Technical Report

UCAM-CL-TR-274
ISSN 1476-2986

Number 274

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Of what use is a verified compiler specification?

Paul Curzon

November 1992

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Of What Use is a Verified Compiler Specification?

Paul Curzon

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

Email: pc@cl.cam.ac.uk

Abstract

Program verification is normally performed on source code. However, it is the object code which is executed and so which ultimately must be correct. The compiler used to produce the object code must not introduce bugs. The majority of the compiler correctness literature is concerned with the verification of compiler specifications rather than executable implementations. We discuss different ways that verified specifications can be used to obtain implementations with varying degrees of security. In particular, we describe how a specification can be executed by proof. We discuss how this method can be used in conjunction with an insecure production compiler so as to retain security without slowing the development cycle of application programs. A verified implementation of a compiler in a high-level language is not sufficient to obtain correct object code. The compiler must itself be compiled into a low-level language before it can be executed. At first sight it appears we need an already verified compiler to obtain a secure low-level implementation of a compiler. We describe how a low-level implementation of a compiler can be securely obtained from a verified compiler specification.

1 Introduction

Traditionally, testing has been used to ensure that software is reliable. However, it is normally infeasible to test all possible combinations of inputs to a program so testing can easily miss errors. Formal verification has been advocated as a complementary approach. Here mathematical techniques are used to prove that the program has the desired properties whatever the inputs. The program semantics and the required specification are modelled using a logic. By manipulating these descriptions in a formal system it can be deduced whether or not the program meets the specification.

Program verification is normally performed on source code. However, it is the object code which is executed and so which ultimately must be correct. The compiler used to produce the object code must thus be correct. Otherwise, it could introduce bugs into the object code which are not present in the source program. The object code then would not meet the specification even though the source code does. To

overcome this, formal methods can be applied to the compiler. Informally, the abstract specification for the compiler is that it generates correct object code—i.e., code which has the same semantics as the source program from which it was derived. The majority of the formal compiler verification literature is concerned with the verification of algorithmic compiler specifications written in a logic. It is proved that an algorithm generates correct code. To obtain correct object code, we must execute the verified compiler, however. It therefore seems we really wish to verify an implementation written in a programming language.

We argue that this need not be so. We discuss ways that implementations can be obtained from verified algorithmic specifications with varying degrees of security. In particular, we describe a way in which the specification itself can be executed by formal proof. This involves proving a theorem which states that applying the compiling algorithm to a program of interest gives particular object code. Using a mechanized proof assistant such as HOL [19], a theorem like this can be obtained automatically with a high degree of security. Furthermore, the object code in question can be automatically derived as part of the formal proof. Previously this technique has been used to test definitions before using them in a formal proof. We suggest that it can also be used to perform high-integrity compilation to produce secure production code. By using compilation by proof in conjunction with an unverified production compiler constructed from the same specification, high security is achieved without hindering the development cycle.

Whilst compilation by proof provides a relatively secure method of compilation, in the long term it may still be advantageous to verify compiler implementations. However, a formally 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 an already verified compiler. We suggest that execution by proof provides a secure way of circumventing this problem. It can be used to obtain a low-level implementation of a compiler from a verified compiler specification. This approach is complementary to a method that has been suggested elsewhere [4]. By combining the approaches, the probability that the resulting compiler is incorrect can be reduced further.

The remainder of this paper is structured as follows. In Section 2 we discuss various ways that correct object code can be obtained, and suggest that verified compilers offer the most attractive solution in the long term. In Section 3 we discuss how the compiler correctness problem can be split into compiler specification correctness and compiler implementation correctness. In Section 4 we discuss various ways in which a correct compiler implementation can be obtained from a correct specification. In particular we describe how an algorithmic compiler specification can be executed securely using a theorem proving system such as HOL. We also discuss how this slow but secure compilation method can be used in conjunction with fast but insecure production compilers so as not to slow the application program development cycle. If the production compiler is used widely this can also increase our confidence in the final code. In Section 5 we discuss a method which has been previously suggested for bootstrapping a correct compiler implemented in a

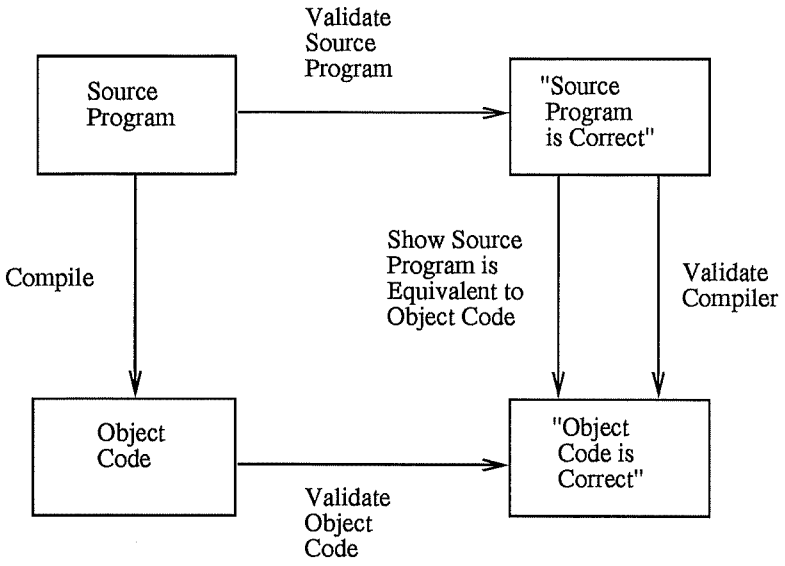

Figure 1: Alternate ways of Obtaining Validated Object Code

low-level language, without already possessing one. We then describe how executing a verified compiler specification using a theorem prover gives an alternative, though complementary way of achieving the same end. Finally, in Section 6, we summarise the work and draw conclusions.

2 Why Compiler Verification is Important

Formally verified object code can be obtained in several ways (see Figure 1).

- The program can be written and verified in the object language.
- The program can be written in a high-level language, but verification performed on the compiled code.
- The program can then be written and verified in a high-level language and then formally shown to be equivalent to the compiled code in a one-off proof.
- The compiler can be verified. The program can then be written and verified in a high-level language.

If the program is written in a low-level language the advantages of high-level programming are lost. Mistakes are easy to make and hard to find. Formal verification of low-level code is much harder than for high-level programs. 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 [16] modified to deal with low-level code. They provide a verification condition generation program which embodies the semantics of the assembly language. Maurer [25, 26] 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 [24]. 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 [10], and also of Z8002 code used in the fuel control unit of the RB211-524G jet engine [29].

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 [8]. A hybrid approach using verification condition generation techniques to verify bit-level microprograms has also been suggested [13].

To retain the advantages of high-level programming, but ensure that the object code itself is validated, the program can be written in a high-level language, and validation of the resulting compiled code performed. This is the normal procedure when testing is the validation method used. An advantage is that industrial strength compilers can be used, so the efficiency of the object code does not need to be compromised for the sake of correctness. 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 because the object code is machine generated. Understanding why it is expected to be correct, a prerequisite for formal verification, is much harder especially if an optimising compiler is used. This approach was adopted by Boyer and Yu [3]. 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.

If the problems of verifying low-level code are to be avoided, a method of converting correctness theorems about source programs to correctness theorems about object code must be devised. One way this may be done is by formally proving that the source program of interest is equivalent to the compiled code in a one-off proof. Validation can then be performed on the source program and the result will be 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. Also such equivalence proofs are not necessarily straightforward. Knowledge of how the object code implements the source program is again needed. This problem is similar to that encountered in the approach of Boyer and Yu of

showing that object code is equivalent to an algorithm. However, here it is more difficult as the semantics of the source language are unlikely to be as clean as the pure logic used to describe the algorithm. Shepherd [35, 36] adopted an approach similar to this to verify 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.

Repeated equivalence proofs such as the above can be avoided by formally verifying the compiler itself. Source programs can be formally verified and these results apply to object code produced by the verified compiler. A single proof (that of the compiler) is sufficient to show that all object programs produced using the compiler correspond to their source programs. However, formally verifying a real 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.

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 will be of more use.

There has been interest in formally verifying compilers from the early days of verification technology, the first work being that by McCarthy and Painter [27] 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 [23]. Notable work, includes that of Polak [31] 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 [28, 38].

The majority of compiler correctness work has been concerned only with the correctness of code generators. Exceptions to this include Polak's work [31] and that of Chirica and Martin [9] 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 [12, 11, 17]. 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. We will only be considering code generation correctness in the remainder of this paper.

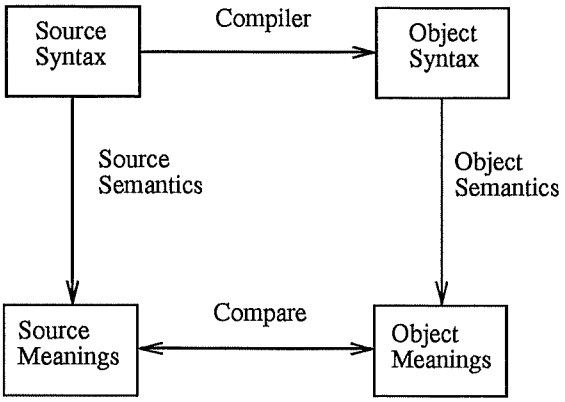

Figure 2: An Abstract Specification of Compiler Correctness

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 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. An algorithm can be shown correct with

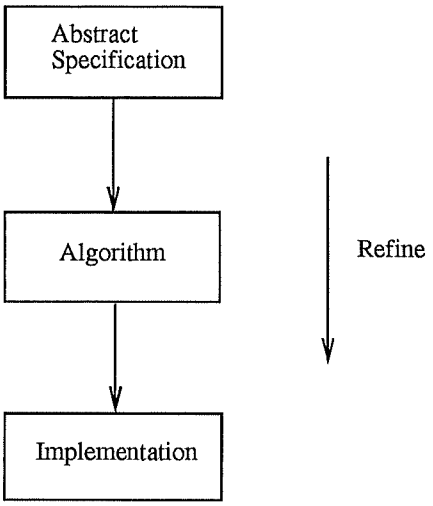

Figure 3: The Refinement Hierarchy

respect to the abstract specification. That is, it can be proved that the semantics of source programs 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, termed *compiler specification 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 [31]. However, a simpler alternative is to use a verified algorithm as a refinement step towards obtaining a verified implementation (see Figure 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 [3]. A similar split has also been used in the verification of protocols [7].

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

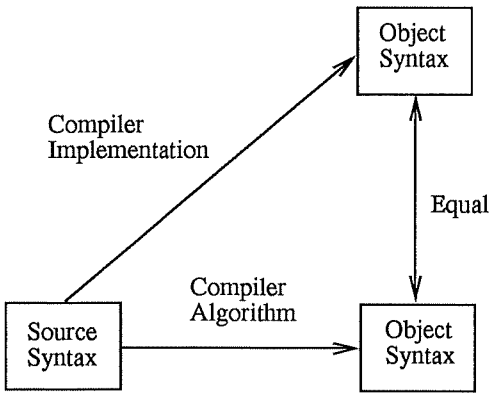

Figure 4: Verifying a Compiler Implementation against an Algorithm

which the compiler is implemented must be 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 [9], Simpson [37] and Buth *et al.* [4]. It is illustrated in Figure 4.

We thus prove the following about the algorithm:

```
⊢ AbstractCompilerSpec CompilerAlgorithm
```

and about the implementation

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

Combining these we obtain the required theorem as illustrated in Figure 5:

```
⊢ AbstractCompilerSpec CompilerImpl
```

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, 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 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 6.

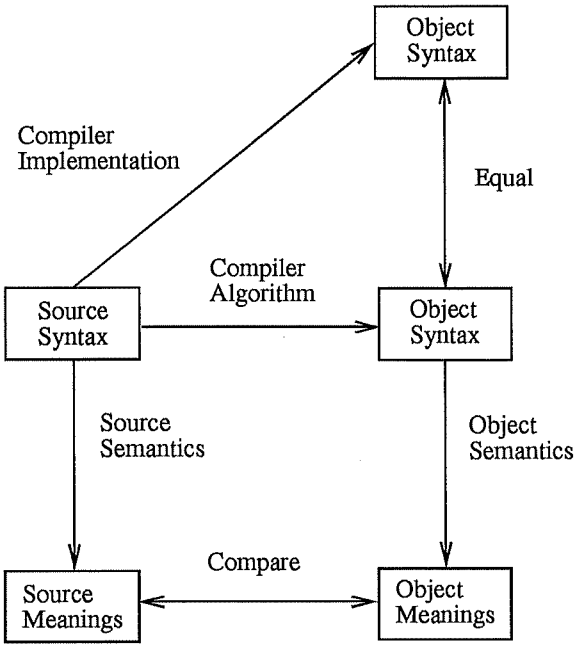

Figure 5: Combining Specification Correctness and Implementation Correctness

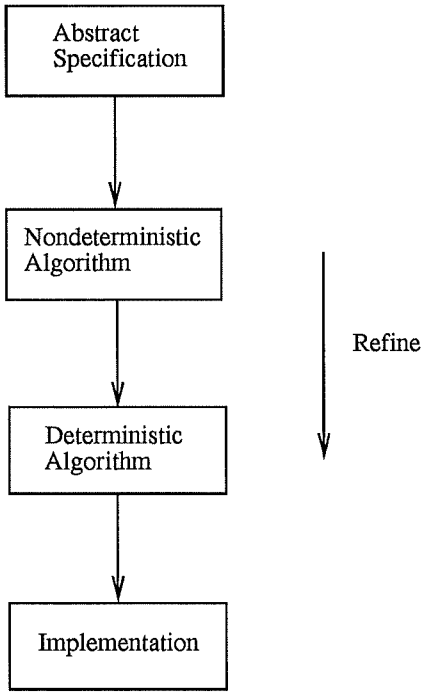

Figure 6: The Refinement Hierarchy with Non-deterministic Specifications

4 Implementations from Specifications

In this section we discuss different ways that relatively secure implementations of compilers can be obtained from a verified algorithm. In particular we suggest that execution by proof provides a convenient and secure method of obtaining compiled code.

The implementation can be informally 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 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.

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 concrete compiler. The work done at Computational Logic Inc. where the Boyer-Moore logic was used is a case in point[28, 38]. 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 be used to do compilation. For more expressive logics which are not executable, it may still be possible to write an interpreter for an executable subset. Algorithms specified using the subset can 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.

Alternatively, it might be possible to rapidly prototype a specification 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 [5]. Hall and Windley [21] have adapted this approach to allow microprocessor specifications to be executed. Rajan also uses similar techniques to automatically translate general deterministic higher-order logic specifications into HOL ML code [33]. A verified compiling algorithm for a subset of Vista [14] is being used as a case study to test this tool [34]. 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.

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 then 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 [22], and VDM

style specifications [20]. 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.

A more secure approach is to use formal proof to perform the compilation [15]. 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 [30]. 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 can also be used to test the definitions prior to verification and 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 [6]. Valuable feedback was obtained to help ensure the definitions were correct before verification was attempted. Goossens [18] 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.

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

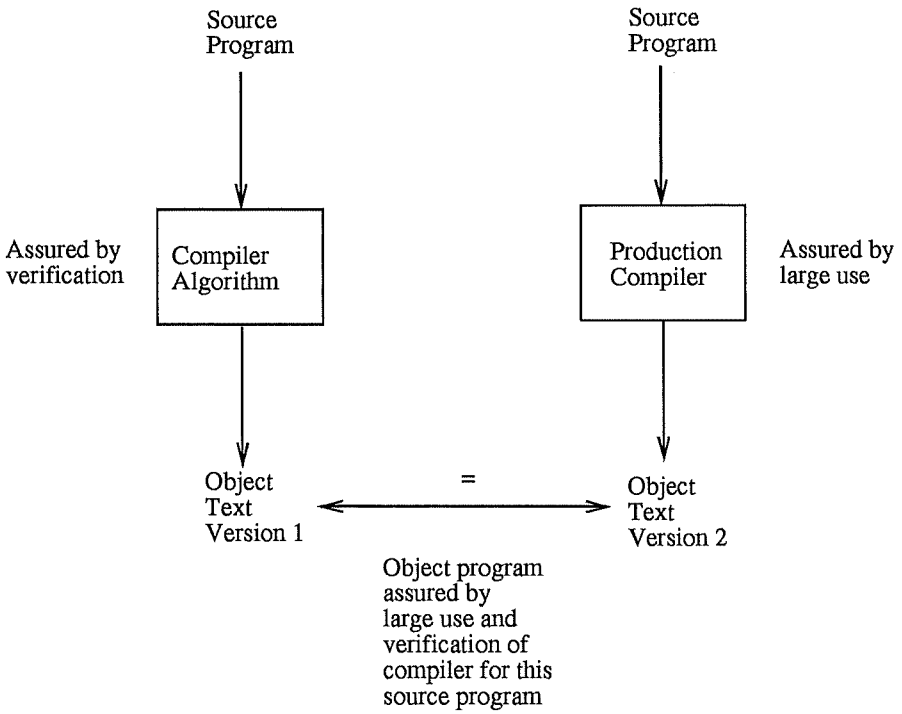

Figure 7: Comparing the Object Code from Different Sources

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.

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 insecure compiler, the extra assurance is retained (see Figure 7). 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.

Comparing the code output from different compilers can also be used as a check on the security of implementations produced in the other ways suggested above. For example the code produced by a compiler rapidly prototyped from the specification, is a source of correct results with which to test a production compiler.

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.

5 Bootstrapping a Correct 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 2: 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.* [4] 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 8). 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 compiler. Buth *et al.* [4] suggest manually applying the specification definitions to execute

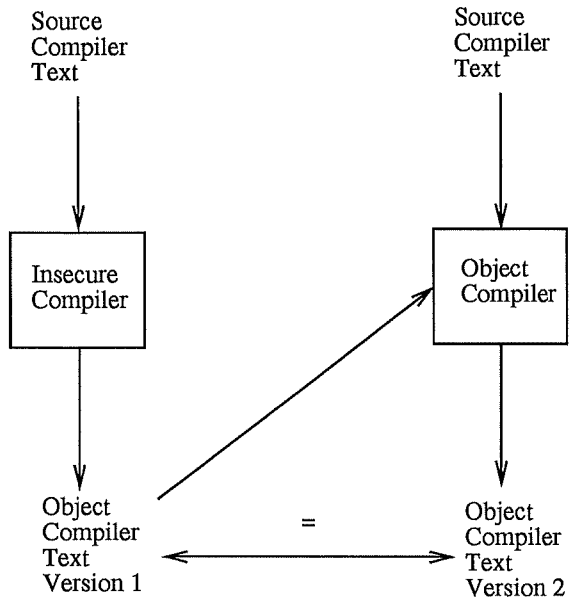

Figure 8: Bootstrapping a Secure Compiler using an Insecure One

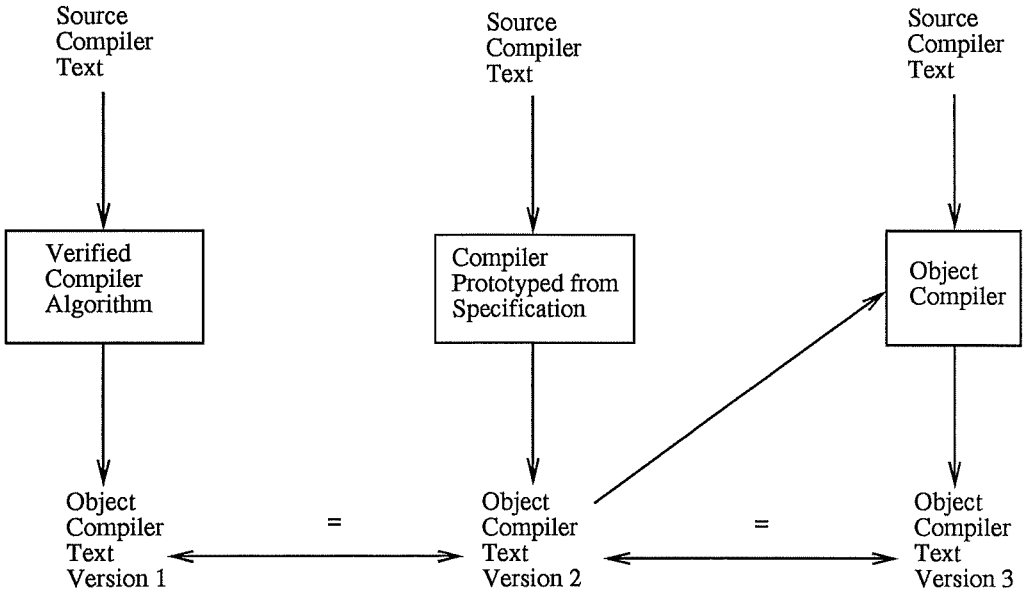

Figure 9: Bootstrapping a Secure Compiler using its Specification

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 9.

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.

6 Conclusions

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.

If two different compilers are produced to meet the same specification, whatever the method, a useful check on their security can be made by running both on the program in question and comparing the results.

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. The latter method is secure but slow. However, it can be used in conjunction with a fast but insecure production compiler so the development cycle is not hindered. This can also increase our confidence in the correctness of the compiled code.

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.

Unfortunately, "*million-to-one chances crop up nine times out of ten*" [32]. 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 only help.

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 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.

Acknowledgements

I am grateful to Mark Aagaard, Gavin Bierman, Philip Core, Mike Gordon, Brian Graham, Jeff Joyce, John Kershaw, Clive Pygott, Sreeranga Rajan, Debora Weber-Wulff and the members of the Hardware Verification Group at Cambridge, for their help and advice. This work has been funded by MoD research agreement AT2029/205.

References

[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] 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.

[4] 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 *Compiler Construction '92*, 1992.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

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

[14] Paul Curzon. Deriving correctness properties of compiled code. In L. Claesen and M. Gordon, editors, *Proceedings of the International Workshop on Higher Order Logic Theorem Proving and its Applications*. North-Holland, 1992.

[15] 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.

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

[17] 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.

[18] K. G. W. Goossens. Operational semantics based formal symbolic simulation. In L. Claesen and M. Gordon, editors, *Proceedings of the International Workshop on Higher Order Logic Theorem Proving and its Applications.* North-Holland, 1992.

[19] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-order Logic.* Cambridge University Press, 1992.

[20] 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.

[21] Kelly M. Hall and Phillip J. Windley. Simulating microprocessors from formal specifications. In L. Claesen and M. Gordon, editors, *Proceedings of the International Workshop on Higher Order Logic Theorem Proving and its Applications.* North-Holland, 1992.

[22] 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.

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

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

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

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

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

[32] Terry Pratchett. *Mort*. Transworld Publishers, 1987.

[33] Sreeranga Rajan. Executing HOL specifications: Towards an evaluation semantics for classical higher order logic. In L. Claesen and M. Gordon, editors, *Proceedings of the International Workshop on Higher Order Logic Theorem Proving and its Applications*. North-Holland, 1992.

[34] Sreeranga Rajan. Private communication, 1992.

[35] 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.

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

[37] 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.

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