

A Guide to VLISP, A Verified Programming Language Implementation

J. D. Guttman L. G. Monk J. D. Ramsdell
W. M. Farmer V. Swarup

The MITRE Corporation*
M92B091
September 1992

Abstract

The Verified Programming Language Implementation project has developed a formally verified implementation of the Scheme programming language, called VLISP. This report summarizes the results of the project. It also provides an overview of a group of reports presenting the details of the VLISP implementation and the logical proofs of its correctness.

Acknowledgment

This work was supported by Rome Laboratories of the United States Air Force under contract F19628-89-C-0001. We are grateful to Dr. Northrup Fowler III for his encouragement and advice.

We would like to express our gratitude to Professor Mitchell Wand of Northeastern University and to his student Dino Oliva. Through a subcontract arrangement, they developed and verified the PreScheme compiler that allows our system to run on real computers. They also contributed many ideas that helped us carry out other portions of the project. David Carlton, an undergraduate at Harvard University, also assisted us in the summer of 1990.

* Authors' address: The MITRE Corporation, 202 Burlington Road, Bedford MA, 01730-1420.

©1992 The MITRE Corporation. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the MITRE copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the MITRE Corporation.

Contents

1	Introduction	1
1.1	Goals of the VLISP Project	1
1.2	Accomplishments	3
2	Discussion	5
2.1	Algorithm-Level Verification	6
2.1.1	Representing Algorithms	7
2.1.2	Coverage of the Algorithmic Level	10
2.2	Prototype but Verify	11
2.3	Denotational and Operational Styles in Semantics	13
2.3.1	Denotational Semantics	13
2.3.2	Advantages of Operational Semantics	19
3	Structure of the Implementation	22
3.1	The Extended Compiler	22
3.2	The Interpreter Program	24
3.2.1	The Stored Program Machine	24
3.2.2	The Microcoded Stored Program Machine	26
3.2.3	The Garbage Collected, Microcoded Stored Program Machine	26
3.2.4	The Finite Stored Program Machine	27
3.3	The VLISP PreScheme Compiler	27
3.4	The VLISP Bootstrap Process	28
4	Structure of the Verification	33
4.1	The Extended Compiler	33
4.1.1	The Byte Code Compiler	34
4.1.2	The Tabulator	34
4.1.3	Faithfulness of the Operational Semantics	34
4.1.4	The Flattener	35
4.1.5	The Linker	36
4.2	The Virtual Machine	36
4.2.1	The Image Builder	36
4.2.2	The Microcoded Stored Program Machine	38
4.2.3	Garbage Collection	38
4.2.4	The Finite Stored Program Machine	38
4.3	The VLISP PreScheme Compiler	38

4.3.1	The VLISP PreScheme Front End	38
4.3.2	The Pure PreScheme Compiler	39
4.4	Unverified Aspects of VLISP	40
5	Conclusion	42
5.1	Directions For Future Work	42
5.1.1	Supplementary Work	42
5.1.2	Fully Compiled Implementation	44
5.1.3	Verifying Critical Portions of a Compiler	45
5.2	The Main Lessons	45

1 Introduction

This is a guide to the Verified Programming Language Implementation Project, which we usually refer to as VLISP (pronounced VEE-lisp). This guide summarizes its goals, accomplishments and conclusions, and it provides an overview of a set of detailed papers [10, 7, 11, 8, 31, 30, 24, 21, 22]. Those papers contain the main content of the verification. Their titles are listed in Table 1.

Citation	Title
[10]	The VLISP Byte Code Compiler
[7]	The Faithfulness of the VLISP Operational Semantics
[11]	The VLISP Flattener
[8]	The VLISP Linker
[31]	The VLISP Image Builder
[30]	The VLISP Virtual Machine
[24]	The VLISP PreScheme Front End
[21]	A Verified Compiler for Pure PreScheme
[22]	A Verified Runtime Structure for Pure PreScheme

Table 1: Detailed VLISP Reports

1.1 Goals of the VLISP Project

The primary goal of the VLISP project has been to produce a formally verified implementation of a programming language.

An implementation for a programming language consists of the ensemble of software required to make a program written in that language execute on a digital computer. A programming implementation may be a compiler, which translates programs in the given language to programs in a lower level language; or it may be an interpreter, which is a program that executes the higher level language directly; or it may be a combination of the two. The VLISP implementation is of this mixed type. It consists of:

- A simple compiler that translates programs in the source language to target programs in an intermediate level language;

- An interpreter (written in a different language) to execute the resulting target programs;
- A second compiler to translate the implementation language of the *interpreter* into assembly language code for commercial workstations (either a Sun 3 with an MC68020 CPU or a Sun 4 with a SPARC processor).

This project has formally verified most aspects of the language implementation. The formal verification consists in giving rigorous mathematical proofs of the correctness of the algorithms used. In general terms, an implementation is correct if, for any application program of the source language, and any input values, the behavior provided by the implementation matches the behavior predicted from the structure of the program by the definition of the language. Implementation correctness in this sense is susceptible to mathematical proof because programming languages can be given mathematically precise definitions; indeed, this is increasingly common practice.

We do not assert that VLISP is a complete verification because there are some aspects of the implementation that are not formally justified, although they have certainly been carefully scrutinized. They are summarized in Section 4.4. Although we think it is important to make the limits of our work clear, we think that it compares very favorably with other research in terms of rigor, clarity, and ambitiousness.

We have selected the Scheme programming language [12, 26] because it is a programming language in active use, with a concise and relatively well worked out formal semantics, and because there are a variety of good public domain implementations that could be used as blueprints.

Supplementary Goals We would also distinguish four supplementary goals:

- To examine different approaches to programming language semantics, and their appropriateness for verification on a substantial scale;
- To develop techniques for structuring a large verification;
- To examine issues involved in verifying memory allocation, garbage collection, and resource exhaustion [30];
- To examine the use of meaning-respecting transformations as a compilation technique [24]. In particular, these transformational techniques,

Component	Lines of Source
Scheme Standard Library (Unverified)	1600
Scheme to Byte Code Compiler	1600
Virtual Machine Interpreter	2200
VLISP PreScheme Front End	4200
Pure PreScheme Compiler	2300
Total	11900

Table 2: Approximate Sizes of Main Parts of the VLISP Implementation

followed by a simple syntax-directed compiler such as the Northeastern Pure PreScheme compiler [21, 22], seem to produce reasonably good assembly language code. Moreover, this combination seems much more straightforwardly verifiable than Kelsey’s transformational compiler [14], which gives the same language a succession of different semantical interpretations.

1.2 Accomplishments

We believe that VLISP has produced:

- The first comprehensive formal verification of any implementation for any programming language in actual use;
- The broadest implementation verification extant, in two senses:
 - It covers almost all the steps from a quite high-level programming language to the level of assembly language;
 - It starts from a highly abstract mathematical semantics and gradually refines it to a very concrete operational semantics.
- A formal verification for an implementation of a second language, namely VLISP PreScheme, a variant of Scheme suited for a variety of high assurance systems programming tasks.

Table 2 gives an estimate of the gross size of the different portions of the system. Moreover, VLISP is certainly the only verified programming language implementation there is that can be used to bootstrap itself. The possibility

of a bootstrap underlines the power of the Scheme source language, as well as the comprehensive character of the VLISP implementation.

Low Error Software The VLISP implementation effort encountered few errors even in relatively early prototype stages. Moreover, the errors that we did encounter were normally quite easy to find and correct. After all, our detailed specifications described what the algorithms should accomplish.

Indeed, our most serious problem, which was very hard to isolate, generally confirms the value of our approach. The problem concerned garbage collection. We had tried to make our implementation more efficient by a shortcut not reflected in the specification. We omitted the memory operations to ensure that each newly allocated memory location contained a distinctive value *empty*, imagining that the memory would be initialized to a useful value before anything could go wrong. In fact, the garbage collector could be called before the initialization. Hence the garbage collector would follow bit-patterns in the newly allocated memory that happened to look like virtual machine pointers. The example confirms the power of programming to rigorous specifications.

An Alternative: Verifying Critical Optimizations An alternative approach to the one we have pursued is not to verify a full implementation, but simply to verify the parts that are most suspect in it. For instance, the implementation for a programming language may do interesting transformations and optimizations before generating a C program; the compiler back end is then simply a C compiler. In the case of an implementation of this type, the main benefit of formal methods may be achieved if the original pre-C transformations are verified. The back end may be considered sufficiently reliable on empirical grounds. Some approaches to parallel programming would be good candidates for this sort of treatment, as they involve sophisticated transformations to introduce parallel tasks or to optimize access to objects. We expect that the techniques that we have developed under VLISP will be highly applicable in this context also.

2 Discussion

One crucial condition for our success was our choice of Scheme as our target language. There were three main points that made Scheme a natural choice:

- The clean and generally accepted formal semantic definition of Scheme;
- The simplicity and uniformity of Scheme's syntax;
- Good public domain implementations, which allowed us to focus our attention on verification issues, rather than on clever implementation strategies; moreover, there was prior work on formal verification of a crucial portion of a Scheme implementation [5].

Each of these conditions suggests some lessons for further work in implementation verification.

First, we consider it well worth the trouble to equip programming languages with formal semantics. A usable formal semantics must be rigorous, but still humanly understandable and mathematically tractable to reason and calculate with. To provide such a semantics for a language requires a substantial amount of labor by highly skilled experts. Nevertheless, there are several important benefits of insisting on a good programming language formal semantics.

- It constrains language design, encouraging integrated languages organized around well-understood constructs;
- It provides a deep model of program behavior, that can be used either by an applications programmer, by the designer of a careful software engineering method, or by the designer of a formal verification method for the language;
- It provides a rigorous guide to the language implementer, useful not only for formal verification of the implementation, but also for informal confidence in correctness for implementations that will not be rigorously verified.

Second, although we are aware that the uniform, Lisp-style syntax of Scheme does not appeal to some programmers, we nevertheless recommend that the verification process start from a simple syntax. If the language has a complex *concrete* syntax, this means that the implementation verification should focus its attention on an *abstract* syntax as its starting point. This

abstract syntax would normally be the tree-like output of a parser. We make this recommendation because the heart of implementation correctness is semantic: Does the output code *do* what the source program demands? The problem of correct parsing by a front end is separable, and ancillary to this main problem. Moreover, there is a very well-understood theory and practice of parsing. If there are substantial questions about the correctness of a particular parser, they can be answered using these familiar techniques.

Third, as we will argue in more detail below (see Section 2.2), verification is most effective when combined with direct practical experience of the algorithms and their representation in code. It is even desirable if a full prior implementation can be examined as a “blueprint” before the verification is begun. Naturally, the eventual verified implementation will be different in many ways. But the insight into the overall structure (and detail) of a full, working prototype is invaluable.

2.1 Algorithm-Level Verification

There are several different levels at which formal methods can be applied to software. Ranged by their rough intuitive distance from the final executable image, among them are:

- Formalization of system correctness requirements;
- Verification of a high-level design specification against formalized correctness requirements;
- Verification of algorithms and data types;
- Verification of concrete program text;
- Verification of the (compiled) object code itself, as illustrated by Boyer and Yu’s work [3].

Broadly speaking, the higher up a particular verification is in this list, the more there is that can go wrong between the verification process and the actual behavior of the program on a particular computer. On the other hand, the mass of detail increases, making the analysis progressively harder. Indeed, there is also a tradeoff in the informativeness and even reliability of the verification: as the amount of detail rises, and the proportion of it that is intuitively understandable goes down, we are less likely to verify exactly the properties we care about, and more likely to introduce mathematical flaws into our proofs.

We have focused on the algorithm level. We will argue that it provides a detailed view of the main potential sources of errors. Moreover, we have found that the direct verification of the concrete program text itself would be much less tractable. Thus it seems to us the preferred level to work, given that our goal is to get rigorous insight into the correctness of a software system of very substantial size.

2.1.1 Representing Algorithms

Two main questions, if algorithm-level verification is to be used, are how to represent algorithms, and how to ensure informally that the concrete program text is faithful to the algorithm as presented. In different portions of the VLISP implementation, we have used three different main approaches to these problems.

The one we have preferred, when it is usable, is to represent the algorithm by the program text itself. In particular, we use an applicative Scheme program¹ directly as a representation of an algorithm. It is very easy to reason rigorously about these programs, which use conditionals and recursion primarily. Data manipulation primitives such as `cons` are interpreted as operating on mathematical objects such as pairs or sequences (partial functions defined for an initial segment of the natural numbers). This approach has been used in the Extended Byte Code Compiler, including the Byte Code Compiler proper and the Tabulator [10], the Flattener [11], and the Linker [8].²

Nevertheless, we do not consider this to be program-level verification. The reason is straightforward. Although the algorithm representation is identical with the program representation, the *interpretation* of the two is different. In particular, the semantics of the Scheme program uses a store, and it uses the continuation-passing style [29]. The former is needed to model the changes in values for variables and data structures as execution of a Scheme program proceeds. The latter is used to represent the transfer of control when an execution error occurs, or when the `call/cc` procedure is used to turn the continuation into an escape procedure object. However,

¹By an “applicative” program, we mean one that makes no use of state-changing operations such as assigning new values to variables or mutating the values in data structures. We would contrast an applicative program with an “imperative” one.

²It was not used in exactly this form in the Image Builder [31], which needs to use mutable vectors for reasonable performance. In the Image Builder, the code is used directly as the representation of the algorithm, but one must be a little more careful reasoning about the algorithm, as one must also keep track of the current (imperative) state.

neither of these characteristics is needed to reason effectively about the algorithms in question, for the purposes we have in hand. For one thing, we know that these particular algorithms are applicative, so that a store semantics is not needed for them. We also know that these programs do not use `call/cc` except as a top-level escape to signal an error to the user. Finally, we need not worry about these error escapes, as our main theorems are “partial correctness” assertions. They have the form:

If e' is the result of running C on the input expression e , then e'
and e agree in their meaning.

Naturally, if C raises an error when run on e , there is no result e' , so that the antecedent of the conditional is always false, and the assertion itself is vacuously true. Thus, given what we know about the algorithms and what we want to prove about them, we are justified in using a much simpler semantics than their status as arbitrary Scheme programs would justify.

In the specification for the VLISP virtual machine [30], which is intrinsically imperative, we use a different method. We identify a set of state-observing primitives and a set of state-modifying primitives. Each of the state-observing primitives extracts a component value out of the (complex) state of the virtual machine, while each of the state-modifying primitives makes an atomic alteration to the state. The formal specification makes explicit all the altered components of the state. Next to it is presented the corresponding PreScheme code implementing the modification, often a single call to a data manipulation primitive of PreScheme. These primitive state-observers and state-modifiers are composed to accomplish the atomic actions at the level of the byte code interpreter state machine. The sequences of observers and modifiers are again presented juxtaposed with the concrete code that can be seen to carry out the same operations.

One complication is due to the fact that VLISP PreScheme, the implementation language for the virtual machine, compiles to a language in which all procedure calls are in tail recursive position. The Front End must substitute the code for a called procedure in line wherever there is a call in the source that is not tail recursive. There are several procedures in the virtual machine that cause memory to be allocated, and each of them, when coded in the obvious way, contains a call to the allocation routine, which in turn calls the garbage collector if the allocation would exceed a bound. These calls are not tail recursive, as the original procedure must normally do some initializations within the new memory object after it has been allocated. Rather than have the code for the garbage collector duplicated in

Original version:

```
(define (caller)
  (callee arg)
  (command1) ... (commandn))

(define (callee arg) ...)
```

Continuation Passing Version:

```
(define (new-caller)
  (new-callee
   arg
   (lambda () (command1) ... (commandn))))

(define (new-callee arg continuation)
  ...
  (continuation))
```

Figure 1: Schematic Example of Continuation Passing Style

line at each of the original call sites, we have programmed the calls to the allocation procedure and garbage collector in a partial “continuation passing style.”

Continuation passing style is a way to replace a call not in tail recursive position with two calls that are in tail recursive position. The callee is rewritten to accept an additional parameter. This “continuation parameter” is assumed to be a procedure. The body of the callee is rewritten to invoke the continuation parameter (tail recursively) after it has done its previous work. The caller is also rewritten, so as to make the call with a suitable procedure as an additional actual parameter. The actual parameter value will carry out the actions that would have been done after the return of the callee. A schematic example is given in Figure 1. We needed only to rewrite a few procedure calls using this method. This is to be distinguished from a full continuation passing transformation, which would apply similar ideas to every call not in tail position. Some compilers (among them Rabbit [28], Orbit [16], and the SML/NJ compiler [1]) use this approach to make the control flow through their source code explicit. Full continuation passing

style conversion is somewhat tricky, and in any case quite different from our adoption of a particular coding style in a few important places.

Finally, in the proofs about the VLISP PreScheme Front End [24], which translates the implementation language of the virtual machine into a subset called Pure PreScheme, we use yet a different approach. The Front End is transformational in nature. It applies a set of meaning-preserving transformation rules repeatedly, starting with the source program, and terminating only if either it obtains a Pure PreScheme program or it observes that no legal, type-correct Pure PreScheme program can be reached. In some cases the Front End would run forever. Some rules are applicable only in certain conditions, for instance when a subexpression returns a value independent of the state in which it is executed, or when it is guaranteed not to modify the state while it executes. These conditions suffice to ensure that application of the rule will leave the meaning of the program unchanged.³ However, the Front End applies the rules in a still more restricted way, gathering different rules into different passes, and not applying certain rules when it is (heuristically) undesirable to do so, for instance because the target code would become too large.

The proof of correctness abstracts completely from this level of the algorithm. The proof focuses on establishing that each rule improves the meaning of the program it is operating on in the sense given in the footnote. Hence, any algorithm is correct if it changes the program only according to the rules, and respects the conditions of applicability, no matter what control structure the algorithm may impose. This portion of the VLISP verification tolerates the greatest gap between the verification and the concrete code.

2.1.2 Coverage of the Algorithmic Level

We believe that the approaches we have described, particularly the first two mentioned above, provide strong protection against realistic errors.

³Or to be more precise, certain rules “improve” the meaning of the source program, in the sense that they may replace a bottom answer \perp by a non-bottom answer. When the given program references a global variable location before it has been initialized, the semantics may predict that its answer is \perp . By contrast, the transformation may eliminate this global variable reference, and thus the resulting program may return a useful computational answer. However, when the semantics predict that source program will return a *non-bottom* answer, then the transformed program will return the same answer.

- **The Wrong Algorithm** Even an experienced programmer can have the wrong idea about what her program is intended to accomplish, or about whether a particular algorithm will accomplish that goal. By proving that algorithms meet their specifications, and that they jointly provide a high level property (faithfulness to a language semantics), we provide direct protection against this error.
- **Incorrect Assumptions about Inputs** To take an example, in proving the correctness of the VLISP flattener [11], we found we could prove our conjecture only on the assumption that the input had a particular property (“respecting its template table”, Definition 6). We then went back to the proof of the tabulator [10] (which creates inputs to the flattener) with a new conjecture to prove, namely that the output of the tabulator respects its template table (Theorem 36).
- **Unreliable Data Type Implementations** At many points in our work, we have given brief informal arguments that the procedures implementing a data type are faithful to the mathematical functions used in a specification (see, e.g., [10, Section 5.2], or [11, Section 5.2]). In doing so, we naturally had to examine the algorithms that those procedures embody. However, further verification work (or re-implementation) can then rely on the mathematical description, rather than on the details of the code.

In some cases we omitted this step for familiar data structures. In particular, the simple hash tables used in the linker [8] and the symbol table used in the image builder [31] were not justified in this way.

- **Unreliable Programming Language Implementation** Perhaps a programmer gets her source code right, but the language implementation causes it to execute incorrectly: How can we protect against that? In our case, because all our programs are written either in Scheme or in PreScheme, and we have provided a full implementation of both languages, our bootstrap process provides substantial (although logically incomplete) protection against unexpected errors in the language implementation (see Section 3.4).

2.2 Prototype but Verify

Some partisans of formal methods seem to regard formal program development as a rather patrician activity. The designer receives some formal re-

quirements. She proceeds to develop a program, in successive stages, using techniques guaranteed to refine the set of possible behaviors. In this process, the data structures to be used are selected. Some information about asymptotic complexity may be available, and some refinement steps may be motivated by the desire to reduce complexity. When all the intellectual work has been done, the now fully refined design may be passed along to a sweaty, unwashed programmer. Or perhaps the programmer can be eliminated, by passing the result of the refinement straight to the compiler.

We find the view caricatured here implausible, not to say unappealing. In fact, we think that our ability to carry out this verification depended on our using a contrasting approach.

At the start of our project, we selected an existing, fully functional Scheme implementation, namely Kelsey and Rees's Scheme48 [13], to serve as a model. We then proceeded to develop a succession of prototypes that recast it into a form closer to the one we wanted to verify. At one stage we constructed a complete prototype of the byte code compiler and virtual machine within Standard ML [19, 18]. This was particularly beneficial for two reasons. First, the great majority of the bugs in the prototype were type conflicts. The ML type checker is a highly efficient way of tracking these down. Second, the Standard ML module system was an almost ideal tool for isolating refinement steps. The ML code for a layer in the refinement of the virtual machine (other than the last) was presented as a functor. Its signature argument contained the primitives at that level. When the functor was instantiated, the procedures at the next refinement level were supplied to serve as their meanings. Thus the ML prototype also served as a refinement-oriented specification.

We gradually replaced the Scheme48 byte code compiler with a whole succession of carefully separated algorithms [10, 11, 8, 31]. This process insured that we gained a good intuitive sense of the subtle points of different versions of the program. In addition, we came to appreciate where logically separate functional requirements complicated the job of producing a transparently correct program. We repeatedly added new phases to the byte code compiler in order to produce separate procedures with simpler correctness conditions.

A variant process governed the development of the VLISP PreScheme Front End [24]. In this case, there was no preexisting implementation available and the design of the language was fluid. Two prototypes were constructed before the final version. The prototypes were used to explore the use of various rule sets, control procedures, and data representations. There

was a self-imposed constraint that the control procedures be simple and straightforward. The ability to select new sets of rules with full knowledge of their impact on the control procedures was extremely useful.

The first prototype demonstrated a naïve implementation of a rule set to be too slow. Although the next prototype compiled a preliminary version of the VLISP VM, the attempt at proving the correctness of its rules exposed several errors in them. This process illustrates the reciprocal interaction between prototyping, which helps to show what might be practically useful, and verification, which can establish what will be reliable.

2.3 Denotational and Operational Styles in Semantics

One of our goals had been to examine the situations in which the denotational and operational styles in semantics are useful and effective, and the difficulties of interrelating the two styles in an effort that we foresaw would require both.

2.3.1 Denotational Semantics

The official Scheme semantics [12, Appendix] is denotational in character. This means that the meaning of any expression in the Scheme abstract syntax is given as a mathematical value, the denotation of a compound expression being determined by combining the denotations of its constituent subexpressions. This approach, which requires some sophisticated mathematics (summarized in [29], for instance), has been one of the dominant frameworks for giving meaning formally to programming languages (see also [27]).

Advantages of Denotational Semantics The main general advantages cited for denotational semantics are:

- Its mathematical precision, and its application of traditional mathematical methods to reason about the denotations of expressions;
- Its independence of a particular execution model;
- Its neutrality with respect to different implementation strategies;
- The usefulness of induction on the syntax of expressions to prove assertions about their denotations, and of fixed point induction to prove assertions about particular values.

These advantages are genuine. As a consequence, we believe that denotational semantics is a good, effective tool for reasoning about the main compilation steps. These are the transformations that introduce the essential execution model, rather than simply refining a data representation, or decomposing a computational transition into a simple sequence of substeps. In particular, the denotational approach was indispensable in the proof of the byte code compiler [10] and the VLISP PreScheme Front End [24].

These large transformations, which embody a procedural analysis of the source code, seem to require the freedom of the denotational semantics. We would consider it a very difficult challenge to verify this sort of transformation using the operational style of the Piton compiler proof [20].

Objections to the Scheme Denotational Semantics We have however a number of objections to the way that the denotational semantics of Scheme has been framed.

Memory exhaustion We have in fact allowed ourselves to modify the official semantics somewhat [10, Section 2.1], in ways that struck us as inessential to its main content. The official semantics always tests whether the store is out of memory (fully allocated) before it allocates a new location. We have removed these tests and introduced the assumption that the store is infinite. We did so for two main reasons:

- Any implementation must use its memory for various purposes that are not visible in the official denotational semantics. Thus, the denotational semantics cannot give a reliable prediction about when an implementation will run out of memory.
- It simplifies reasoning to treat all possible memory exhaustion problems uniformly at a low level in the refinement process.

For instance, all Scheme implementations must use memory to record the return point for procedure calls that are not tail-recursive, and every implementation can exhaust memory in this way. However, the Scheme semantics for procedure call predicts that memory cannot be exhausted by nested procedure calls.

To take another example, many Scheme implementations have a relatively small bound on the number of arguments that can be given in any particular procedure call (it is 256 in the case of VLISP), and all implementations can in principle run out of stack space for accumulating the arguments

to a call. But the official semantics predicts that accumulating arguments to a procedure call cannot exhaust memory. We are not arguing that these forms of memory exhaustion are errors that should be faithfully reflected in the semantics. On the contrary, we believe that the semantics should be framed uniformly, at a level of abstraction at which no type of memory exhaustion is represented.

In VLISP, we have chosen to specify the finiteness of memory only at the very lowest level in the refinement of the virtual machine [30, Chapter 4]. At this level all sources of memory exhaustion are finally visible. Moreover, many proofs at earlier stages were made more tractable by abstracting from the question of memory exhaustion. For instance, in a state machine refinement proof, it would frequently be difficult to ensure that the more refined machine would run out of memory in precisely the same conditions as the less refined machine. Nor would it be informative. Although a programmer may know that one program is better than another program because it is less apt to run out of memory, it is very rarely a part of the intended functionality of a program to run out of memory in particular circumstances but not others. Certainly Scheme would not be the most suitable programming language for any such program.

Semantics of Constants The official Scheme semantics contains a semantic function \mathcal{K} which maps constants—expressions in the abstract syntax—to denotational values. But \mathcal{K} is simply characterized as “intentionally omitted.” In some cases, its behavior seems straightforward: for instance, its behavior for numerals. However, it is far from clear how one would define \mathcal{K} for constants that require storage.

Semantics of Primitives Although the official Scheme semantics contains some auxiliary functions with suggestive names such as *cons*, *add*, and so on, it gives no explicit account of the meaning of identifiers such as `cons` or `+` in the standard initial environment. Apparently the framers of the semantics had it in mind that the initial store σ might contain procedure values created from the auxiliary functions such as *cons*, while programs would be evaluated with respect to an environment ρ that maps identifiers such as `cons` to the locations storing these values.

However, this presupposes a particular, implementation-dependent model of how a Scheme program starts up. The idea is that it should start in a store that already contains many interesting expressed values.

But other approaches are also possible. For instance, for our purposes it was more comprehensible to have the program start up with a relatively bare store. In the VLISP implementation, the byte code program itself is responsible for stocking the store with standard procedures. These standard procedures are represented by short pieces of byte code that contain special instructions to invoke the data manipulation primitives of the virtual machine. The initialization code to stock the store with these primitives makes up a standard prelude: The byte code compiler prepends the initialization code to the application code generated from the user's program.

For this reason it would have been particularly difficult to show that the VLISP implementation's treatment of primitives is compatible with that of the denotational semantics. For even had we filled in the sketch of a semantics of primitives provided by the framers, we would still have had a conflict between their "initial store" approach and our "initialization code" approach. It would have been necessary to show that the VLISP initialization code invoked its continuation with a store matching the initial store of the filled-in semantics. This would have been awkward mathematically, and probably unenlightening to boot.

Normality conditions Consider a Scheme program fragment using a global variable x :

```
(let ((y (cons 1 2)))
  (set! x 4)
  (car y))
```

We have a right to expect this to return 1. That is, we have a right to expect that the storage allocated for a pair will be disjoint from the storage allocated for a global variable. However, nothing in the official semantics ensures that this will be the case. Indeed, the association between identifiers, such as x , and the locations in which their contents are kept, is established by an "environment" ρ . But ρ is simply a free variable of the appropriate type. Thus, the semantics makes no distinction between the environments that respect data structures in the heap and those that do not.

Similar problems concern overlap among vectors, pairs, and strings, and overlap between storage used by mutable and immutable objects of the same kind. In proving the faithfulness of the operational semantics [7], we needed to introduce a comprehensive list of these "normality conditions." Unfortunately, the denotational semantics does not seem to have a natural way to

include these conditions as a “top-level” part of the semantics of the programming language itself. Nevertheless, an implementation that violated them would be considered wrong by any programmer. Peter Lee [17] makes a related objection (among others) to denotational semantics as it is normally practiced.

Tags on Procedure Values A procedure object is treated in the semantics as a pair, consisting of a store location and a functional value. The latter represents the behavior of the procedure, taking the sequence of actual parameters, an expression continuation, and a store as its arguments, and returning a computational answer. The location is used as a tag, in order to decide whether two procedure objects are equivalent in the sense of the Scheme standard procedure `eqv?`. The exact tag associated with a procedure value depends on the exact order of events when it was created. Similarly, the locations of other objects will depend on whether a location was previously allocated to serve as the tag for a procedure object.

By contrast, the semantics for VLISP PreScheme does not tag procedure values [24]. As a consequence, many transformations can be proved correct for VLISP PreScheme, but the corresponding arguments for Scheme would not be sound. On the other hand, in PreScheme, as in ML, procedures cannot be tested for equality. We prefer this tradeoff: It seems unfortunate to sacrifice the possibility of doing a wide variety of verifiable optimizations just to have a test for equality on procedures. The more so as the test in Scheme is fairly unpredictable [26, page 13].

Artificial Signature for Expression Continuations The semantics specifies the type for expression continuations as $E^* \rightarrow C$, which means that evaluating a Scheme expression may (in theory) contribute a finite sequence of “return values,” as well as a store, to determining what answer the continuation will return.

In fact, every expression in IEEE standard Scheme that invokes its continuation at all uses a sequence of length 1. This suggests that, in some intuitive sense, an implementation for Scheme, conforming to the IEEE standard, need not make provision for procedures returning other than a single value.

Although it would be sufficient for the language, as currently defined, to require just a single return value, the semantics, we are told, was written in this more complicated way partly so that it would not need modification

Variable	Value
ρ	$\mathbf{F} \mapsto \ell_1$
σ	$\ell_1 \mapsto \phi_1; \ell_2 \mapsto \text{unspecified}$
κ	$\lambda\epsilon^* \sigma . \#\epsilon^*$
ϕ_1	$\langle \ell_2, \lambda\epsilon^* \kappa \sigma . \kappa \langle 1, 2, 3 \rangle \rangle$

Table 3: Naïve Counterexample to Single-Valued Implementations

when Scheme adds a mechanism for procedures that return multiple values, as Common Lisp programs may do using the `(values ...)` form and **T** programs [25] may do using the `(return ...)` form. It is in fact expected that the next (fifth) revision of the *Report on the Algorithmic Language Scheme* will incorporate a mechanism like **T**'s.

However, as a consequence, in the most literal sense, an implementation is unfaithful to the formal semantics as written if it makes no provision for multiple-value returners.

We can make this point clear with an example. Consider the program (**F**), which calls a procedure with no arguments. Which procedure value gets called depends on which procedure value is stored at the location that the environment associates with the identifier **F**. The Scheme semantics determines a computational answer when an environment ρ , an initial store σ , and an initial continuation κ are supplied. Given the values described in Table 3, we can easily see that the semantics predict that the correct computational answer is 3.

However, saying that an implementation makes no provision for multiple-value returners is in effect to say that if κ is implemented at all, then it always results in a computational answer of 1. Clearly, in order to explain the sense in which a single-valued implementation is “good enough,” we must formalize what is obviously inappropriate about the σ and κ shown, as potential initial arguments. To justify a single-valued implementation, we must demonstrate that the implementation will compute the answer predicted by the semantics, assuming that the initial arguments are not inappropriate in whatever sense is formalized.

To our knowledge, there has been no previous formal work that justifies an implementation in making this decision. We think that it was probably ill-advised to introduce the idea of procedures returning multiple-values into

the semantics without having developed some theory to explain its compatibility. In [10], we have developed a theory that identifies an appropriate criterion of correctness, and adapts a Clinger-style proof of the Byte Code compiler to this criterion of correctness.

2.3.2 Advantages of Operational Semantics

By an operational semantics we mean a description of a state machine of some flavor, with identifiable atomic computational steps. While an operational semantics lacks the abstraction and implementation-neutrality of a denotational presentation, it is highly effective for a variety of purposes. In particular, although induction on abstract syntax and fixed point induction do not play a major role, nevertheless operational semantics offers a separate, frequently useful sort of induction. This is induction on sequences of computational steps. We found operational semantics to be more appropriate at many stages in the VLISP proofs.

When only Operational Semantics is applicable In some cases, denotational semantics do not really make sense. For instance, consider a language containing a jump instruction taking its offset as a numerical argument. The effect of executing the instruction is to advance the instruction counter by the number of bytes specified as the offset. It seems pointless to give such a language a denotational semantics. The cardinal goal of denotational semantics is to give the phrases of a programming language a compositional semantics: that is, the meaning of a compound expression is to be a function of the meanings of its immediate subexpressions. In this kind of a language, by contrast, the meaning of a program depends not just on the meanings of the subexpressions, but also on their widths in the sense of the number of bytes they occupy in the instruction stream. If we want to analyze the correctness of particular numerical offsets in programs, then we are better off to formulate the semantics operationally. In fact this was our motivation for switching from denotational semantics to operational semantics with the Tabular Byte Code TBC. We needed to make the transition before we could justify the flattener algorithm [11].

Primitives, Particularly for I/O Sometimes it is more convenient to use an operational style to formulate particular primitives for manipulating data. Although in some cases this is no more than a matter of convenience, input and output operations are in fact conceptually easier to understand in

an operational framework than in the denotational manner of the Scheme semantics.

I/O is different from many other aspects of program meaning because the order of events is an intrinsic part of its intuitive significance. In the denotational model of Scheme, two programs have the same meaning if they deliver the same final computational answer (or lack thereof) when applied to the same initial arguments. But this seems far too crude a criterion of meaning when I/O behavior is considered. We will examine two aspects of the issue.

First, by way of example, consider two versions of a “repeater” program: each takes a sequence of characters as input, and outputs the same sequence of characters. However, in one program, the length of the output sequence is never more than a fixed number of characters shorter than the number of characters it has consumed. In the other program, until the input sequence is empty, it is never more than a fixed number of characters away from one tenth of the number of characters it has consumed. These programs are not interchangeable for practical purposes. Moreover, it seems that any acceptable semantic account of I/O ought to be able to distinguish between these two patterns of behavior. Yet a semantics organized around “final computational answer” may be hard pressed to formalize the distinction.

Second, a semantics of the “final computational answer” is also hard pressed to distinguish among various non-terminating programs. A program that does I/O delivers information to its environment about its computational behavior in spite of not terminating. Thus, a non-terminating program may be more useful if it consumes plaintext characters and key material and outputs encrypted characters, than if it outputs a random sequence of characters. It is not clear how to make these distinctions in a denotational framework.

On the contrary, it is perfectly plain how to do so in an operational setting. For this reason, we do not consider the Scheme denotational semantics suitable for framing an all-encompassing definition of the adequacy of a Scheme implementation. Some aspects of adequacy are more natural to express at a lower level of abstraction than others.

Garbage Collection Implementations of Scheme and other Lisp-like languages have garbage collector algorithms. Garbage collectors are crucial to the effectiveness of these languages: an implementation without a garbage collector would not be acceptable for most any normal purpose.

A garbage collector recycles previously used storage locations when their contents can no longer influence the rest of the computation. It may also re-locate objects that are still in use, taking care also to update pointers to the objects. If we regard memory as a directed graph, where the pointers form the arcs, then a garbage collector is required to transform the portion of memory reachable from registers, regarded as a graph, to an isomorphic graph.

Given that a garbage collector is required in a Scheme implementation, one might expect that the denotational semantics would entail some theorem justifying garbage collection as an implementation strategy. As far as we know, there is no such theorem to be had. The problem is that the denotational semantics uses very rich domains. The domain $\mathbf{C} = \mathbf{S} \rightarrow \mathbf{A}$ of command continuations, for instance, whose members deliver a computational answer when supplied with a store, contains all continuous functions of this type. Thus, in addition to functions sensitive only to the structure of the store as a graph, it contains also functions sensitive to the actual locations that store different objects. Thus, some members of \mathbf{C} are not invariant under garbage collection. Garbage collection is justified only because these command continuations are not actually represented in implementations.

It might be possible to work around this problem. Subsets of the denotational domains may exist that are small enough to contain only objects that are invariant under garbage collection, but large enough to be closed under the operations used in the semantics. We have not, however, developed a theory of this kind.

In order to prove that garbage collection is a legitimate implementation strategy, we have deferred the proof to a lower level with an operational semantics. In the more restricted world of its state machine, we can indeed justify garbage collection. Let us say that two states are *similar* if their reachable portions are isomorphic as graphs. Then we can prove that each state machine transition preserves similarity. Moreover, for terminal states, if the states are similar, then the computational answers they deliver are equal. From this it follows that an implementation with garbage collection is justified if the garbage collector always returns a state as its result that is similar to the argument it was invoked on. This proof is carried out in [30].

3 Structure of the Implementation

Following several previous implementations [13], and some earlier verification efforts such as Clinger's [5], the VLISP implementation has three main parts. The first is a compiler stage, which translates Scheme source programs to an intermediate level language based on more complex instructions than machine code. The second stage is an interpreter, which is a program capable of executing the complex instructions of the intermediate level language. The intermediate level language is referred to as "byte code," as it is common to pack each instruction into a single byte. The interpreter is written in a language called VLISP PreScheme, which may be considered as a highly restricted sublanguage of Scheme. It is derived from a language invented by Richard Kelsey and Jonathan Rees [15, 13]. The third portion of our implementation consists of a compiler from VLISP PreScheme to assembly language. We frequently refer to the interpreter program as the VLISP Virtual Machine or VM.

There is a simple division of labor between the compiler stage and the interpreter stage. The compiler is responsible for analyzing the flow of control in a Scheme source program into simple instructions like those of an assembly language. The primitive data structures of the Scheme language are not reduced by the compiler stage of the implementation, however, and therefore memory allocation and reclamation cannot be handled, either. Instead, the interpreter is the stage responsible for supporting Scheme data structures, and managing the memory at the level of concrete, numeric addresses.

The compiler stage is responsible for transforming a given Scheme application program into a program in the intermediate language. The resulting program is written to a file in a binary representation; we call a file containing a binary representation of this kind an "image." To run the VLISP byte code interpreter on an image, we specify the filename of an image as a Unix command line parameter. The resulting Unix process may take input and produce output as usual, and, if it terminates, it returns a Unix exit code as its final computational answer.

3.1 The Extended Compiler

The extended compiler acts on source programs in the Scheme language, and ultimately produces programs in a linearized byte code that we call "stored byte code" [30]. Its input programs consist of abstract syntax trees in a format only slightly different from the format used in the Scheme Stan-

dard [12]. The stored byte code output is in a binary form suitable for execution in the VM.

The extended compiler consists of a sequence of procedures which rewrite the original source program in a succession of intermediate languages.

- The **byte code compiler** [10] (properly so called, as distinguished from the extended compiler, of which it forms the first stage) rewrites the Scheme source code into a tree-structured byte code based on Clinger's target language [5]. We will refer to this tree-structured byte code as the **basic byte code** or BBC. The BBC makes the flow of control in the Scheme program explicit, including procedure call and return, the order of argument evaluation, and the operations involved in building environment structures for local variables.
- The **tabulator** [10] transforms BBC procedures into templates in a **tabular byte code** or TBC. A template consists of some code, together with a table listing global variables, constants, and embedded templates (representing locally defined procedures). Instructions using these objects, such as the instructions to reference or modify the value of a global variable, are modified by the tabulator to use an index into this template table. The TBC is still tree-structured in that the code sections executed on the two branches of a conditional, or the code representing the target to return to after a procedure call, occurs nested within the instruction that governs it.
- The **flattener** [11] transforms procedures in the TBC into a form that is linear rather than tree-structured. Conditionals are represented in a familiar way using `jump` and `jump-if-false` instructions that involve a numeric offset to the target code. Similarly, the target code to which a procedure should return is indicated by a numeric offset from the instruction. We call the output language of the linearizer the **flattened byte code** (FBC).
- The next stage of the extended compiler is the **linker** [8]. Its job is to transform a collection of templates in the FBC into a single linear structure representing the full program. Its output language is referred to as the **linked byte code** (LBC). It is needed because an FBC program is given in a tree-structured form in which each template contains subordinate templates for the subprocedures that are local to it. In addition, constants referred to in the FBC code may

be tree-structured, such as lists of lists. The linked byte code output by the linker has a linear sequence of constants and a linear sequence of templates. These are arranged so that whenever one constant is a subexpression of another—for instance, an element of some list—then the subexpression occurs earlier. Similarly, when one template references another one, the referenced template occurs earlier. As a consequence, the VLISP image builder program can create the structures sequentially; whenever a reference is needed, it can output a pointer to the address of a previously constructed object.

- Finally the **image builder** [31] writes out the image to a file. The image builder is responsible for transforming the syntax of the linked code into a succession of bytes. However, the linked code program also has a variety of internal references within it, so that the image builder must compute the widths of the representations of successive objects. The image builder then converts the linked code cross references into 32-bit pointers embedded in the resulting image file.

3.2 The Interpreter Program

The easiest way to describe the underlying structure of the VM interpreter is to describe a succession of progressively less abstract views of its operation. In each of these “views,” the VM will be regarded as a state machine. These state machines comprise:

- The stored program machine SPM;
- The micro-coded stored program machine MSPM;
- The garbage-collected, micro-coded stored program machine GSPM;
- The finite garbage-collected micro-coded stored program machine FSPM.

They are described in detail in [30]; we will describe them in intuitive terms here.

3.2.1 The Stored Program Machine

The most abstract view of the interpreter is as a state machine whose state consists of:

- A *store*, which is used to hold:

- The *program* that the machine is executing, as a read-only data structure.
 - The values of *global and local variables*, which change as execution proceeds.
 - *Data structures* such as pairs (“cons cells”) and vectors, the components of which may be side-effected as execution proceeds.
- A collection of *ports*, divided into input ports and output ports. The state machine has operations to read a character from an input port and write a character to an output port.
 - A set of *registers*, consisting of:
 - A *template* register. Its value is an object containing a sequence of byte code instructions and a table used to locate constants, global variables, and other templates. A template together with an environment represents a procedure value.
 - An *offset* register. It contains a number, used like a program counter to indicate the position of the current instruction within the byte code instruction sequence of the current template.
 - A *value* register. It contains the representation of a Scheme object, and serves as a sort of accumulator.
 - An *argument stack*. It contains a stack of Scheme values, that are being collected as the arguments to a procedure call.
 - An *environment* register. It associates locations with the local variables of the procedure executing.
 - A *continuation* register. It contains a stack-like object that represents the future of the computation, to be invoked when the procedure currently executing returns. The continuation register is similar to the control stack in many implementations.

Each primitive instruction causes the values of several of these state components to change. For instance, the `global` instruction causes the contents of the store location associated with a global variable to be loaded into the value register. It also causes the offset register to be incremented by 2, to pass the current op-code, and its one operand.

In this machine, we allow the store and various numerical objects to be unbounded. That is, addresses are not limited to those that can be represented in a word of fixed width. Similarly, the numbers that represent

offsets are not bounded. Naturally, the actual implementation can only correctly emulate the behavior of the SPM on a particular class of computations, namely those in which no quantities arise that are bigger than certain bounds.

3.2.2 The Microcoded Stored Program Machine

The MSPM operates on a state of the same kind as the SPM. The difference is that its transitions are defined in terms of primitives called *state observers* and *state transformers*. Each one of the state observers is a function that returns as its value some simple value in the state, such as the value in a particular register, or the value at the store location specified in an address (given as an argument to the function).

The state transformers alter the value of a single state component, or allocate contiguous locations in the store for a new data object.

3.2.3 The Garbage Collected, Microcoded Stored Program Machine

The state of the MSPM may be regarded as a graph in which addresses (pointers) are arcs, and the registers and data objects in the store are nodes. The portion of this graph that can be reached from the registers will be called the “rooted portion of the graph.” Let us say that two states s_1, s_2 are *similar* if the rooted portion of the graph for s_2 results from the rooted portion of the graph for s_1 when we alter the numerical values of the addresses, according to some one-to-one scheme.

The observers and transformers have the property that they produce similar results when applied in similar states. This entails that the MSPM can be implemented by a machine in which the store is garbage collected, and in which objects in the rooted portion are relocated periodically so that storage can be reused. So also, therefore, may any machine which, like the SPM, can be built by sequencing these observers and transformers.

The garbage collected GSPM has the same set of observers and transformers as the MSPM. It differs only in that it has two stores, only one of which is active at any time. Periodically a garbage collection may occur, in which rooted objects are shifted around using the other store. Unrooted objects may disappear. Since, at this level of description, each store is unbounded in size, there is no reason (understandable in these terms) why garbage collection should take place at any particular time. We conceive of

it as happening whenever a not-further-specified condition becomes true.

3.2.4 The Finite Stored Program Machine

In principle it is wrong to use computing devices that have a bounded finite memory. Given a machine with a particular bound on the size of its memory, we can easily specify a computation that cannot be carried out correctly on that machine. However, practical considerations leave us no alternative.

In a less stringent sense, a finitely bounded computer can be quite acceptable. We need only ensure that whenever the bounds are violated, computation ceases in an unambiguously erroneous state. We may then use the device reliably: when it computes an answer, we know that the answer is correct. When it crashes, we know that we need to pay for a machine with a larger store or word size.

The FSPM implements the (unbounded) GSPM in this weaker sense. It uses finitely bounded data objects to emulate the behavior of the GSPM whenever possible.

3.3 The VLISP PreScheme Compiler

It might seem desirable to implement systems programming tasks in Scheme. Scheme's expressivity, simplicity, and regularity draw programmers. Scheme implementations usually provide a highly interactive programming environment with dynamic linking and powerful debugging tools.

There is a problem with this use of Scheme. Scheme implementations provide an extensive run-time system which includes automatic storage reclamation, first class procedures, and many other advanced features. Most systems programming tasks cannot afford the overhead of this run-time system.

PreScheme is a restricted dialect of Scheme intended for systems programming. It was invented by Jonathan Rees and Richard Kelsey [15]. The language was defined so that programs can be executed using only a C-like run-time system. A compiler for this language will reject any program that requires run-time type checking and need not provide automatic storage reclamation.

PreScheme was carefully designed so that it syntactically looks like Scheme and has similar semantics. With a little care, PreScheme programs can be run and debugged as if they were ordinary Scheme programs.

VLISP PreScheme is our system's programming oriented Scheme dialect inspired by PreScheme. The major syntactic difference between the two dialects of PreScheme is that VLISP PreScheme has no user defined syntax or macros, or compiler directives. The only other difference is that they each provide a different set of standard procedures.

The compiler for VLISP PreScheme does not accept the entire language. Compilation occurs in three stages, the first stage produces a new program by expanding most of VLISP PreScheme's derived syntax. The second stage translates the macro-free program into an equivalent program by using meaning preserving transformations. If the translation succeeds, the resulting program is in a very restricted subset called Bounded PreScheme. This subset has properties that make it easy to compile. One of its properties is that all programs require a bounded amount of storage for control information, hence the name. The final stage of compilation translates Bounded PreScheme into machine language.

The Bounded PreScheme language was inspired by Pure PreScheme, a language defined by Dino Oliva and Mitch Wand. They are creating a verified compiler for Pure PreScheme [21]. A straightforward translation converts a Bounded PreScheme into a Pure PreScheme program. The Pure PreScheme compiler was developed and verified by Dino Oliva and Mitchell Wand, and is documented in detail in [21, 22].

3.4 The VLISP Bootstrap Process

VLISP is certainly the only verified programming language implementation there is that can be used to bootstrap itself. The possibility of a bootstrap underlines the power of the Scheme source language, as well as the comprehensive character of the VLISP implementation. Most previous work in compiler verification has not had any possibility of performing a bootstrap for any of several reasons:

- The source language supported by the implementation was quite different from the language used to develop the implementation itself, the source language being low-level, and the implementation language being the language of an automated proof system [20, 6];
- The verification covered only a limited portion of the implementation, and a bootstrap would mainly exercise the unverified portion [23, 5];
- Only certain key ideas in an implementation were verified, not a whole implementation [many examples].

<code>vscm.scm</code>	Scheme to Byte Code compiler
<code>vvm.scm</code>	The VLISP VM in VLISP PreScheme
<code>vps.scm</code>	The VLISP PreScheme Front End
<code>pps.scm</code>	The Pure PreScheme compiler
<code>prims.c</code>	I/O primitives for Pure PreScheme

Table 4: VLISP Source Files

Structure of the Bootstrap There are five source files and two binary files in a VLISP distribution. One binary file is an executable version of the VLISP VM and the other is an image which results from compiling the Scheme to Byte Code compiler. The source files are listed in Table 4. One cycle in the bootstrap process involves using these files to produce a new version of the binary files.

The first action of the VM is to read an image which encodes the initial state of the machine. One produces a new version of the Scheme to Byte Code image by simply using the old image to compile its own source. This new image is used to compile the other Scheme programs, `vps.scm` and `pps.scm`. A new VM is produced by first translating the source for the VM into Pure PreScheme using the image produced from `vps.scm`, and then into assembly language using the image produced from `pps.scm`. An executable is produced by linking the assembly language with Pure PreScheme's I/O primitives from `prims.c`. Since I/O primitives simply call the operating system, and since we are not in a position to prove anything about the effect of a call to the operating system, we did not consider it worth the trouble to code the I/O primitives directly in assembly language. Apart from compiling these C-coded primitives, the only task of the `gcc` C compiler is to combine the result with the assembly language output `vvm.s`, and to call the assembler.

One cycle in the bootstrap processes is diagrammed in Figure 2. The small circles in the diagram indicate that the virtual machine given by its right arrow is executed. The image used to initialize the machine is indicated by the top arrow. The input is taken from the file named on the left arrow, and the output is written to the file named on the bottom arrow.

Results of the Bootstrap Table 5 presents the time required for each step when run on a Sun4 IPX, a 25 SPECmark SPARC-based computer.

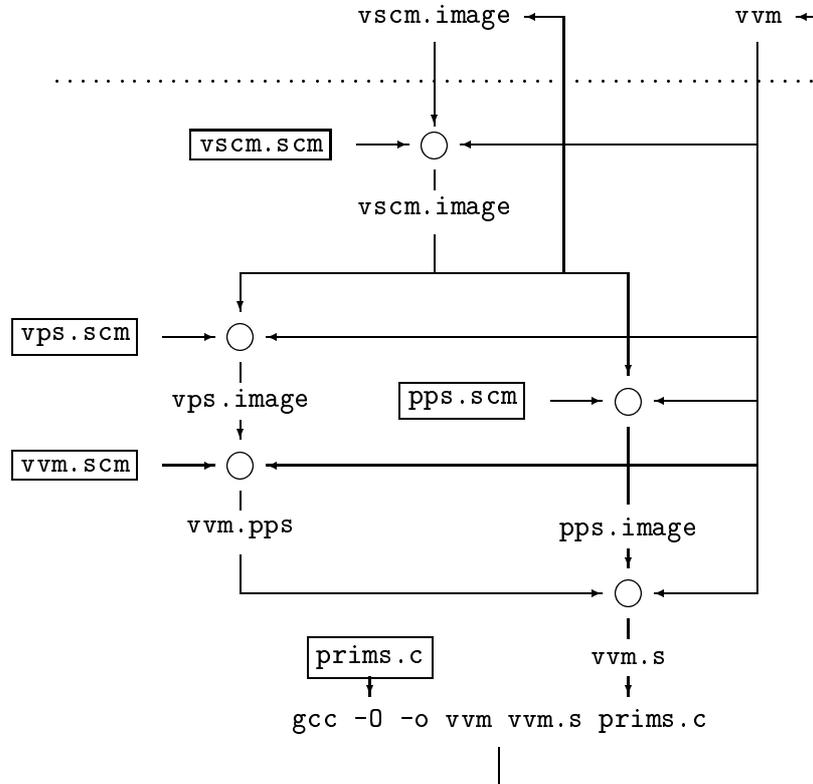


Figure 2: VLISP Bootstrap Process

Image	Input	Output	Sun4 Run Time (min.)
old vscm.image	vscm.scM	vscm.image	49
vscm.image	vps.scM	vps.image	64
vscm.image	pps.scM	pps.image	2664 (44 hrs.)
vps.image	vvm.scM	vvm.pps	36
pps.image	vvm.pps	vvm.s	24

Table 5: Sun4 Bootstrap Run Times

After the first bootstrap cycle, the assembly source for the VM is unchanged by any succeeding bootstrap cycle. After the first bootstrap cycle, the image of the Scheme to Byte Code compiler is unchanged by a pair of two bootstrap cycles. A single bootstrap operation produces a slightly different image, however, because the Scheme to Byte Code compiler uses a “gensym” procedure to generate fresh symbols while expanding Scheme’s derived syntax. This procedure, when called with an argument *sym*, returns a symbol of the form *symn*, where *n* represents the least integer such that *symn* does not yet exist as a symbol. Hence, the symbol generator will not generate the same symbols being used by the executing compiler. For instance, if the currently executing image uses *sym0* but not *sym1*, then the next image will use *sym1* but not *sym0*, and conversely. Hence two cycles in the bootstrap process will restore the original names.

Two initial versions for the VM were constructed. The Scheme sources for the front end and the Pure PreScheme compiler were compiled using Scheme->C [2]. These programs were used to compile the VLISP VM. A second initial VM was constructed by directly implementing the VM algorithms in C.

Two initial images of the Scheme to Byte Code compiler were also constructed. The Scheme to Byte Code compiler was compiled using Scheme->C, and this executable was used to compile the source. The image was also constructed using Scheme48 [13]. The bootstrap process was insensitive to the initial image or VM.

VLISP Virtual Machine Performance In order to estimate the efficiency of our interpreter program, as compiled by the VLISP PreScheme Front End and the Pure PreScheme compiler, we have also compared it with CVM, the C implementation of the algorithms.

CVM was carefully optimized to ensure that it achieved the maximum performance possible with the type of interpreter we implemented. For instance, we scrutinized the Sun 4 assembly code produced by the gcc C compiler to ensure that registers were effectively used, and that memory references into arrays were optimal.

Differences in performance are due to two main sources. First, we structured the virtual machine program to facilitate our verification. In particular, we used abstractions to encapsulate successive refinement layers in the machine. We also have many run-time checks in the code. They ensure that the conditions for application of the formally specified rules are in fact met

as the interpreter executes. Second, `gcc` has sophisticated optimizations, for which there is nothing comparable in the Pure PreScheme compiler.

Nevertheless, the ratio of the speed of CVM to the speed of the VLISP virtual machine, when run on a Sun 3, is only 3.3. On the Sun 4, it is 6.3. The ratio is higher on the Sun 4 partly because `gcc` makes very good use of the large Sun 4 register set. Also, `gcc` does instruction scheduling to keep the Sun 4 instruction pipeline full when possible. Finally, some optimizations to primitives in the PreScheme compiler have been implemented in the Sun 3 version, but not yet in the newer Sun 4 version.

We consider these numbers quite good, particularly because there are several additional optimizations that could be verified and implemented for the PreScheme compiler.

4 Structure of the Verification

The verification has been structured into seven main parts.

- A denotational portion, which makes use of the denotational semantics of Scheme, the BBC and the TBC to prove that the compiler proper and the tabulator preserve meaning. These theorems are proved in [10].
- A proof [7] that an operational semantics for the Tabular Byte Code is faithful to the denotational semantics used in the previous theorem.
- An operational portion of the proof of the extended compiler, covering the correctness of the flattener [11], the linker [8], and the image builder [31].
- A succession of proofs that each step in the refinement of the interpreter program is faithful to its predecessor. These proofs are all operational in character. They are presented in [30].
- A proof that the VLISP Front End faithfully translates VLISP PreScheme, the implementation language of the VM, into Pure PreScheme [24]. It is denotational in character, and is a variant of the transformational approach to compilation proposed by Kelsey [14]. However, this variant is more suited to formal verification because all the transformations take place within the same language, and preserve meaning with respect to the same semantics.
- A denotational proof [21] that the first portion of the Pure PreScheme compiler faithfully translates Pure PreScheme into an intermediate language. It uses techniques similar to those of Clinger [5].
- A proof [22] that the second portion of the Pure PreScheme compiler faithfully represents the objects manipulated by the intermediate language. There are two main elements to this representation: first, tree-like data objects must be laid out in a linear store; and second, tagged objects must be implemented by untagged bit-patterns.

4.1 The Extended Compiler

The proof of correctness of the extended compiler is spread across a succession of reports and involves many techniques in operational and denotational semantics.

4.1.1 The Byte Code Compiler

There are essentially two main theorems to be proved about the byte code compiler [10, Theorems 2 and 23].

Theorem 2 asserts that the output of the compiler is always syntactically correct Basic Byte Code BBC, as defined by the BNF for the language. This is a straightforward inductive proof. It follows the algorithmic structure of the compiler itself very closely. The proof was nevertheless very informative, as the BNF for BBC was initially wrong in several different ways.

Theorem 23 is a descendent of Clinger's compiler correctness theorem [5]. It establishes that the output of the compiler is always semantically equivalent to its input. However, a great deal of work was needed in order to define an appropriate notion semantic equivalence, and to adapt his theorem to it. This is because the official semantics of Scheme are too general in the sense discussed in Section 2.3.1.

4.1.2 The Tabulator

The tabulator required three main theorems. These include a syntactic correctness theorem [10, Theorem 26] and a semantic correctness theorem [10, Theorem 39]. Each is a straightforward inductive proof. In addition, we prove that the output of the tabulator has a special property, namely that it respects its template table [10, Theorem 36]. The proof of correctness for the VLISP flattener [10] uses the assumption that its input has this property.

4.1.3 Faithfulness of the Operational Semantics

The remaining algorithms in the extended compiler are proved correct relative to a succession of operational semantics. If the upper levels of the implementation are shown correct relative to a denotational semantics, and the lower levels are shown correct relative to an operational semantics, why should we believe that the whole implementation is correct relative to *any* single semantics?

Since the transition from the denotational style to the operational style is carried out at the level of the Tabular Byte Code (TBC) output by the tabulator, we will discuss the issue in those terms. The TBC denotational semantics associates a function with each program; when this function is supplied with arguments of appropriate kinds, it returns a value which is considered the ultimate answer computed by the program. We need not be concerned what kind of object this answer actually is.

The operational semantics for the TBC describes a kind of state machine. The states of this machine have components that are described by the syntactic terms of a particular language, which we will call the “Augmented Byte Code” ABC. Some of these components contain templates, and thus involve pieces of the code. Naturally, it is possible to apply the denotational semantics to these state components. Other state components describe Scheme objects, and it is also possible to extend the denotational semantics to apply to these expressions. Combining these various pieces, one can define a map that applies to an arbitrary state description, and returns a value in the denotational semantics of the kind that serves as a computational answer. The map can be defined so, in the case of a final state, at which the state machine halts, the computational answer may be very easily read off from the state. If s is a term in the ABC, we may refer to its value under this map as $\mathcal{D}[[s]]$. We must now prove two assertions:

- For an initial state s_0 , $\mathcal{D}[[s_0]]$ corresponds directly to the value used in the compiler proof in [10].
- $\mathcal{D}[[s]] = \mathcal{D}[[tr(s)]]$, where tr is the transition function of the state machine that gives the TBC operational semantics. In proving this, we rely on the fact that the individual transition rules for the state machine are directly modeled on the clauses in the denotational semantics.

If a computation proceeds from initial state s_0 to a final state s_f , then we may conclude that the answer we read off from s_f is identical to the answer predicted in the compiler correctness theorem. Hence, when the operational semantics determines a result, that result is identical to the one predicted by the denotational semantics.

4.1.4 The Flattener

The correctness of the flattener (linearizer) is the first of many state machine refinement proofs [11]. The correctness theorem [11, Theorem 11] concerns a more abstract state machine, which executes TBC byte code, and a more concrete state machine, which executes linearized FBC byte code. It establishes that if the abstract machine is started with a program t , and the concrete machine is started with the result $F(t)$ obtained by executing the flattener on t , then the two machines compute the same eventual value. This means that if either machine terminates with a number in its value register, then the other machine terminates with the same value in its value register.

Otherwise neither machine is considered to have computed a non-bottom computational result.

The proof follows a simplified version of Wand and Oliva’s “storage layout relations.” We define a relation of *similarity* between states of the more abstract state machine, which executes TBC byte code, and states of the more concrete state machine, which executes linearized FBC byte code.

The proof then consists of two main parts. First, we show that the flattener establishes similarity [11, Theorem 8]. That is, if s is an initial state of the abstract machine, using the program t , and s' is the corresponding initial state of the more concrete machine, using the program $F(t)$, then s and s' are similar. Second, we show that the computation of the two machines preserves similarity [11, Theorem 10]. This proof required an exhaustive examination of the behavior of the machines under each of the byte code rules. Since for terminal states similarity as defined immediately entails equality of computational result, a straightforward induction on the lengths of computation histories establishes the main theorem.

4.1.5 The Linker

The correctness of the linker [8] is very similar in general conception to the correctness of the flattener.

4.2 The Virtual Machine

The correctness of the virtual machine is proved in [31, 30]. It consists of four main proofs. The first shows that the image builder treats the stored program machine faithfully as a refinement of the linked byte code machine. The second shows that the microcoded stored program machine gives a correct sequential analysis of the stored program machine. In addition, the microcoded stored program machine stores its stack in the heap in a faithful way. The third proof demonstrates that garbage collection is correct. The last stage shows that the finite stored program machine is correct in a certain sense.

4.2.1 The Image Builder

The image builder proof follows the general pattern of refinement proof established in the flattener and in Wand and Oliva’s work [32]. However, it requires an additional subtlety.

The linked byte code state machine and the stored program machine use the store in essentially different ways. The latter uses the store to hold several components of the state that are not represented in the store in the linked byte code state machine. These components are the program itself, environments, and continuations. As a consequence, when two states correspond intuitively, the placements of corresponding objects within them may freely differ. Thus we need to be able to state the condition under which a linked code (abstract) store is faithfully represented in a stored program machine (concrete) store. However, these stores may contain circular data structures. As a consequence, the correspondence condition for the stores cannot be defined by an ordinary inductive condition. Our solution is to use the power of the second order existential quantifier in stating and proving the main theorems.

Suppose that f is a partial function that maps locations to locations, which means, in effect, natural numbers to natural numbers. Let us say that f is a *store correspondence* between abstract store s and concrete store s' if the following conditions are met:

1. f is injective where defined, i.e. $f(\ell) = f(\ell')$ implies $\ell = \ell'$; and
2. if ℓ is in the domain of s , and f is defined for ℓ , then $f(\ell)$ is in the domain of s' .

Next, suppose that f is a store correspondence between s and s' , and let $\alpha = \langle f, s, s' \rangle$. We may extend f by a straightforward inductive condition to a relation $v \simeq_\alpha v'$ between values v manipulated by the abstract machine and values v' manipulated by the concrete machine. We say that \simeq_α is a *term correspondence* if f is a store correspondence and moreover:

3. if $s(\ell) = v$ and $s'(f(\ell)) = v'$, then $v \simeq_\alpha v'$; and
4. if $c[\ell] \simeq_\alpha v'$, then $f(\ell)$ is defined, for a specific collection of “constructors” c .

Finally, consider an abstract state Σ containing the store s and a concrete state Σ' containing the store s' . We define Σ and Σ' to be similar if there exists a store correspondence f between s and s' , and \simeq_α is a term correspondence, and moreover $v \simeq_\alpha v'$ for each pair of corresponding register values v in Σ and v' in Σ' .

The proof itself must then show how to define a particular f for initial states of the upper and lower machine. Moreover, it must show how to extend a given f for any transition of the abstract state machine that allocates

new memory. Of course, it must also show that the similarity is preserved (with respect to the same f) for transitions that do not allocate new storage.

4.2.2 The Microcoded Stored Program Machine

The proof that the microcoded stored program machine is a faithful refinement of the stored program machine involves two conceptual ingredients. First, it must be shown that particular compositions of state observers and state modifiers accomplish the transitions of the more abstract machine. Second, it must also be shown that a particular strategy for representing the stack in the store is faithful. This latter ingredient necessitates a store correspondence proof like the one for the image builder.

4.2.3 Garbage Collection

The garbage collection proof is again a store correspondence proof following the pattern of the image builder.

4.2.4 The Finite Stored Program Machine

A finite stored program machine (with any fixed word size) is of course not correct in the strongest sense. It cannot simulate the computation histories of the garbage collected machine, because there will always be computations that require more pointers than can be stored in words of the given size.

The finite machine is only partially correct. That is, any computation history of the finite machine simulates a computation history of the garbage collected machine. Thus, if the finite machine returns a computational answer, then the garbage collected machine computes the same computational answer. The finite machine is easily shown to be correct in this weaker sense.

4.3 The VLISP PreScheme Compiler

4.3.1 The VLISP PreScheme Front End

The Front End verification demonstrates that a useful collection of transformation rules are meaning-refining. A rule is meaning-refining if it does not affect non-bottom computational results. More precisely, R is meaning refining if for every program P , if the denotation of P is non-bottom, then the denotation of $R(P)$ equals the denotation of P . The denotation of a program is given by the semantics in [24, Section 3].

Under this criterion, a meaning-refining rule can transform a program with bottom denotation into a program which produces a (non-bottom) answer. For example, the program

```
(define two (+ 1 one))
(define one 1)
two
```

is transformed into a program which produces the answer 2.

This odd behavior is tolerated so as to allow constant propagation without performing a dependency analysis. In the above example, 1 is substituted for the occurrence of the immutable variable `one` even though *undefined* should have been substituted.

The purpose of justifying a rule is to gain confidence in the correctness of the compiler. Justifications focus on aspects of rules which are likely to cause problems. Indeed, several proposed rules were shown to be applicable in contexts which did not preserve the meaning of a program. These rules were modified or eliminated.

Justifications do not focus on all aspects of a rule. The compiler avoids name conflicts by using α -converted expressions. Therefore, issues arising from name conflicts are not addressed. A formal semantics for each primitive has not been provided, therefore, the rules specific to primitives have not been justified. When the justification of a rule is too obvious, it has been omitted, with the exception of the justification of the `if` in an `if`'s consequence rule.

The justification of many rules employs structural induction involving a large number of cases. The complete proof is only sketched, with a detailed analysis provided only for the most interesting cases.

The formal semantics of VLISP PreScheme require that the order of evaluation within a call is constant throughout a program for any given number of arguments. Most proofs assume arguments are evaluated left-to-right, and then the operator is evaluated. The reader will observe that the order of evaluation is relevant only in the rotate combinations rule.

4.3.2 The Pure PreScheme Compiler

The Pure PreScheme compiler proof is described in detail in [21, 22]. It involves techniques similar to those described above.

4.4 Unverified Aspects of VLISP

There are several reasons why we do not assert that VLISP is a complete verification:

- The verification covers the algorithms and data structures used, rather than the concrete code. In most cases the relationship is straightforward, and indeed particular coding approaches were often adopted so that the relationship would be transparent. However, sometimes there is an “interesting” gap between the form of the specification and that of the code. See Section 2.1.1 for a more detailed discussion.
- The VLISP implementation provides all the standard procedures stipulated in the Scheme language definition. However, because no formal specification has been provided for these user-level procedures, we have not had anything to verify these procedures against.

To mitigate this objection, we note that all source code is provided to the user. The majority of the standard procedures are coded in Scheme itself, and can be replaced with any variants the user considers more trustworthy.

- The Scheme language stipulates that some derived syntactic forms should be provided. However, there is no formal account of the expansion process.

A user suspicious of these expansions can write programs directly in the primitive Scheme syntax. This is still a high level programming language by any standard.

- In some areas, such as the VLISP Pre-Scheme Front End [24], or the Clinger-style ([5]) proof in [10, Section 4.2.2], only case that appeared to us to be “representative” or “interesting” have been selected for detailed proof.
- Proofs have not been checked using a mechanical theorem prover.
- Oliva and Wand’s verification of the Pure PreScheme compiler [21, 22] does not cover all of the steps to a concrete assembly language.

In particular, some of the atomic instructions of the verified output correspond to several instructions in MC68020 assembly language. Moreover, conditionals in the code are represented in a tree-structured form.

Unverified filters currently translate this output to MC68020 or SPARC assembly language.

To provide a rigorous verification of this last stage would require two main ingredients. First, the linearization of the code, using jump and conditional jump instructions, and taking numerical operands, must be proved. Second, a rigorous model of the particular assembly language would be needed, to prove that the atomic instructions are faithfully represented. The MC68020 model of Boyer and Yu [3] might serve as a starting point for this.

There are no plans to carry out the second stage. However, Oliva and Wand are now developing techniques for giving proofs of linearization at the assembly code level. When these techniques have been refined, they will be applied to the linearization procedure used in the Pure PreScheme compiler.⁴ This sort of proof is known to be feasible, as a proof of linearization for a somewhat different language has been carried out under the VLISP effort in [11]. Related techniques were developed in [32].

Although we think it is important to make the limits of our work clear, we still think that it compares very favorably with other research in terms of rigor, clarity, and ambitiousness.

⁴We anticipate that the proof of correctness of the Pure PreScheme linearization will be presented as a portion of Oliva's Ph.D. thesis, or as a separate Northeastern University Technical Report.

5 Conclusion

We believe that VLISP has demonstrated that comprehensive verification of a programming language implementation is feasible.

5.1 Directions For Future Work

We will describe three kinds of potential future work. The first consists of a group of efforts that would supplement the current VLISP verification. The second consists in developing a comprehensively verified compiler, in place of the VLISP implementation, which is a mixed compiler/interpreter design. The third consists of a number of potential verifications in which only certain critical portions of a language implementation would be rigorously verified. Naturally, there are also a number of natural applications areas for VLISP Scheme and for VLISP PreScheme, but we will not describe them here.

5.1.1 Supplementary Work

Although we believe that the VLISP verification provides a very high degree of assurance of correctness, there are a number of supplementary activities that could be undertaken:

Extend the PreScheme Compiler Proof The current Pure PreScheme compiler carries its proof down only to an abstract, tree-structured assembly language [22]. Some of its instructions correspond to short sequences of instructions in an assembly language for a CPU such as the MC68020 or the SPARC.

Linearization The main difference between the output of the verified portion of the Pure PreScheme compiler and a concrete assembly language is that the former has not been linearized. However, the linearization algorithm used in the compiler has been carefully scrutinized. The gap in the proof will be filled by our collaborators at Northeastern University. A rigorously proved linearization algorithm for the Pure PreScheme will appear either as a Northeastern University technical report or as a portion of Dino Oliva's Ph.D. thesis.

Formally Model a Concrete Assembler Work has been done on formally modeling the semantics of MC68020 machine language [3]. This work could be made to serve as the basis for a proof that

a translation from the abstract assembly language to a concrete one was faithful to a formal semantics of the latter.

Optimize the Virtual Machine The VLISP virtual machine could be optimized in a number of ways to improve its execution speed. Two examples are:

Incorporate Additional Byte Code Instructions Speed could be improved by incorporating additional instructions to achieve the effect of a common sequence of separate instructions, or to achieve the effect of a common instruction with a particularly common operand. For instance, Scheme48 [13] has special instructions to fetch the value of a local variable in the nearest environment rib, or in the nearest rib but one. It would be straightforward to verify a peephole optimizer on the byte code to replace the currently generated instructions with the new ones. This would be most conveniently done at the Tabular Byte Code level [11].

Stack-Like Treatment of Continuations We recently devised a way of storing the continuation records within the heap, but in an almost stack-like way. As a consequence, when a procedure returns, the virtual machine can simply pop the old continuation. A prototype indicates that this approach saves very substantially on the number of garbage collections. For instance, using our byte code compiler (compiling itself) as a benchmark, we found that two thirds of the garbage collections could be eliminated.

To verify this optimization is not trivial. Nevertheless, it is clear how to approach the task using the current operational semantics. The natural level to carry out this verification is immediately below the level of garbage collection [30].

Mechanize Some Proofs The proofs given in this project are rigorous, but they have not been checked using a mechanical theorem prover such as MITRE's Interactive Mathematical Proof System IMPS [9]. A modest amount of effort was invested early in 1992 to see what would be required to use IMPS to formalize some of our proofs. Much of the reasoning would fit very nicely within the approach that IMPS supports. However, some additional functionality would be called for so that the many abstract syntaxes and state machines used in the VLISP proofs could be effectively presented to the prover. We now have a good grasp on how to supply the needed functionality.

5.1.2 Fully Compiled Implementation

The obvious way to get much better performance than a byte-coded implementation like VLISP can offer is to replace it with a real compiler from Scheme to assembly code. We have done some preliminary work in this direction.

The most natural approach would involve three stages. In the first stage, Kelsey’s transformational method [14] would be used to transform the Scheme source program. These transformations would focus on introducing explicit mechanisms to handle references to lexical variables in the source program and also to handle procedure calls not in tail recursive position. As in our proofs about the VLISP PreScheme Front End [24], we would want to prove that these transformations preserve the semantics of the source program.⁵ The output of this process would be a Scheme program in a special form. References to lexical variables in programs in this form have been replaced by accesses to an environment data structure.

In the second stage, a syntax-directed compiler like our byte code compiler would be used to translate Scheme programs in the special form to PreScheme programs. They would be linked with a set of run-time routines for allocation and garbage collection like those verified in the VLISP virtual machine. Finally, the resulting PreScheme code would be compiled to assembly language. A PreScheme compilation strategy that might supplant Northeastern’s approach [21, 22] is documented in [24, Appendix B].

In retrospect, the VLISP strategy of verifying a byte code interpreter (written in PreScheme) against the Scheme semantics, and separately verifying a PreScheme compiler, has had some drawbacks. The two verification groups (at MITRE and Northeastern) wound up facing some of the same problems, and often wound up using similar solutions. However, we would not then have had a clean, well worked out prior implementation to serve as a model, in the way that Scheme48 served as a model for the byte-compiled implementation. Although direct compilation might have required more “engineering imagination” and “engineering management” than we could have mustered at the beginning of the project, it now seems, by contrast, quite feasible.

⁵The Scheme semantics treatment of procedure objects as tagged by store locations would complicate these proofs. An alternative treatment might have to be introduced.

5.1.3 Verifying Critical Portions of a Compiler

As we mentioned in the Introduction, an alternative approach is not to verify a full implementation, but simply to verify the parts that are most suspect in it. For instance, the implementation for a programming language may do interesting transformations and optimizations before generating a C program; the compiler back end is then simply a C compiler. In the case of an implementation of this type, the main benefit of formal methods may be achieved if the original pre-C transformations are verified. The back end may be considered sufficiently reliable on empirical grounds.

Some approaches to parallel programming would be good candidates for this sort of treatment. Sisal [4] is an implicitly parallel programming language; the programmer writes his algorithm in an essentially applicative form. The Sisal compiler then performs many optimizations, and introduces an explicitly parallel execution model. The final result is C with explicit parallel constructs for particular multiprocessors. Performance of the resulting code is very good. A very natural application of formal methods would be to provide high assurance that the optimizations and explicit parallelization introduced by the Sisal compiler were faithful to a formal semantics for the language. As Sisal does not currently have a formal semantics, this would have to be developed as part of the task.

5.2 The Main Lessons

We would like to emphasize six lessons from our discussion.

Algorithm-Level Verification Algorithms form a suitable level for rigorous verification (see Section 2.1). In particular, they are concrete enough to ensure that the verification will exclude the main sources of error, while being abstract enough to allow requirements to be stated in an understandable way. In addition, rigorous reasoning is fairly tractable. We consider the decision to focus on algorithm-level verification as crucial to our having been able to verify a system as complex as the VLISP implementation.

Prototype but Verify Interleaving the development of prototypes with verification of the algorithms is highly effective (see Section 2.2). The two activities provide different types of information, and together they yield effective and reliable results.

Choice of Semantic Style There are different areas where denotational and operational styles of semantics are appropriate (see Section 2.3). The two methods can be combined in a single rigorous development using (for instance) the methods of [7].

Requirements at Several Levels Some system-level requirements cannot be stated at the level of abstraction appropriate for others (see Section 2.3.2). For instance, we do not know how to give a satisfactory specification of input-output behavior in the top-level Scheme denotational semantics. It is more natural to represent lower down, in an operational framework.

Small Refinement Steps The VLISP proofs separate out a very large number of independent refinement steps (see Sections 4.1.4–4.2.4). In our experience, this was crucial in order to get the insight into the reasons for correctness. That insight is in turn a strict prerequisite for rigorous verification.

Finiteness Introduced Late In our case it was crucial to model the fact that the final concrete computer has finite word size, and can thus address only a finite amount of virtual memory. However, this property is certainly not accurately expressible at the level of the denotational semantics (see Section 2.3.1). Moreover, it complicates many sorts of reasoning. We benefited from delaying this issue until the very last stage, so that all of our proofs (except the last) could use the simpler abstraction.

We believe that these elements have allowed us to carry out a particularly substantial rigorous verification.

References

- [1] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge and New York, 1992.
- [2] Joel F. Bartlett. SCHEME- \rightarrow C a portable scheme-to-c compiler. WRL Technical Report 89/1, Digital Equipment Corporation, January 1989.
- [3] R. S. Boyer and Y. Yu. Automated correctness proofs of machine code programs for a commercial microprocessor. In D. Kapur, editor, *Automated Deduction — CADE-11*, pages 416–430. 11th International Conference on Automated Deduction, Springer Verlag, 1992.
- [4] D. Cann. Retire fortran? a debate rekindled. *CACM*, 35, 1992.
- [5] Will Clinger. The Scheme 311 compiler: An exercise in denotational semantics. In *1984 ACM Symposium on Lisp and Functional Programming*, pages 356–364, New York, August 1984. The Association for Computing Machinery, Inc.
- [6] Paul Curzon. A verified compiler for a structured assembly language. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 253–262. IEEE Computer Society Press, 1991.
- [7] W. M. Farmer, J. D. Guttman, L. G. Monk, J. D. Ramsdell, and V. Swarup. The faithfulness of the VLISP operational semantics. M 92B093, The MITRE Corporation, 1992.
- [8] W. M. Farmer, J. D. Guttman, L. G. Monk, J. D. Ramsdell, and V. Swarup. The VLISP linker. M 92B095, The MITRE Corporation, 1992.
- [9] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: an Interactive Mathematical Proof System. Technical Report M90-19, The MITRE Corporation, 1991. Submitted to *Journal of Automated Reasoning*.
- [10] J. D. Guttman, L. G. Monk, W. M. Farmer, J. D. Ramsdell, and V. Swarup. The VLISP byte-code compiler. M 92B092, The MITRE Corporation, 1992.

- [11] J. D. Guttman, L. G. Monk, W. M. Farmer, J. D. Ramsdell, and V. Swarup. The vLISP flattener. M 92B094, The MITRE Corporation, 1992.
- [12] IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.
- [13] Richard Kelsey and Jonathan Rees. Scheme48 progress report. Manuscript in preparation, 1992.
- [14] Richard A. Kelsey. Realistic compilation by program transformation. In *Conf. Rec. 16th Ann. ACM Symp. on Principles of Programming Languages*. ACM, 1989.
- [15] Richard A. Kelsey. PreScheme: A Scheme dialect for systems programming. Submitted for publication, 1992.
- [16] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. In *SIGPLAN Notices*, volume 21, pages 219–233, 1986. Proceedings of the '86 Symposium on Compiler Construction.
- [17] P. Lee. *Realistic Compiler Generation*. Foundations of Computing. The MIT Press, Cambridge, MA, 1989.
- [18] Robin Milner and Mads Tofte. *Commentary on Standard ML*. The MIT Press, Cambridge, MA, 1991.
- [19] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, MA, 1990.
- [20] J S Moore. Piton: A verified assembly-level language. Technical Report 22, Computational Logic, Inc., Austin, Texas, 1988.
- [21] Dino P. Oliva and Mitchell Wand. A verified compiler for pure PreScheme. Technical Report NU-CCS-92-5, Northeastern University College of Computer Science, February 1992.
- [22] Dino P. Oliva and Mitchell Wand. A verified runtime structure for pure PreScheme. Technical Report (forthcoming), Northeastern University College of Computer Science, 1992.

- [23] W. Polak. *Compiler Specification and Verification*, volume 124 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.
- [24] J. D. Ramsdell, W. M. Farmer, J. D. Guttman, L. G. Monk, and V. Swarup. The vLISP PreScheme front end. M 92B098, The MITRE Corporation, 1992.
- [25] J. A. Rees, N. I. Adams, and J. R. Meehan. *The T Manual*. Computer Science Department, Yale University, fifth edition edition, 1988.
- [26] Jonathan Rees and William Clinger eds. Revised³ report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, December 1986.
- [27] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Wm. C. Brown, Dubuque, IA, 1986.
- [28] Guy L. Steele. Rabbit: A compiler for Scheme. Technical Report 474, MIT AI Laboratory, 1978.
- [29] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, 1977.
- [30] V. Swarup, W. M. Farmer, J. D. Guttman, L. G. Monk, and J. D. Ramsdell. The vLISP byte-code interpreter. M 92B097, The MITRE Corporation, 1992.
- [31] V. Swarup, W. M. Farmer, J. D. Guttman, L. G. Monk, and J. D. Ramsdell. The vLISP image builder. M 92B096, The MITRE Corporation, 1992.
- [32] M. Wand and D. P. Oliva. Proving the correctness of storage representations. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 151–160, New York, 1992. ACM Press.