-order logic. The LAMBDA theorem prover was then used to execute designs written in picoELLA.

We have written an execution by proof tool for the top level of our verified compiler [15]. Given a Vista source program written in the abstract syntax, it returns a theorem giving the compiled Visa code. The tool could easily be extended to compile to binary code. The theorem obtained when the tool is applied to the multiplication program instantiated for the VIPER1 microprocessor given earlier is given in Figure 3.1. A generic version of this theorem could also be obtained automatically. It could be retargeted to other machines or to programs which used other values of the constants,

We have identified a series of simple techniques that can be used when writing execution conversions to speed up the execution. They include:

- writing the conversions in a modular way so that rewrites are only applied when they have a chance of being useful;
- pre-proving theorems which embody the expansion of multiple definitions;
- avoidance of duplicating expressions before they are rewritten;
- assuming that error results will not occur (we can ensure there are no compile-time errors using an insecure compiler first);
- avoidance of rewriting terms fully until a late point to prevent large terms from being generated;

```
⊢ CompileProgram VIPER1rep
      (PROGRAM
       DATA arg1, arg2, ans, t

       PROC multiply
        BEGIN
        A := #00000000000000000000000000000000;
        WHILE X VIPER1_NOT_EQUAL #00000000000000000000000000000000 DO
         A := A VIPER1_ADD t;
         X := X VIPER1_SUB #00000000000000000000000000000001 OD;
        END

       X := INPUT arg1;
       Y := INPUT arg2;
       t := VIPER1_ID Y;
       CALL multiply;
       OUTPUT A ans;
       STOP
      FINISH) =

      [CODE
        ([VisaINPUT X(VisaAT(‘Default‘,0));
          VisaINPUT Y(VisaAT(‘Default‘,1));
          VisaSTORE Y(VisaAT(‘Default‘,3));
          VisaCALL(‘Default‘,0);
          VisaOUTPUT A(VisaAT(‘Default‘,2));
          VisaSTOP],[]),
       (‘Default‘,9),
       ‘Default‘,0;

       CODE
        ([VisaSLINK(‘default‘,1);
          VisaMONADIC VIPER1_ID A(VisaLITERAL #00000000000000000000);
          VisaCOMPARE VIPER1_NOT_EQUAL X(VisaLITERAL #00000000000000000000);
          VisaJMPNOTB(‘Default‘,7);
          VisaALU VIPER1_ADD A A(VisaCONTENTS T(‘Default‘,3));
          VisaALU VIPER1_SUB X X(VisaLITERAL #00000000000000000001);
          VisaJMP(‘Default‘,2);
          VisaRTN(‘default‘,1)],[]),
       (‘Default‘,0),
       ‘Default‘,0]
```

Figure 3.1: The Compilation Theorem for the Multiplication program

- memoisation to ensure that identical terms are only rewritten once

It is safe to make the assumption that error results do not occur; it will not cause incorrect code to be obtained. If the result should be an error, the appropriate rewrite rules will not be a present and so the term will just not be fully reduced. The above are all general purpose techniques: their use is not restricted to the execution of compilers.

Execution of a verified algorithm by proof is very secure. The actual definitions rather than some translated form are executed. Also the problem, encountered when using a programming language to execute the definitions, of a mismatch between the semantics of the logic and that of a programming language is avoided. The only point of insecurity in this methodology is in the theorem prover itself. A faulty theorem prover could incorrectly rewrite the compiler definitions, producing incorrect target code. In a system such as HOL, the execution strategy is the application of primitive inference rules and axioms of the logic. Type checking ensures that the system only accepts valid theorems as theorems. The tool programmer cannot make programming mistakes which cause incorrect compilation to occur. If a theorem of the above form is obtained it must have been produced using primitive inference rules. It must really be a theorem about the algorithmic specification. The compiled code can only be wrong if there is a mistake in the few basic primitive inference rules of the system or in the type-checker. The kernel of such a system which must be secure is thus small. All proofs using the system are dependent on the correctness of this kernel. Therefore, if the theorem prover is used widely for formal proof, as is the HOL system, these mechanisms are widely tested. Furthermore, the theorem prover is already being used to obtain the compiler correctness theorem. The compiled code can be trusted to the same degree as the correctness proof itself. If the proof is to be trusted effort must already be expended in ensuring that the theorem prover is sound. No additional validation overhead is incurred by compiling programs in this way. Further confidence can be obtained by performing the compilation using different implementations of the theorem prover and comparing the results.

In fact, in the process of proving that the algorithm is correct, we also increase our confidence that this execution strategy is correct. In doing a compiler correctness proof the same reasoning is used as when "executing" the compiler. To prove that the compilation of a particular command is correct, the compiler definition is rewritten until target instructions are obtained. The semantics of the resulting code is then compared with that of the original code. We can use the same tools as used to execute the code to do this. Thus if there is an error in the theorem prover that means the wrong code is produced it is likely that it would also have caused the compiler proof to fail. If not then it suggests that the "wrong" code has the right semantics. The code is, if not the desired code, still correct. Alternatively, it could mean there is a further error in the proof which nullifies the original error. Since this means the correctness proof is invalid, code from a compiler satisfying the specification can not be relied on whatever means were used to produce it. Thus lemmas used in the correctness proof are also indirectly correctness results stating that the execution strategy produces correct code. Whilst this increases our confidence in the code, it is not a firm guarantee, since the lemmas will only be about fragments of code. They are also symbolic.

For example, to prove that the translation of a simple form of assignment command is correct, we first obtain a theorem of the form:

```
⊢ Compile env (a := e) = STORE (TransVar env a) (TransExp env e)
```

This states that executing an assignment in some environment `env` yields a `STORE` instruction of the value obtained by translating the expression `e` to the location obtained by translating the variable `a`. If the theorem prover is faulty and, for example, instead generates the "theorem" below, the remainder of the correctness proof should fail. This "theorem" suggests that the compiled code is a jump. However, the semantics of the jump do not correspond to that of the assignment.

```
⊢ Compile env (a := e) = JMP 0
```

The correct theorem above is symbolic. It does not tell us about the execution of expressions, for example. However, similar theorems are also used to prove the correctness of expressions, so their execution is also checked. Even so, a problem could arise if the execution strategy fails due to an interaction between expression and command translation.

Whilst being a relatively secure method of compilation, execution by proof is very slow. Using it to compile large programs during the development cycle is infeasible. However, at this stage a secure compiler is not essential. An insecure but fast compiler can be used for development, with the specification being executed just once to produce the final production code. Use of an insecure compiler during development has advantages other than speed. It could include much more complex error detection, reporting and recovery facilities, for example. The secure compiler is restricted to the safety critical core, thus simplifying its validation.

An alternate way to view this is that compilation-by-proof is just used to check the code produced by the insecure compiler, rather than to do the compilation itself. Only once the application program is considered correct is it worth checking that the compiler has produced correct code. Indeed, it is simple to write a tool using the compilation-by-proof conversion to do this checking automatically. Given a source program and object code produced by an untrusted compiler, the tool would either return a theorem stating that the latter was correct compiled code for the former, or it would indicate that the untrusted code was incorrect. This idea of providing a secure tool to check compilation has been suggested before [3]. However, it was previously suggested that a translated version of the compiler perform the checking.

A problem with using an insecure compiler for development is that the verified compiler is only used for producing the final production code. The extra confidence in the verified compiler which would otherwise be achieved by large use, is lost. However, if on the production run the compiled code from the verified compiler is found to be syntactically identical to that produced by the

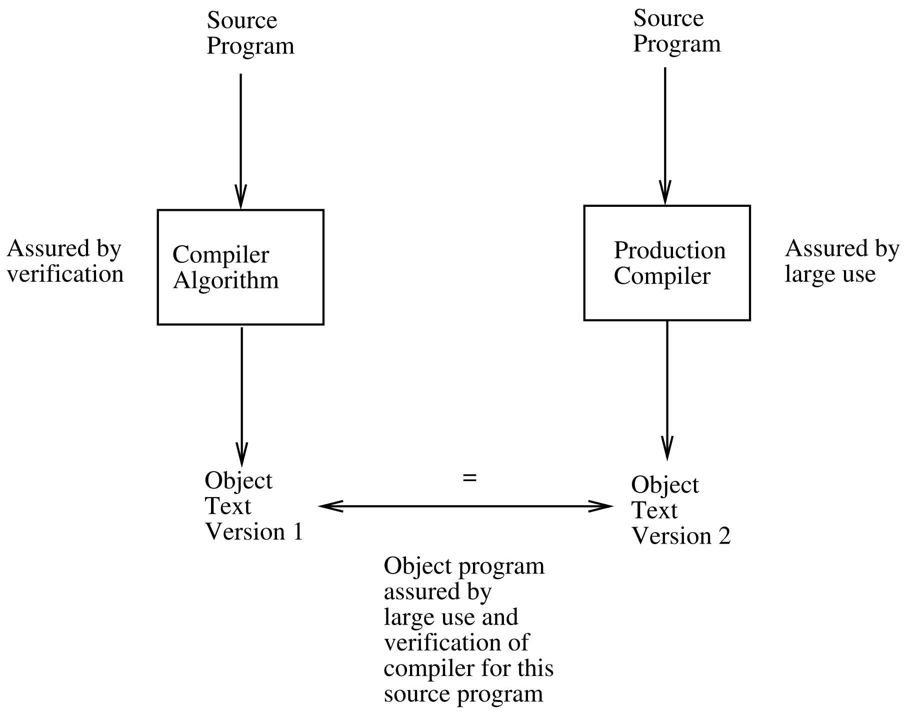

Figure 3.2: Comparing the Object Code from Different Sources

insecure compiler, the extra assurance is retained (see Figure 3.2). Indeed, if the production compiler is largely trusted due to its wide-use, possibly outside safety critical situations, this combined use actually increases our confidence in the correctness of the final code. A question arises over what to do if the results are different. Both compiled programs could still be correct—for example, they could have just identified a variable with different locations. This would almost certainly arise if the production compiler was taken off the shelf and not developed with the specification in mind. If, however, the secure compiler was used as the specification of the insecure one when the latter was developed, the difference is a failure of the compiler to meet its specification. It is indicative of a bug that needs to be fixed. Note that using the secure compiler as the formal specification during the development of the insecure one does not imply that the insecure one has to be formally verified. Validation can also be targeted at any differences found between the results of the compilers to determine whether they are critical.

A further problem is that a user could gain confidence in the correctness of an incorrect program from testing with an incorrect compiler implementation. The production code produced using a secure compiler would then not have the expected properties. Even if the application program had been verified this might occur due to the formal specification of the application program being too weak. This problem can easily be removed if all the tests performed on the insecure code are rerun on the final production code. This ensures that the confidence in the insecure code gained from testing is not lost to the secure version. Comparing the text of the programs obtained with the secure and insecure compilers would also suffice. If they are identical, then the verification and test results apply to both compilers for that program.

Compilation-by-proof would also be of use if a compiler implementation was to be produced. Even if the implementation was to be formally verified, additional testing would still be desirable to remove more obvious bugs. However, a problem when testing is of how to know what the correct outputs for particular inputs should be. Compilation-by-proof could provide correct test cases. Execution of the specification would give the required target code for given source programs. These test cases would only need to be executed once before the compiler implementation was developed. The results could then be used to help validate any subsequent implementation produced from the verified specification. This would obviously not overcome the main short-comings of testing, but would at least ensure that the results of the tests that were carried out could be trusted.

3.3 Bootstrapping a Correct Compiler Implementation

Suppose we have verified a compiler specification, and have written an efficient implementation of it in a high-level language. Suppose also that we have verified our implementation against the specification, so we are happy that the implementation is correct. We still have a problem. To execute the implementation we must compile the high-level program into a low-level language. It is the low-level version which we will actually execute. We need a verified compiler before we can obtain our verified compiler implementation in the low-level language! How do we obtain the first verified compiler to start the process?

The first secure compiler could be obtained using one of the other methods suggested in Section 1: it could be written and verified in a low-level language; the program could be compiled into an assembly language for which there is an available proof theory; or a one-off proof of correctness between the compiler's source and target code could be performed. These approaches entail a significant amount of work. A better solution is available, however. It is possible to bootstrap a secure compiler by implementing it in the source language it compiles. Buth *et al.* [5] suggest compiling the compiler using a possibly insecure compiler that is already available. Even if that compiler has not been verified, a great degree of confidence can still be obtained in the resulting code using a bootstrap self-test. The low-level version of the compiler that is produced by the insecure implementation can be used to recompile the high-level version a second time (see Figure 3.3). The resulting code should be identical to that produced by the insecure compiler. The probability that they produce the same incorrect code is very small. It would require that the bug in the host be such that it implants an identical bug in the target code.

An alternative is to use the compiler specification as the first verified

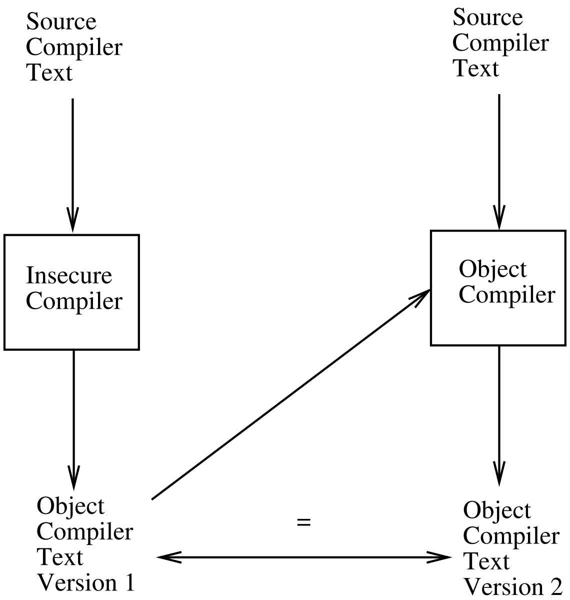

Figure 3.3: Bootstrapping a Secure Compiler using an Insecure One

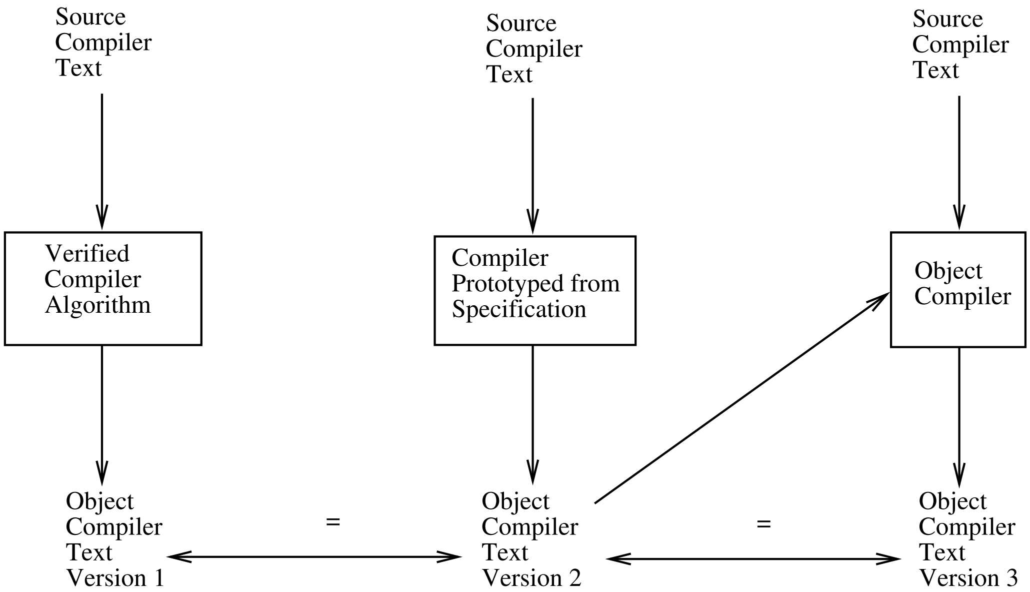

Figure 3.4: Bootstrapping a Secure Compiler using its Specification

compiler. Buth *et al.* [5] suggest manually applying the specification definitions to execute the compiler. As they point out this is intractable due to the large size of the specification and implementation. However we have already seen in the previous section that we can use a theorem prover to execute the specification securely. Furthermore, we need only do this once and then we will have a verified implementation in a low-level language which we can use for further compilations. It can then be used as the compiler to produce verified compiler implementations for other languages.

Since the source text is a verified source text this gives greater assurance than just using an insecure compiler. Only the execution mechanism is insecure. As noted previously the execution strategy can be trusted at least to the same degree as the proof of correctness of the compiler. To gain even more confidence in the code the specification can be prototyped in one or more of the other ways suggested, such as automatically producing a functional language version. The resulting code can then be compared. The two versions originate from the same verified source program: that of the compiler specification. However, they use completely different execution strategies: application of primitive inference rules and interpretation of a functional language. These execution environments are unlikely to introduce the same bug. Therefore if they produce the same code for the compiled compiler, then it is highly likely to be correct.

The above two methods are complementary, since execution by proof can be seen as providing just another possibly insecure compiling mechanism (albeit one that is more secure than other methods). We can apply the compiled code obtained from executing the specification to the source code to give a second version. Once more the resulting code should be identical. This approach is illustrated in Figure 3.4.

Buth *et al.* also note that an interpreter implementation can be bootstrapped in a similar way. This then gives further tests of equality of different bootstrapped code such as comparing the code produced by the interpreter and that produced by the low-level version of the compiler when applied to the compiler.

Chapter 4

Summary

4.1 The Achievements of the Project

The initial aims of this project were to investigate the application of formal methods to the validation of an implementation of a real language, Vista, for a real microprocessor, VIPER. The "implementation" to be verified was actually a compiling algorithm specified in higher-order logic. An algorithm for compiling a significant subset of Vista to a flat, mnemonic assembly language for VIPER has been verified. We have thus succeeded in our initial aim. There have been several other accomplishments of the project.

1. We have shown that if source and target language schemas rather than concrete languages are defined, the compiling algorithm and associated proofs can be generic. This means that a single correctness proof can be performed and then reused for other target machines within the same family with a minimum of additional proof work. Obtaining a verified implementation of a structured assembly language for a particular microprocessor consists of filling in the gaps in the syntax and semantics of the language, and discharging some simple explicit assumptions of the proof. If formally verified compilers are to become commonplace, techniques such as this are vital [21].

2. Previous compiler verification work has used file-based models of input and output, if the problem has been considered at all. In this model input and output consists of reading and writing to files. The ordering of events to different files is not considered. We have shown that a more general model of input/output in which all such orderings are recorded can be incorporated into our framework [18].

3. A reason for formally verifying a compiler is so that correctness properties proven about source programs apply to the object code. This is important as it is the object code not the source code which is actually executed. Often compiler correctness theorems have been considered as ends in themselves. Exceptions to this have been work to combine compiler correctness theorems with microprocessor correctness proofs, producing verified system stacks. There has been no previous work in formally

combining compiler correctness theorems with verification systems for applications programs however. The closest to this has been suggestions that the raw semantics used in the correctness proofs be used to reason about application programs. This is not the most suitable framework to do such proofs. A more suitable framework is a verification system based on a programming logic. We have shown that a compiler correctness theorem can be formally linked with a programming logic [20, 22, 16]. This allows proofs of application programs written in the source language to be performed and theorems asserting that the same properties hold of the compiled code to be obtained. No such formal link has previously been made.

4. We have investigated a novel definition of total correctness in which a single specification gives all the relevent information about the I/O behaviour [20, 22, 16]. Having this information in one place means that it can be verified against more abstract properties such as "an output follows every input". This may be required to prove safety properties of the system for example.

5. Previous compiler correctness work has involved proving more than may strictly be needed. If the target language is deterministic, only half of the normal compiler correctness result is needed. We have shown that this is so. In particular we have shown that with the simplified correctness result, correctness properties of object code can still be obtained from correctness properties of source programs [21, 22, 16].

6. We have verified a compiling algorithm specified in a non-executable logic. The majority of previous work has also only considered the correctness of algorithms. Such proofs are important as the majority of programming bugs arise in the specification rather than the implementation. However, ultimately an executable implementation is needed. We have identified several different ways that implementations can be obtained with differing degrees of security and time cost. We have suggested that a secure approach is to execute the compiler algorithm itself by formal proof. If an insecure compiler implemented from the algorithmic specification is also available then the development cycle of application programs need not be delayed. Indeed, this may actually improve the security of the compiled application code [21, 19, 15]. An alternate way of executing higher-order logic specifications is to translate them to a functional programming language such as ML. In conjunction with researchers at UBC we have investigated this approach with respect to compiler specifications. The UBC tool which automatically produces executable ML code from a HOL specification was used to produce an executable version of the verified Vista compiler [23]. This provides a way of automatically producing an implementation of the compiler specification to use for the development of application programs. Hence, compilation by proof and by translation to ML offer complimentary ways of executing compiler specifications.

7. A highly secure approach to obtaining correct object code is to verify an implementation with respect to the verified algorithm. However, a verified implementation of a compiler in a high level language is not sufficient to obtain verified object code. The compiler must itself be compiled into a low level language before it can be executed. To obtain a secure low-level implementation of a compiler, we apparently need a verified compiler in advance. We have suggested a secure way in which a low level implementation of a compiler can be obtained from a verified compiler specification without already possessing a verified implementation. This approach is complementary to a method that has been previously suggested. By combining the approaches, the probability that the resulting compiler is incorrect can be reduced further [19].

4.2 A Development Methodology

Our work suggests the following methodology to obtain verified object code, given a verified compiler algorithm such as ours with associated derived programming logic. The source program of interest is first compiled using a fast but potentially insecure compiler prototyped from the verified compiler algorithm. This could be produced by automatic translation to ML. The compiled version is thoroughly tested using traditional techniques. Once confidence in its correctness has been achieved, the source program is formally verified against a formal specification using a derived programming logic for the source language. This gives a correctness theorem about the source program. A derived inference rule based on the compiler correctness theorem is then used to give a theorem stating that the compiled code produced by the verified compiling algorithm also satisfies the formal specification of the program. Execution by proof is used to give the compiled code specified by the compiler algorithm. This code is the production version of the program. A correctness theorem which textually includes this code can be obtained for documentation. The production code is then compared with that produced by the insecure compiler. This could be done automatically be the execution-by-proof tool. If the two versions of the code are textually identical then the confidence in the correctness of the code gained by traditional methods also applies to the production code. If they are not, this is an indication that the insecure compiler contains an error. This should be remedied. All the tests could also be rerun on this code. Since the slow but secure method of producing compiled code is only used once at the end of the development cycle, the use of a verified compiler should not slow that cycle. Also the confidence in the correctness of the insecure compiler gained from its frequent use is not lost.

4.3 Errors Found

During the verification of the compiler and derivation of the programming logic, several errors were discovered in the initial definitions. Little attempt to validate the definitions prior to proof was made, so it is unsurprising that errors were found. Many of the errors would probably have been found by traditional testing on a rapidly prototyped implementation, though this may not have been so. It is interesting to note that many of the errors were in the semantic definitions not in the translator. We will briefly consider each error discovered in turn.

A bug was found in the procedure declarations part of the compiler. If a procedure was compiled to a given address, it was assumed when compiling the body that it would start at that address. This was not so, as the body was preceeded by an instruction to store the link address. The only command to use the base address in the subset considered was the While command which used it to calculate jump addresses. The error would thus only have manifested itself in procedures which contained While loops. In this situation a jump to the start of the loop would actually jump to the instruction just prior to the start. A jump out of the loop would actually jump to the jump back to the start of the loop. This bug was corrected and the proof completed.

When adding procedures an accidental change to the definition of the While loop compiler was also made which introduced an error that had not been present in previously verified versions. The address used to jump out of the loop was one less than it should have been, so the loop would never be exited. This bug was corrected and the proof completed.

When implementing the programming logic, two errors were found in the formal semantics of Vista. These errors were found because the expected programming logic rule could not be derived from the Vista semantics.

The first error was in the semantics of machine instructions. It was specified that I/O never occured. However, the program could be halted by the command due to an overflow, for example. In that situation a `DONE` event should occur. There was also a corresponding error in the semantics of Visa which meant the compiler correctness proof did not pick up the mistake. The semantics was changed and the compiler correctness theorem reproved.

The second mistake, concerned the `FINISH` statement at the end of the program. This should compile to a Stop command. The semantics of a program should always end in a Halt state. This was not so. The implementation allowed a program to terminate in a Run state. This was not corrected. However, the programming logic rule does not allow programs to be proved correct if they do not end in a Halt state. This must currently be achieved with an explicit Stop command.

It was noted during the verification that the initial `PROGRAM` statement should have been compiled to a Jump to the base address of the body of the program. On reset, VIPER is initialised with address zero in the program counter, so all programs should start there. However, the compiler compiles procedures before the main body, thus execution would start with the first procedure body. As reset was not modelled in the semantics, this error was not picked up by any of the proofs. An explicit assumption in the proofs was that execution would start with the program counter holding the start address of the program. This emphasises one of the weaknesses of formal proofs: the proof is only as good as the assumptions made and of the models used. However, the error was noticed because of the explicit assumption in the correctness theorem.

This error was not corrected, though the verification work involved would not have been too great. This also highlights an anomily in the design of Vista and its informal documentation. There appears to be no mechanism provided in the informal language definition which ensures that address zero is in a code region. A user program could, for example, declare it to be part of the data region or not part of any region. Pressing reset with such a program loaded would result in random data being executed.

When devising the formal semantics of the Vista subset many ambiguities were also found in the informal documentation of the semantics.

4.4 Reducing the Work Required to Formally Verify a Compiler

We have suggested that for verified compilers to become commonplace, the work required to produce one must be minimised. This minimisation falls into two categories: reducing the work required to produce an initial verified compiler and reducing the work required to then produce further verified compilers for other systems. We now briefly overview the techniques we employed to achieve these aims.

We have made use of several ways to reduce the work needed to initially verify a compiler. First we noted that the task was made more tractable if split into specification correctness and implementation correctness. We then noted that specification correctness alone was sufficient as execution by proof and automatic conversion to ML then provide secure execution mechanisms. We also noted that for deterministic languages only half of the proof effort is required. This is because only one of the two theorems normally considered to represent compiler correctness are actually needed to obtain the result that properties proved of source programs also hold of the compiled code. Splitting the compiler algorithm itself into layers also made the proof more tractable, ensuring that the proof of each level was concerned with only a few concepts. Finally, by noting that, for example, the semantics of all the comparison operations have a similar structure, with identical operations being performed at the lower level, we can do a single proof which applies to them all. The alternative would be to repeat a similar proof for each separate instruction.

To speed the re-targeting of the compiler theory to other architectures we also employed several methods. The first was to use generic theories to allow a single theory to apply to an architecture family for a given source language. The second was to split the compiler into multiple passes. This technique leaves open the possibility of replacing a level or set of levels with ones specific to a new machine. Only the verification work involving the changed levels would be needed to be redone. In our work, the Visa and Kaa languages abstracted away from the full details of the VIPER microprocessor. Only features needed to implement Vista were provided. This could be a disadvantage if the lower levels of the compiler were to be used as the target for other high level languages. Then useful features of the underlying processor might not be available. It would be of use if the compiler was to be retargeted to different microprocessors however.

Only the very bottom level proof in which the full description of the processor was used would need to be redone for machines which had all the features used. Since we did not implement this bottom level, even without the generic features of our proofs, the proofs would apply to a wide range of VIPER designs. If this was done in the simplest possible way, new features of the different designs would not be exploited of course. However, old programs would still run on the new machines with assurances of their correctness being quickly obtained. Even without the above techniques, much of the underlying theory and experience gained is likely to be of use in further compiler proofs. A second proof could be done much more quickly than the first.

4.5 Philosophical Considerations

The work could not have been completed with any degree of assurance without some form of mechanisation. We have found the HOL system to be a versatile tool, giving the user control over the proof whilst also providing some automation of mundane tasks. It is useful for the user to direct the proof, as a greater understanding of why the program is correct is obtained. This is one of the advantages of performing formal proofs. It gives more faith that the theorems proved are useful, and makes it easier to correct mistakes when theorems cannot be proved.

The only axioms we used were those upon which the HOL system is built. This approach is very time consuming since all theorems that are not pre-proved in the system have to be proved. However, it increases our confidence in the correctness of the work. Proving theorems which are true is often straightforward. The effort is expended in coming up with the suitable definitions and lemmas in the first place. We spent much time working with definitions and goals that were later modified when lemmas could not be proved. With an axiomatic approach, such mistakes could be missed. Using an expressive specification language such as higher order logic helps avoid mistakes. Whilst perhaps not essential, it is useful in that ideas can be expressed more naturally than in a first order logic and so mistakes are less likely to be made. For example, when defining the semantics we frequently pass semantic relations as arguments. Other techniques such as animation of definitions prior to formal proof would have helped to avoid some of the problems. Our use of generic definitions also saved much proof work. Not only can the work be re-targeted to different microprocessors, but also, for example, a single correctness proof suffices for all binary machine instructions.

The fact that a compiler correctness theorem has been proved, whether by hand or by machine, does not give a guarantee that absolute faith can be placed in the compiler. Hand proofs often contain mistakes. Theorem provers can contain bugs. Even if the theorem is valid, it may not describe the real world sufficiently accurately. Correct code produced by the compiler might be corrupted before it is executed; the wrong version of the code might be used or the code might be loaded to the wrong location. Problems such as these correspond to the explicit assumptions in the correctness theorem not being adhered to. Alternatively, implicit assumptions might be invalid. For example, the semantics of the target machine used in the proof might not correspond to the actual semantics of the machine used. Also, the correctness theorem might simply be inadequate. For example, it might not guarantee that execution of the compiled code terminates when the source program does. It might merely state that if the compiled code did terminate it would have the same meaning as the source program.

Some of these problems can be alleviated by proving other theorems. For example, a loader to be used with the compiler could be verified. Similarly the correctness theorem could be combined with other theorems about the correctness of the hardware with respect to the semantics. It could also be proved (as we did) using the compiler correctness theorem that if the source code is totally correct so is the compiled code. However, at some point we move from the mathematical world into the real world. It is always possible that the models we have reasoned about do not correspond to reality to a sufficient extent. The main advantage of performing verification is that it forces us to thoroughly examine the system in question and the reasons we believe it to be correct.

4.6 Conclusions

The project has illustrated the feasibility of using the HOL system to verify compilers for real languages. Whilst we only considered a subset of the language, the proofs required for many of the additional features would be straightforwardly based on those considered. For example, the remaining atomic commands such as assignment to the B register are similar to those considered; other forms of declarations are similar to data declarations; much of the infra-structure for region declarations is in place; methods of escaping from loops could be treated analogously to the Stop command; other forms of structured commands such as case statements are variations on the proof of the While loop; adding preface blocks would also be a simple extension of the While loop. The principles behind these are similar to those implemented, so no new insight would have been gained by implementing them. They could be implemented and verified with minimal effort. Omitted features that would need more effort include vectors, nested procedures and the ability to change regions. However, no great difficulty is foreseen with these features.

Verifying a compiler implementation can best be done by splitting the task into specification correctness and implementation correctness. That is, we first verify an algorithmic version. We then show that the implementation is correct with respect to the algorithm. These two proofs can be combined to give the required correctness theorem about the implementation. In the specification correctness proof the correspondence between the semantics of the source and target language are considered. This is simplified because the compiler specification is given in a logic with clean semantics. Implementation details do not need to be considered. In the implementation correctness part of the proof, the implementation is compared with the specification. The details of the semantics of the programming language in which the compiler is implemented must be considered. However, the semantics of the source and target languages of the compiler do *not* need to be considered. Only their syntax is important.

Implementations are easier to verify if they have a structure close to that of the algorithm. This also makes it easier to write a correct implementation in the first place. If the algorithmic specification does not have the structure intended to be used for the implementation, a second specification should be given with an appropriate structure and shown correct with respect to the original, since specifications are easier to reason about than implementations.

It is possible to build a generic compiler correctness theory for a structured assembly language which is applicable to a family of target machines. This is done using type variables in place of the unspecified parts of the system and using a representation tuple to provide the missing semantic information. Targeting the theory to a given machine involves instantiating the types, providing a value for the tuple and proving some simple theorems.

The oracle model of I/O behaviour can be incorporated into both a relational semantics and interpreter semantics. These semantics can be used in a compiler correctness proof. This use of oracles for I/O in conjunction with a relational style semantics appears to be very versatile. It could be adapted for other languages.

Application program verification systems can be formally combined with a compiler correctness statement within HOL. This allows results about the compiled code to be formally deduced from results proved about source programs. It can also be generic and so allow generic source programs to be verified with results about generic object code being obtained. An additional advantage is that deriving the semantics embodied in the verification system can uncover mistakes in the semantics used for the compiler correctness proof.

Once a compiler algorithm has been verified, it can be used to produce high assurance implementations in various ways other than by verifying it against an implementation: the algorithm may be executable itself; it might be automatically converted to an executable language; it might be written in an executable language that is semantically embedded in the logic of the theorem prover; or it might be automatically executed by theorem proving giving as a side effect a theorem stating that compiling the source code gives the target code. Translation to ML provides a quick and fairly secure method of executing suitably written HOL definitions. Execution by proof is very secure but slow. However, it can be used in conjunction with a fast but insecure production compiler, perhaps produced by translation to ML. The development cycle of application programs is then not hindered. This can also increase our confidence in the correctness of the compiled code. This means that a verified compiler implementation is not strictly needed. Execution by proof can be used to produce correct test cases with which to validate an implementation if one is produced. It can also be used to do generic compilation.

If an implementation of the compiler in the compiler's source language is verified against the algorithm, then a secure implementation in a low-level language can be bootstrapped from it either using an insecure compiler or by executing the specification in the theorem prover. These two approaches are complementary. Our confidence that the bootstrapping approaches give correct code relies on intuitive arguments that the probabilities of particular events are negligible and that different methods are independent and so will not introduce the same bug. Such arguments can turn out to be fallacious such as the once held belief that code written by independent programming teams would not contain identical errors. In practice it often turns out that they do because their past experiences are not independent. Even if the probability of errors being missed via one method is thought to be low, combining complementary methods can not do harm.

4.7 Further Work

The project has highlighted further areas where research is warranted.

Vista is a relatively simple language and VIPER a simple microprocessor. Further work is required to show that similar results can be obtained for more complex languages and target microprocessors, and also for more complex compilers such as optimising compilers.

Previous verification work has been concerned with the verification of a single compiler. For verification to become widespread it must be demonstrated that once a result is obtained for one compiler, similar results can be quickly obtained for different compilers. We made a preliminary investigation of one such generic approach. We have shown that a compiler for a structured assembly language can be verified which is applicable to different target machines of the same family. This suggests that a verified generic compiler could be produced which would be applicable to a whole architecture family. Further work is required to demonstrate this, and also to show that the results scale to more complex commercial architectures.

In our work, the same details were abstracted from all levels of the compiler description in the generic theory. This meant that only features common to all levels could be abstracted out. However, in the multiple-pass approach, there are only limited differences between the source and target languages of each level. All other details remain unchanged. This suggests that the generic techniques could be applied on a level basis. Each level would have its own generic features, thus reducing the work done on that level and allowing it to be reused for a wider range of source and target machines. Levels could be combined by instantiating their particular features to give generic theories with only the common parts abstracted away. This technique could help scale our methods to more complex source languages. Since we use interpreter semantics for the low-level languages, a generic interpreter theory such as that suggested by Windley [51] could also be used.

Techniques such as the above will only partially help if the new source languages or target architectures differ widely from those previously considered. It is, however, possible that proof scripts can be quickly adapted to new languages if they are written and structured with this in mind. One possibility worth investigating is to develop a coherent toolkit of useful lemmas and verified language constructs, which can be incorporated into proofs. The former would also simplify the proofs of similar constructs within a single language, in a similar way to the way that using a generic binary machine function instruction allowed one proof to suffice for all the binary machine functions of the language.

We have suggested that execution by proof can be a useful tool to perform real compilation. Further work is required to demonstrate this for real application programs. Execution by proof requires inference rules to be derived which perform the execution for a particular compiler. An investigation of ways to automate the generation of these rules would also be of use. Techniques to further optimise the proof process will also need to be developed. For example, a single pass compiler would be more efficient than the multiple pass compiler more suited to verification. It may be possible to automatically convert a multiple pass description to a single pass one automatically using proof. The two descriptions would then be assured to be equivalent. This would also give a base from which to prototype more efficient compiler implementations such as by translation to ML.

We have also suggested how a compiler implementation written in its own source language could be used to bootstrap a secure low level version of the implementation. This approach needs to be demonstrated.

The compiler we verified produced symbolic assembly code, rather than machine code. A further assembly step is required to produce machine code. Such an assembler would be very simple. Its proof would be similar to the proof of the Visa to Kaa assembly step. However, the proof would be much simpler, since the semantics of the two levels would be basically the same. The difference is mainly in the syntax: symbolic instruction names against boolean valued words. Similarly, combining this correctness theorem with those of the other levels would be similar to that of combining the other two levels. As a consequence, little new insight would be gained from verifying such an assembler. However, it would allow the compiler correctness work presented here to be formally linked to the proofs of the VIPER microprocessor. More work is required to investigate how compiler correctness proofs connected to high level proof systems can and should be combined with microprocessor proofs. It is clear that we require the compiler to ensure that the object code to possess the properties that we have shown the source code to posses. The properties we wish to hold at and below the register level of a microprocessor are less clear.

The formal semantics we have given for Vista can only directly reason about the properties of terminating programs. All that can be said about non-terminating programs is that they do not terminate. This is because for a non-terminating program, there would be no final state corresponding to the initial state, for which the semantic relation would be true. However, many interesting programs do not terminate. It is their I/O behaviour which is important rather than a final state. Our semantics could be used to reason about non-terminating programs indirectly, by specifying the properties of terminating segments of the program. Such fragments could possibly be generated from an annotated program using a technique similar to verification condition generation. Alternately, our semantics could be modified to handle non-terminating programs. Further research is required.

We have assumed that specifications are given in source language terms such as using variable and channel names rather than addresses. This will be natural if a refinement methodology is used, so that the specification is a refinement of some higher-level requirements specification. The compiler is then providing a further automatic refinement step to a lower level language. However, we may need to know, for example, which particular address a channel refers to, to ensure that an I/O device is correctly connected. We thus would need aspects of the specification to be in target language terms. In our work, the problem is not too great because few data abstraction facilities are provided: the high-level concepts are not far removed from the target level. However, this would not be so for more complex languages. This demands further investigation.

Acknowledgements

Many people gave help and advice during this project for which I am grateful. I am particularly indebted to Gavin Bierman, Mike Gordon, Jeff Joyce, John Kershaw, Clive Pygott and Sreeranga Rajan. The members of the Automated Reasoning Group at Cambridge provided a stimulating research environment.

Bibliography

[1] Mark Aagaard and Miriam Leeser. Verifying a logic synthesis tool in Nuprl: A case study in software verification. In *Proceedings of the 4th Workshop on Computer Aided Verification*, 1992.

[2] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van-Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Theorem Provers in Circuit Design*, pages 129–156. North-Holland, 1992.

[3] Jonathan Bowen, He Jifeng, and Paritosh Pandya. An approach to verifiable compiling specification and prototyping. In P Deransart and J Maluszyński, editors, *Programming Language Implementation and Logic Programming*, volume 456 of *Lecture Notes in Computer Science*, pages 45–59. Springer-Verlag, 1990.

[4] Robert S. Boyer and Yuan Yu. Automated proofs of object code for a commercial microprocessor. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 416–431. Springer-Verlag, 1992.

[5] Bettina Buth, Karl-Heinz Buth, Martin Fränzle, Burghard v. Karger, Yassine Lakhneche, Hans Langmaack, and Markus Müller-Olm. Provably correct compiler development and implementation. In V. Kastens and P. Pfahler, editors, *Compiler Construction '92*, volume 641 of *Lecture Notes in Computer Science*, pages 141–155. Springer-Verlag, 1992.

[6] Albert John Camilleri. Executing behavioural definitions in higher order logic. Technical Report 140, PhD Thesis, University of Cambridge, Computer Laboratory, February 1988.

[7] Juanito Camilleri. Symbolic compilation and execution of programs by proof: A case study in HOL. Technical Report 240, University of Cambridge, Computer Laboratory, December 1991.

[8] Rachel Cardell-Oliver. Using higher order logic for modelling real-time protocols. In S. Abramsky and T. S. E. Maibaum, editors, *Proceedings of TAPSOFT'91*, Lecture Notes in Computer Science, pages 259–282. Springer-Verlag, 1991.

[9] William C. Carter, William H. Joyner, Jr., and Daniel Brand. Microprogram verification considered necessary. In Saki P. Ghosh and Leonard Y. Liu, editors, *AFIPS Conference Proceedings 1978 National Computer Conference*, volume 47, pages 657–664. AFIPS Press, June 1978.

[10] Laurian M. Chirica and David F. Martin. Towards compiler implementation correctness proofs. *ACM Transactions on Programming Languages and Systems*, 8(2):185–214, April 1986.

[11] D. L. Clutterbuck and B. A. Carré. The verification of low level code. *Software Engineering Journal*, pages 97–111, May 1988.

[12] Avra Cohn. The correctness of a parsing algorithm in LCF. Technical Report 21, University of Cambridge, Computer Laboratory, 1982.

[13] Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning 5*, (5):127–138, 1989.

[14] Avra Cohn and Robin Milner. On using Edinburgh LCF to prove the correctness of a parsing algorithm. Technical Report 20, University of Cambridge, Computer Laboratory, 1982.

[15] Paul Curzon. Correct compilation by proof. Submitted to *ACM SIGSOFT '93: Symposium on the Foundations of Software Engineering*.

[16] Paul Curzon. Deriving correctness properties of compiled code. To appear in *Formal Methods in System Design*, Kluwer. This is an extended version of [22].

[17] Paul Curzon. A structured approach to the verification of low level microcode. Technical Report 215, PhD Thesis, University of Cambridge, Computer Laboratory, February 1991.

[18] Paul Curzon. Compiler correctness and input/output. In *Proceedings of the 3rd IFIP Working Conference on Dependable Computing for Critical Applications*, Dependable Computing and Fault-Tolerant Computing. Springer-Verlag, 1992.

[19] Paul Curzon. Of what use is a verified compiler specification? Technical Report 274, University of Cambridge, Computer Laboratory, November 1992.

[20] 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*, pages 403–408. Springer-Verlag, 1992.

[21] 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.

[22] Paul Curzon. Deriving correctness properties of compiled code. In L. Claesen and M. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*, IFIP Transactions A-20. North-Holland, 1993.

[23] Paul Curzon, Sreeranga Rajan, Jeff Joyce, and Carl Seger. Executing compiler specifications in higher order logic. Submitted to *the Seventh International Workshop on Software Specification and Design*, 1993.

[24] Joëlle Despeyroux. Proof of translation in natural semantics. In *IEEE symposium on Logic in Computer Science*, pages 193–205, 1986.

[25] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Mathematics*, volume 19, pages 19–32, 1967.

[26] P. Gloess. An experiment with the Boyer-Moore theorem prover; a proof of correctness of a simple parser of expressions. In *Proceedings of the 5th Conference on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 154–169, 1980.

[27] K. G. W. Goossens. Operational semantics based formal symbolic simulation. In L. Claesen and M. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*, IFIP Transactions A-20. North-Holland, 1992.

[28] Michael J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 387–439. Springer-Verlag, 1989.

[29] Kelly M. Hall and Phillip J. Windley. Simulating microprocessors from formal specifications. In L. Claesen and M. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*, IFIP Transactions A-20. North-Holland, 1992.

[30] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.

[31] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2:335–355, 1973.

[32] Jeffrey J. Joyce. Totally verified systems: Linking verified software to verified hardware. In M. Leeser and G. Brown, editors, *Specification, Verification and Synthesis: Mathematical Aspects*. Springer-Verlag, 1989.

[33] Jeffrey J. Joyce. A verified compiler for a verified microprocessor. Technical Report 167, University of Cambridge, Computer Laboratory, March 1989.

[34] Jeffrey J. Joyce. Generic specification of digital hardware. Technical Report 90–27, The University of British Columbia, Department of Computer Science, September 1990.

[35] Jeffrey J. Joyce. Private communication, 1990.

[36] Paul Arthur Lamb. *A verification condition generator for Intel 8080 microprocessor assembly language programs.* PhD thesis, George Washington University, 1982.

[37] W. D. Maurer. Proving the correctness of a flight-director program for an airborne minicomputer. In *Proceedings of the ACM SIGMINI/SIGPLAN Interface Meeting on Program Systems in the Small Processor Environment*, pages 103–108, 1976. Appeared as a special issue of SIGPLAN Notice V11(4), April 1976.

[38] W. D. Maurer. An IBM 370 assembly language verifier. In P. A. Willis, editor, *Proceedings of the 16th Annual Technical Symposium on Systems and Software: Operational Reliability and Performance Assurance*, pages 139–146. ACM, June 1977.

[39] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. In J. T. Schwartz, editor, *Proceedings of Symposia in Applied Mathematics*, volume XIX, pages 33–41, 1966.

[40] Thomas F. Melham. Abstraction mechanisms for hardware verification. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 267–292. Kluwer Academic Publishers, 1988.

[41] MoD: Interim defence standard 00-55 on the procurement of safety critical software in defence equipment, 1991.

[42] J. Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5:461–492, 1989.

[43] I. M. O'Neill, D. L. Clutterbuck, P. F. Farrow, P. G. Summers, and W. C. Dolman. The formal verification of safety-critical assembly code. In W. D. Ehrenberger, editor, *Proceedings of the IFAC Symposium on Safety of Computer Control Systems 1988 (Safecomp '88) Safety Related Computers in an Expanding Market*, 1988.

[44] Lawrence Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[45] Wolfgang Polak. *Compiler Specification and Verification*, volume 124 of *Lecture Notes in Computer Science.* Springer-Verlag, 1981.

[46] Sreeranga Rajan. Executing HOL specifications: Towards an evaluation semantics for classical higher order logic. In L. Claesen and M. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*, IFIP Transactions A-20. North-Holland, 1992.

[47] David Shepherd. Using mathematical logic and formal methods to write correct microcode. In *Proceedings of the 20th Annual Workshop on Microprogramming*, 1988. Appeared as a special issue of *Sigmicro Newsletter*, 19(1 and 2), June 1988.

[48] David Shepherd. Verified microcode design. *Microprocessors and Microsystems*, 40(10):623–630, December 1990.

[49] Todd G. Simpson. Design and verification of IFL: a wide-spectrum intermediate functional language. Technical Report 91/440/24, University of Calgary, Department of Computer Science, July 1991.

[50] Susan Stepney, Dave Whitley, David Cooper, and Colin Grant. A demonstrably correct compiler. *Formal Aspects of Computing*, 3:58–101, 1991.

[51] P. J. Windley, K. Levitt, and G. C. Cohen. The formal verification of generic interpreters. Technical Report 4403, NASA Contractor Report, October 1991.

[52] William D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5:493–519, 1989.