, , 1–28 ()

VLISP: A Verified Implementation of Scheme*

JOSHUA GUTTMAN guttman@mitre.org
JOHN RAMSDELL ramsdell@mitre.org
The MITRE Corporation
202 Burlington Road
Bedford, MA 01730-1420

MITCHELL WAND wand@ccs.neu.edu
College of Computer Science
161 Cullinane Hall
Northeastern University
Boston, MA 02115

Keywords: verified, programming languages, Scheme, compiler

Abstract. The VLISP project showed how to produce a comprehensively verified implementation for a programming language, namely Scheme. This paper introduces two more detailed studies on VLISP 13, 21. It summarizes the basic techniques that were used repeatedly throughout the effort. It presents scientific conclusions about the applicability of the these techniques as well as engineering conclusions about the crucial choices that allowed the verification to succeed.

Table of Contents

* The work reported here was carried out as part of The MITRE Corporation's Technology Program, under funding from Rome Laboratory, Electronic Systems Command, United States Air Force, through contract F19628-89-C-0001. Preparation of this paper was generously supported by The MITRE Corporation.

1. Introduction

The VLISP project showed how to produce a comprehensively verified implementation for a programming language, namely Scheme [4, 15]. Some of the major elements in this verification were:

- The proof was based on the Clinger-Rees denotational semantics of Scheme given in [15]. Our goal was to produce a "warts-and-all" verification of a real language. With very few exceptions, we constrained ourselves to use the semantic specification as published.

  The verification was intended to be rigorous, but not completely formal, much in the style of ordinary mathematical discourse. Our goal was to verify the algorithms and data types used in the implementation, not their embodiment in code. See Section 2 for a more complete discussion of these issues.

  Our decision to be faithful to the published semantic specification led to the most difficult portions of the proofs; these are discussed in [13, Section 2.3–2.4].

- Our implementation was based on the Scheme48 implementation of Kelsey and Rees [17]. This implementation translates Scheme into an intermediate-level "byte code" language, which is interpreted by a virtual machine. The virtual machine is written in a subset of Scheme called PreScheme.

  The implementation is sufficiently complete and efficient to allow it to bootstrap itself. We believe that this is the first verified language implementation with these properties.

- The proof followed the structure of the Scheme48 implementation. It is organized into more than a dozen separable components, each of which presents an independent verification problem. This elaborate division was crucial to arriving at a tractable verification task.

  However, the proof used only a small collection of basic techniques:

  1. Semantics-preserving source-to-source transformations.
  2. Structural inductions using the denotational semantics, in the fashion of Wand and Clinger [31, 3].
  3. Verification of representations and refinements using operational semantics and the method of storage layout relations [32].
  4. The gap between denotational and operational semantics is bridged by soundness or faithfulness proofs; for the PreScheme compiler an adequacy result was achieved as well.

  Because of this small repertoire of basic techniques, we believe that the proof is accessible to readers without advanced mathematical background. With a very few exceptions, it does not require knowledge of domain theory, category theory, etc.

> We believe that this proof architecture can form a basis for other language-implementation proofs, as it uses both operational and denotational techniques to their best advantage.

The project involved 4 senior researchers and 1 Ph.D. student, with significant contributions by others, during a 3 year period from October 1989 until September 1992, when the verified system bootstrapped. The total effort was about 10 person-years. The present papers were written in 1993 and revised in 1994, with a small amount of additional funding. The specifications and proofs total about 600 pages of technical reports. The final implementation, as used in the bootstrap process, consists of about 10,000 lines of Scheme and 166 lines of C.

The techniques used in the proofs are described in detail in [13, 21]. In this paper, we discuss those aspects of the project that go beyond the individual proofs. In Section 2, we discuss the crucial choices that allowed the verification to succeed, and especially the degree of rigor we have aimed for. In Section 3, we discuss the implementation in more detail and present the results of the bootstrapping experiment. In Section 4, we discuss the structure of the proof in more detail, and introduce the small collection of basic techniques that were used repeatedly in the verification process. In Section 5, we assess the strengths and weaknesses of operational and denotational styles in various stages of the verification. Lastly, in Section 6 we discuss the implications of our work for other language-verification efforts.

One basic condition of our success was of course to embark on a reasonably well specified task. It would have been hopeless to undertake an effort on this scale without there already being a fairly well polished formal semantics for the language to be implemented. The official Scheme semantics [15, Appendix A] has been stable for several years, and it has already served as the basis for substantial work, for instance [3]. To face the modeling issues of how to formalize the semantics at the same time that one is trying to develop verification proof techniques would be very difficult indeed.

We also chose to base our implementation on the well thought out design that Kelsey and Rees used in Scheme48 [17]. Scheme48 is divided into three major parts. The first is a compiler stage, which translates Scheme source programs to an intermediate level "byte code" language. The second stage is an interpreter to execute the complex instructions of the intermediate level language. The interpreter is written in PreScheme, which may be considered as a highly restricted sub-language of Scheme. The third portion is a compiler to translate PreScheme programs, the interpreter in particular, into runnable code. VLISP has adhered not only to this overall structure, but also to many specific choices made in Scheme48.

1.1. What is Scheme?

The Scheme programming language, "an UnCommon Lisp," is defined by the language Report (currently in its fourth revision [4]), and by an IEEE standard [15]. We have taken [15] as our definition.

However, the definition consists, for our purposes, of two radically different parts. The first and much larger part provides a carefully written but non-rigorous description of the lexical and syntactic structure of the language, and of its standard procedures and data structures. Many of the standard procedures are conceptually complex; they are generally implemented in Scheme itself, using a smaller set of procedures as data manipulation primitives.

The short, second portion consists of Appendix A, which provides an abstract syntax and a formal denotational semantics for the phrases in that syntax. The semantics is concise and relatively abstract; however, for this reason, it provides only a loose specification for the implementer. Most of the standard procedures, even the data manipulation primitives, correspond to nothing in the denotational semantics; the denotations of constants are mostly unspecified; moreover, the semantics contains no treatment of ports and I/O.

The VLISP verification has taken Appendix A, in the slightly modified form given in [10], as its starting point. We have therefore concentrated on verifying those aspects of the language that it characterizes; some other aspects—especially the finiteness of address spaces—have only been introduced into our specifications at lower levels in the process of specifying and verifying our implementation.

2. Rigor and Prototyping

In this section we will discuss our approach to achieving rigor, and how we exploited non-rigorous, empirical kinds of evidence.

2.1. The Emphasis on Rigor

Rigor is not identical with formality. A formal theory is always expressed in a fully defined logical theory, with a precise syntax and semantics; formal proofs in the theory are carried out using a particular deductive system that is sound for it. There must be a syntactic decision procedure to establish whether a given purported proof is in fact a formal derivation in the system.

Full formalization is certainly a way of achieving rigor, but not the only way. Rigor has also been achieved when we know how to complete a formalization, even though a substantial amount of skilled labor may be needed to carry it out.

In our view, formality is only a means: the goal is rigorous understanding. We have chosen to emphasize the latter over the former because of our view of the purpose of verification.

Perhaps the most obvious view is that the point of verification is to eliminate all sources of error, or if that is beyond human powers, as many sources of error as possible. That is not our view.

We believe that the point is to provide *insight* into the structure of an implementation and the reasons for its correctness. We believe that several benefits follow from the mathematical insight that rigor produces:

Less Likelihood of Abuse When the intended behavior of software is rigorously described, it is less likely to be subtly misunderstood and put to uses for which it is intrinsically not suited.

Increased Reliability Errors are likely to be dramatically less frequent. Moreover, when they are found, they are likely to be matters of detail rather than fundamental design flaws.

Easier Correction of Errors When errors are encountered, they are apt to be far more easily understood, localized, and fixed.

Greater Adaptability Because of the rigorous specifications of the different portions of the system, those portions are easier to combine with new pieces in a predictable way. New but related functionality is easier to introduce without destroying old functionality.

Because these practical advantages are essentially by-products of rigorous mathematical insight, we have tried to organize the project to emphasize humanly understandable rigor rather than complete machine-checkable formality.

There are two reasons why there is a potential conflict between aiming for rigor and aiming for formality:

- Fully formalized specifications and proofs require great effort, particularly in the case of a project spanning contrasting semantic styles and covering a large implementation in detail. There is a risk that the attempt to achieve full formality would prevent covering large portions of the implementation with any acceptable degree of rigor.

- Full formality requires choices on many details that every logician knows can be handled in a variety of acceptable ways. As a consequence, the proportion of informative content in the total written specification decreases. It may be just as difficult for a human being to extract a rigorous insight into a system from a full formalization of it than it would have been starting from a non-rigorous engineering description. Similarly, fully formalized or mechanized proofs may focus attention on the need to manipulate a particular theorem-proving system at the expense of acquiring and recording an insight into the reasons why the theorems are true and the system correct.

We motivate two practical decisions from this emphasis on rigorous insight. First, we have expressed our specifications in what we believe to be a lucid and reliable form, but without the syntactic constraints of a formalized specification language. Second, we have decided to focus our verification specifically on the algorithms used throughout the implementation, rather than on their concrete embodiment in code. Our division of the proof into a large number of separately understandable verifications was also partly motivated by a desire to make each separate proof surveyable as a whole.

2.1.1. Rigorous, but Not Completely Formal Specifications

VLISP required three main different kinds of languages for specification. In no case did we completely formalize the notation, although there are no interesting mathematical problems that would have to be faced to do so. The first of these rigorous but informal specification languages is the familiar mathematical notation for denotational semantics. The second is a notation for expressing algorithms; for this purpose we have used predominantly the applicative sub-language of Scheme. There are a variety of conventions needed to interrelate these two language, such as conventions relating `cons` and the other Scheme list primitives to the denotational notations for mathematical pairs and sequences. Finally, a language was introduced to present state machine transition functions.

At several points formality might have helped us to save labor, had we had sufficiently flexible mechanized theorem proving support. These were situations in which many detailed cases had to checked, for instance, those described in [13, Sections 3.2 and 6.2]. However, tool flexibility is crucial; these situations involve quite different semantic contexts.

2.1.2. 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 [2].

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, if the verification is lower in the list, the mass of detail is much greater, creating 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 have found that our algorithmic verification was an effective way to ease the development of correct software. Very few bugs emerged in software we wrote based on algorithmic specifications, as opposed to our more exploratory prototype programming. Those we did find were easy to correct, with one notable exception.

Indeed, our most difficult bug, which was very hard to isolate, generally confirms our decision. The problem concerned garbage collection. It arose precisely where we took a shortcut not reflected in the specification, for the sake of efficiency. We omitted the memory operations to ensure that newly allocated memory locations would contain a distinctive value *empty*, since we were sure the program would initialize the memory to the correct useful value before the garbage collector could be called. In this we departed from our specification, which assumed that newly allocated memory contained *empty*. But the garbage collector could in fact be called before the initialization. Hence the garbage collector would follow bit-patterns in the newly allocated memory that happened to look like pointers. Later, however, we specified a version of allocation and garbage collection which does not require the initialization to *empty*; this we successfully verified and implemented.

The algorithmic level of verification was particularly appropriate for our work. This is because we were able to arrange our code so that the great majority of it took on certain special forms. Thus, we could reason about the algorithms in a tractable way, and convince ourselves informally that these abstract algorithms matched the behavior of the code. We used three primary techniques for this purpose.

- Whenever possible we used applicative programs. As a consequence, we could regard the very same text as specifying the abstract algorithm and also as providing its implementation. In reasoning about the abstract algorithm, we used forms of reasoning such as β-reduction freely, which are not valid for all Scheme programs. However, to convince ourselves that the reasoning was a reliable prediction of the behavior of the actual Scheme code, we needed only to check that the procedures did not use state-changing primitives, I/O, or `call/cc`, and that they would terminate for all values of the parameters.

- In many other cases, we organized programs to implement a state machine. This design abstraction uses a dispatch procedure to select one of a set of transformation procedures. Each transformation procedure performs a relatively simple state transformation before tail recursively invoking the dispatch procedure. This is an organization that is very natural for a byte code interpreter in any case. Moreover, it is easy to match a short piece of non-applicative code with the specification for these simple transformation procedures.

- In the case of the PreScheme Front End, we organized the program using a set of *rules* and a control structure. Each rule specifies a simple syntactic transformation on expressions matching a particular pattern. These transformations are proved to preserve the semantics of the source program being compiled. However, the choice of whether actually to apply a rule and the choice of the order to apply rules throughout the source program is made by code implementing a control structure that need not be verified at all. This control code may freely apply the rules in any way that seems heuristically useful in optimizing the source program; since no individual rule can alter the semantics of the source program, any *finite* sequence of rule applications whatever will be safe.

  However, the Front End does not terminate for all inputs. Indeed the question whether, for a given PreScheme program P, there exists a finite sequence of rule applications that transforms P to an acceptable output form appears to be algorithmically undecidable. Thus, we could not reasonably demand a terminating control structure for the Front End.

2.2. "Prototype but Verify"

A traditional view of program verification envisages first designing, developing, and debugging the programs, and then proving that the programs meet their specifications. A contrasting approach [6, 9] is that programs and their proofs should be developed hand-in-hand, with proof ideas leading the way. Our experience with the VLISP project suggests that an intermediate approach is preferable for developing *large* verified programs.

Our approach was to develop initial executable prototypes of the desired programs, using normal software engineering techniques such as modular design and type-checking. We then used proof ideas to *refine* these programs to a form amenable to verification proofs. The programs were partitioned into about a dozen components, each of which presented an independent verification problem. Each component had a clear purpose embodied in the interface specification of the component. We then used programming ideas to *optimize* the algorithms to achieve desired performance characteristics. We continued using proof ideas to refine the algorithms to achieve, or restore, clarity and correctness. Optimizations often made the intended correctness proofs much harder, so we used the prototype to estimate performance improvements to decide whether the benefits were worth the costs.

Thus, we found it indispensable to have running prototypes of portions of the system being verified, and eventually of the entire integrated system. Portions of the prototype were gradually replaced by rigorously verified code as it was developed.

3. The VLISP Implementation

Although the VLISP implementation is similar to Scheme48, the two systems differ in three main ways:

- Scheme48 uses a real stack, unlike VLISP, which stores the corresponding information in the heap. The Scheme48 stack is a fairly complex data structure, and it is not clear what invariants it maintains during execution. The Scheme48 stack also necessitates a more complex garbage collector.
- Scheme48 adds a variety of additional byte codes to speed execution of common patterns. Although most of them would not have called for changes of method, they would have added to the bulk of the verification, both in the extended compiler and in the virtual machine.
- Scheme48 handles interrupts, such as the user typing control-C at the keyboard to interrupt a long-running procedure. It was not clear how to formalize this.

Scheme48 was under active development as we were working. Moreover, its code is sparsely commented, so the easiest way to get a deep understanding of the code we were to verify was to write it ourselves.

3.1. The VLISP Bootstrap Process

We believe that VLISP is the only verified programming language implementation that has been used to bootstrap itself.

One cycle in the bootstrap process produces new versions of two binary files by executing old versions of those files. One of these binary files contains an executable version of the VLISP VM; we call this file `vvm`. The other contains a particular byte code program that the VM will execute, namely the byte code image of the VLISP byte code compiler itself. We call this file `old-vscm.image`.

The process requires five main steps; in each step another source file is used as input. In each of these five steps, in addition to the input file, there are two other ingredients. These are the executable version of the VLISP VM, and some byte code image that the VM can run. In the first step, this byte code image is `old-vscm.image`. The five steps are:

1. Produce `vscm.image` from input file `vscm.scm` using `vvm` as virtual machine and `old-vscm.image` as the byte code program. `vscm.scm` is the Scheme source for the VLISP byte code compiler. This step reconstructs the first of the two binary files. The remaining steps are all devoted to reconstructing the other, namely the executable `vvm`.

2. Produce `vps.image` from input file `vps.scm` using `vvm` as virtual machine and `vscm.image` as the byte code program. The input `vps.scm` is the Scheme source for the VLISP PreScheme Front End program, and `vps.image` is a byte code image of it.

3. Produce `pps.image` from input file `pps.scm` using `vvm` as virtual machine and `vscm.image` as the byte code program. The input file `pps.scm` is the source of the Pure PreScheme compiler, and `pps.image` is its compiled byte code image.

4. Produce `vvm.pps` from input file `vvm.scm` using `vvm` as virtual machine and `vps.image` as the byte code program. The input file `vvm.scm` is the VLISP PreScheme source of the virtual machine, and the output `vvm.pps` is the translation of it into Pure PreScheme produced by running the Front End. We always give files containing VLISP PreScheme a filename extension of `.scm`, because they are in fact Scheme programs, even though they are of a very special kind that can also be executed in a different way.

5. Produce `vvm.s` from input file `vvm.pps` using `vvm` as virtual machine and the file `pps.image` as the byte code program. The output is the assembly language version of the new VLISP VM.

To create the new `vvm` executable, we then use `gcc` to link `vvm.s` with the C code for the primitives that call the operating system, and to assemble the results:

```
gcc -O -o vvm vvm.s 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 and linker. The structure of this process is expressed in Figure 3.1. In this figure, the left, top and right in-arrows are always the input file, the byte code image, and the virtual machine executable, respectively.

Two initial executable versions `vvm` and `cvm` for the VM were constructed.

`vvm` We compiled the VLISP VM using the VLISP Front End and the Pure PreScheme compiler. These in turn had been compiled using Scheme->C [1].

`cvm` This VM was constructed by directly implementing the VM algorithms in C.

Two versions of the initial image `old-vscm.image` of the Scheme to Byte Code compiler were also constructed.

- The Scheme to Byte Code compiler source was compiled using Scheme->C, and this executable was used to compile the same source to a byte code image.
- Another image was constructed using Scheme48 [17] to run the Scheme to Byte Code compiler on itself as input.

The bootstrap process was almost completely unaffected by the choice of initial image or VM. The only difference concerned the default case of the Scheme reader. When the system is bootstrapped from an implementation that reads symbols in a particular case by default, the resulting image retains the property. We found this behavior necessary in order to allow VLISP to be bootstrapped starting with any Scheme implementation, regardless of its default case. When we modified Scheme48 to cause its reader to prefer upper case, we obtained exactly the same results as from Scheme->C.

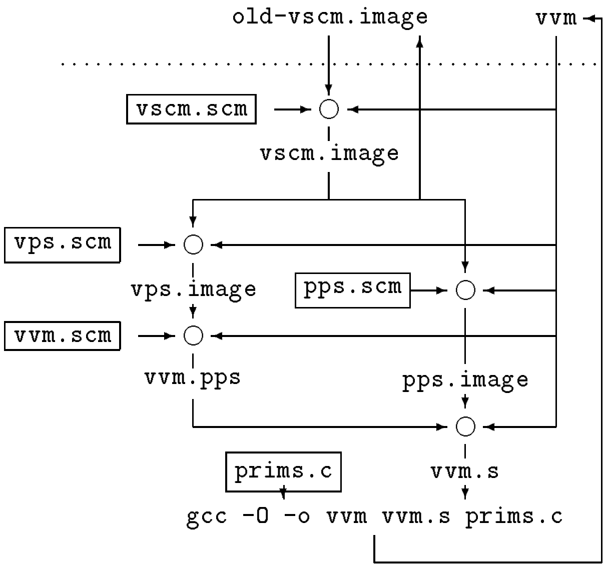

Figure 0. VLISP Bootstrap Process

Table 1. Bootstrap Run Times in Minutes, using `vvm`

Input	Byte Code Image	Output	Sun4	Sun3
`vscm.scm`	`old-vscm.image`	`vscm.image`	26	262
`vps.scm`	`vscm.image`	`vps.image`	34	352
`pps.scm`	`vscm.image`	`pps.image`	19	192
`vvm.scm`	`vps.image`	`vvm.pps`	196	
`vvm.pps`	`pps.image`	`vvm.s`	13	

After the first bootstrap cycle, the assembly source for the VM is unchanged by any succeeding bootstrap cycle. Because of a peculiarity in the way symbols are generated, the image of the Scheme to Byte Code compiler is unchanged by a pair of *two* bootstrap cycles.

3.2. VLISP Virtual Machine Performance

Table 1 presents the time required for each step when run on a Sun SPARCStation 10.

Table 2. Bootstrap Run Times in Minutes, using `cvm`

Input	Byte Code Image	Output	Sun4	Sun3
`vscm.scm`	`old-vscm.image`	`vscm.image`	4.3	78.8
`vps.scm`	`vscm.image`	`vps.image`	5.5	87.6
`pps.scm`	`vscm.image`	`pps.image`	3.2	49.9
`vvm.scm`	`vps.image`	`vvm.pps`	33.8	
`vvm.pps`	`pps.image`	`vvm.s`	2.3	

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 with its optimizing flag `-O` to ensure that registers were effectively used, and that memory references into arrays were optimal. Timings are indicated in Table 2.

Differences in performance are due to three main sources. First, `cvm` was programmed to keep certain crucial global variables in machine registers. The PreScheme compiler currently does not put any global variables into machine registers. Second, 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. Finally, `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 between 3 and 4. On the Sun 4, it is about 6. 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 reasonably good, particularly because there are many additional optimizations that could be verified and implemented for the PreScheme compiler. Indeed, more recent variants of the PreScheme implementation seem to perform much better.

4. Structure of the Proof

The Vlisp implementation is organized into more than a dozen separable components, each of which presents an independent verification problem. This division was crucial to arriving at a soluble verification task.

4.1. Refinement Layers

The proof steps are grouped into three major parts, corresponding to the major parts of the implementation. The first half dozen are devoted to the compiler stage, which translates Scheme source programs to byte code. A second group of three main verifications justify the interpreter that runs the resulting byte code programs. We frequently refer to the interpreter program as the VLISP Virtual Machine or VM. The interpreter is written in VLISP's variant of PreScheme. The compiler for VLISP PreScheme also required several verification steps.

Each of the proofs connects one programming language equipped with a semantics with another, although in some steps only the semantics may differ. We always provide the semantics either in the form of a standard denotational theory or in the form of a state machine. Thus the natural interfaces between the independent components are always a language equipped with a semantics. Each component is justified by showing the component exhibits the semantics required for its upper interface assuming its lower interface meets its semantic definition.

The large number of components was crucial to achieving a tractable and understandable verification. The sharply defined interfaces allowed division of labor. We could undoubtedly have worked far more efficiently if we had defined these interfaces earlier in our work, instead of trying to make do with a smaller number of less closely spaced interfaces.

4.2. The Main Techniques

The VLISP project turned out to require only a relatively small collection of techniques, several of which we were able to use repeatedly in different portions of the verification. We consider this a lucky outcome, as we think that these techniques can be made routine and applied in a variety of projects.

In this section we will summarize the main techniques used in the project. More detailed descriptions are spread out in [13, 21]. They play specific roles in the architecture that we have developed.

The VLISP Architecture. The VLISP Scheme and PreScheme implementations share a common structure. Each may be divided into three main sections:

1. A stage of source-to-source transformations. In the VLISP Scheme implementation this consists merely of expanding defined syntax, and since the latter has

no role in the formal semantics, the verification process begins only after the stage.

By contrast, this is a major element in the PreScheme implementation, comprising the Front End [21, Section 3]. The front end does a wide range of transformations designed to bring the source PreScheme code to a very special syntactic form that can be compiled more easily to efficient code. Each of these transformations is proved to preserve the denotation of the program.

The same transformational approach can be used to justify many source-to-source optimizations in other programming languages, including Scheme itself [27, 18, 16].

2. A syntax-directed compiler in the Wand-Clinger style [31, 3]. Its purpose is to analyze the procedural structure of its source code. The compilation algorithms use recursive descent, and the proof of correctness is a corresponding structural induction on the syntax of the source code. The proof establishes an equality (or a strong equivalence) between the denotations of its input and output.

3. A succession of representation decisions and optimizations. These steps are justified by reference to an operational semantics, by supplying proofs that one state machine *refines* another. For this we have repeatedly used the method of storage layout relations [32, 13, Section 4.2]. Here the main proof technique is induction on the sequence of computation steps that the state machines take.

Between the denotational methods of steps 1 and 2 and the operational methods of step 3, there is needed a proof of "faithfulness." This proof is needed to show that the first operational semantics is a sound reflection of the last denotational semantics, or, in essence, that answers computed by the state machine are those predicted by the denotational semantics.

We will briefly discuss techniques for the three steps above as well as for the proof of faithfulness.

4.2.1. Transformational Compilation

The front end of the VLISP PreScheme compiler implemented a significant number of optimizations as source-to-source transformations. The optimizations implemented include constant propagation and folding, procedure inlining (substituting the body of a procedure at a call site), and β-conversion. These optimizations allow one to write programs using meaningful abstractions without loss of efficiency.

These transformations were justified relative to the formal semantics VLISP PreScheme. Each transformation T is meaning-refining, by which we mean that, for any program P, if the semantics predict that executing P will produce a non-bottom answer, then

$$\mathcal{P}[\![P]\!] = \mathcal{P}[\![T(P)]\!],$$

where $\mathcal{P}$ is the semantic function for PreScheme programs.

The correctness proofs were made possible by carefully designing the VLISP Pre-Scheme semantics as well as the form of each transformation rule. Three differences from the Scheme semantics greatly facilitate the justification of transformations that involve considerable code motion.

- A `lambda`-bound variable is immutable and no location is allocated for it.
- A procedure has no location associated with it, so its meaning depends only on the environment in which it was defined, and does not depend on the store as it is when the procedure is defined.
- The semantics of a `letrec` expression was defined in a fashion that ensured that the meaning of its bindings also depend only on the environment in which it is defined.

Many of the transformation rules looked unusual. For example, a set of interacting rules was used to implement β-conversion. The most significant contribution of the work on the front end is the identification of a collection of transformation rules that can both be verified relative to the formal semantics of the source language, and can also form the basis of a practical optimizing compiler.

The Front End itself uses a sophisticated control structure to apply the rules in an efficient way. In some cases the algorithm refrains from applying a rule to a phrase to which it is validly applicable, for instance, if the resulting code would become too large. However, since every change to the program is in fact carried out by applying the rules, the correctness of the rules guarantees that there is no issue of correctness for the heuristics embodied in the control structure.

4.2.2. Wand-Clinger style Compiler Proof

The Wand-Clinger style [31, 3] of compiler proof is designed to prove the correctness of a compiler phase that takes a source language to a tree-structured intermediate code. In the forms that we use, it distinguishes tail recursive and non-tail recursive calls and arranges the evaluation of arguments to procedure calls. It is also responsible for analyzing conditional expressions, and for replacing lexical variable names with numerical lexical addresses that reference the run-time environment structure. It does not analyze the primitive data structures supported by the programming language at all.

One of the ideas of [31] was to define the semantics of the target language using the same semantic domains as the source language. Thus, the correctness of the compiler could be stated as the assertion that the denotation of the output code (possibly when supplied some initial parameters) is equal to the denotation of the source code. The Wand-Clinger style correctness theorem for PreScheme [21, Theorem 4] takes just this form. Although the corresponding theorem for the Scheme implementation [13, Theorem 12] is more complex, it is still a strong form of equivalence of denotation.

The compiler algorithm is a straightforward recursive descent: to compile (e.g.) a lambda expression, one compiles its body and prefixes some additional code to it, which will place the arguments to the call in a run-time environment display rib. Similarly, to compile a conditional, one compiles the test, consequent and alternative, and then joins the resulting pieces together with some additional code to make the selection based on the value computed by the test code. In each case, the results of the recursive calls are combined according to some simple recipe.

Because the algorithm is a recursive descent and the correctness condition is (in the simplest versions) an equality of denotation, a natural form of proof suggests itself, namely an induction on the syntax of the source code. Each inductive case uses the hypothesis that all subexpressions will be correctly compiled to target code with the same denotation. Thus, in effect the content of each inductive case is to show that way the target code combines the results of its recursive calls matches the way that the semantics of the source language combines the denotations of its subexpressions.

Naturally, in practice the situation is somewhat more complex. There may be several different syntactic classes of phrases in the programming language (as there are in the PreScheme verification, for instance), and each of these will furnish a differently stated induction hypothesis. There may also be different recipes depending on syntactic features, for instance whether a procedure call is tail recursive or not. These also lead to distinct induction hypotheses. Moreover, the algorithm must pass some additional parameters, most importantly a "compile-time environment" which specifies how occurrences of lexical variables are to be translated into references to the run-time environment structure. The induction hyptheses, which are based on [3], spell out how this data is to be used.

4.2.3. Faithfulnesss of an Operational Semantics

An essential ingredient in the effectiveness of the Wand-Clinger style of compiler proof is that it should be relatively easy to give an operational interpretation as well as a denotational interpretation to the target code. We accomplish this by selecting a target code with a very regular denotational semantics. In the Scheme byte compiler, for instance, we follow Clinger [3] in giving each instruction a denotation that may be regarded as acting on four registers together with a store; in most cases, the denotation of an instruction invokes the code that follows it with different values for some registers or the store. As a consequence of this regularity, it is fairly straightforward to write down a corresponding set of operational rules for a state machine.

A Simple Example. Let us compare the clause specifying the denotation of a `make-cont` instruction, which builds a continuation preparatory to making a procedure call in non-tail recursive position, with the operational rule for the same instruction. They are presented in Table 3.

Table 3. Denotational and operational treatments of `make-cont`

$\mathit{make\_cont} : \mathtt{P} \to \mathtt{N} \to \mathtt{P} \to \mathtt{P}$
$\mathit{make\_cont} = \lambda\pi'\nu\pi\ .\ \lambda\epsilon\epsilon^*\rho_R\psi\ .\ \#\epsilon^* = \nu \to \pi\epsilon\langle\rangle\rho_R(\lambda\epsilon\ .\ \pi'\epsilon\epsilon^*\rho_R\psi),$
$\qquad\qquad\qquad\qquad\qquad\mathit{wrong}$ "bad stack"

Rule 1: **Make Continuation**
Domain conditions: $\quad b = \langle\texttt{make-cont}\ b_1\ \#a\rangle :: b_2$
Changes: $\quad b' = b_2; \quad a' = \langle\rangle\ ; \quad k' = \langle\textsc{cont}\ t\ b_1\ a\ u\ k\rangle$

In the denotational clause, π' refers to the code to be executed after the next return, while π refers to the code to be executed immediately after this instruction. The ν is an integer representing the compiler's expectation about the height of the argument stack at execution time. The second block of λ-bound variables represent the register values when the instruction is executed.

In the operational rule, b represents the current code, a represents the argument stack, and k represents the continuation register; a machine state takes the form $\langle t,\ b,\ v,\ a,\ u,\ k,\ s\rangle$. In place of the denotational version's explicit conditional testing $\#\epsilon^* = \nu$, the operational rule is applicable only if $\#a$ appears as argument to the instruction. There is simply no rule covering the other case, so that if the equality fails, then the operational semantics predicts that the state machine will not advance past its current non-halt state.

Primed variables in the *Changes* section represent the values of the registers after the state transition, and registers for which no primed value appear are left unchanged in the transition; so the rule can be written more explicitly as:

$$\begin{array}{l}\langle t,\ \langle\texttt{make-cont}\ b_1\ \#a\rangle :: b_2,\ v,\ a,\ u,\ k,\ s\rangle \\ \quad\Longrightarrow \langle t,\ b_2,\ v,\ \langle\rangle,\ u,\ \langle\textsc{cont}\ t\ b_1\ a\ u\ k\rangle,\ s\rangle\end{array}$$

Just as ϵ and ρ_R appear unaltered as arguments to π in the consequent of the conditional, v' and u' do not appear in the *Changes* section of the rule. The compound term $\langle\textsc{cont}\ t\ b_1\ a\ u\ k\rangle$ codes the same information as the denotational continuation $(\lambda\epsilon\ .\ \pi'\epsilon\epsilon^*\rho_R\psi)$, and the two treatments of the **`return`** instruction, which may eventually invoke the continuation, arranges to treat them in parallel ways.

Form of the Faithfulnesss Proof. The main idea of the Scheme faithfulness proof is to associate to each state in the oeprational semantics a denotation in the domain $\mathtt{A}$ of answers. If Σ is an initial state, then the denotation of Σ agrees with the denotational value of of the program that it contains, when applied to the initial parameters used in the compiler proof. For a halt state Σ, the denotation is the number contained in its value register (accumulator), and the bottom answer $\bot_{\mathtt{A}}$ if the value register does not contain a number. This choice must be compatible with the initial continuation selected to supplement the official denotational semantics.

Moreover, for each rule, it is proved that when that rule is applicable, the denotation of the resulting state is equal to the denotation of the preceeding state.

Thus in effect the denotation of an initial state equals the expected denotational answer of running the program on suitable parameters, and the process of execution leaves the value unchanged. If a final state is reached, then since the operational answer function *ans* is compatible with the initial continuation, the computational answer matches the denotational answer, so that faithfulness is assured.

In many cases, more can also be proved. For instance, Theorem 9 of [21] establishes the *adequacy* of an operational semantics. By an adequacy theorem we mean a converse to the faithfulness theorem, showing that the operational semantics will achieve a final state when the denotational semantics predicts a non-$\perp$, non-erroneous answer.

4.2.4. Refinement via Storage-Layout Relations

After the switch from the denotational framework to the operational one, many aspects of the implementations had still to be verified. These aspects were responsible for linearizing code, for implementing the primitive data structures of the languages, for representing the stack and environment structures, for introducing garbage collection in Scheme, and for omitting type tags in PreScheme.

To justify these steps we repeatedly proved *state machine refinement* theorems in both the Scheme and the PreScheme verifications. State machine refinement allows us to substitute a more easily implemented state machine (a more "concrete" machine) in place of another (a more "abstract" machine), when we already know that the latter would be acceptable for some class of computations. To justify the replacement, we must show that the former computes the same ultimate answer value as the latter when started in a corresponding initial state. As described in more detail in [13, Section 4.2], we have developed the technique of *storage layout relations* [32] to prove these refinement theorems. A storage layout relation is a relation of correspondence between the states of the concrete and abstract machines such that:

1. A concrete initial state corresponds to each abstract initial state;

2. As computation proceeds, the correspondence is maintained;

3. If either of a pair of corresponding states is a halt state, then so is the other, and moreover the two states deliver the same value as computational answer.

A storage layout relation thus establishes that the concrete machine simulates the abstract machine in a sense. The advantage of the storage layout relation as a method of proving refinement is that in establishing clause 2, which generally requires the bulk of the effort, only states linked immediately by the transition relation need be compared.

A Simple, Inductive Form of Definition. In many cases, the machine states (and their components) may be regarded in a natural way as syntactic objects, i.e. as terms in a language defined by a BNF. Then a principle of inductive definition is valid for these objects. A property of the terms (representing states and state components) may be defined in terms of the form of the term together with the value of the property for its immediate subterms. Storage layout relations defined in this way are suited for many purposes; all but two of the storage layout relations used were of this kind.

This approach to defining a storage layout relation is used when two conditions are met:

- For atomic terms in the two state machines, we can tell immediately whether they represent the same abstract computational object; and
- For compound terms, we can tell whether they represent the same abstract computational object if we know their structure and whether this correspondence holds of their immediate subterms respectively.

This approach can be made to work when the objects are not finite, but are rational trees [5], as in the proof of the linear-data machine in the PreScheme compiler [21, Section 6]. However, on this approach, when a stored object contains a reference to another store location then generally the pointers must point to the *same* location. Thus for instance, suppose a cell c stores a pair that contains a pointer to a location ℓ in which another pair is stored. Then another cell c' could correspond to c only if it too stores a pair with a pointer to ℓ. This inductive form of definition is not suitable when one wants to allow the second pointer to reference ℓ', so long as ℓ' contains appropriate objects.

This is the situation in justifying garbage collection: the structural similarity of the stored objects should suffice, without their needing to be stored in the same locations. In these cases, where the instructions of the machine create and modify structures with cyclic sequences of references, a different form of definition must be used.

Cyclic Structures: A Second-order Form of Definition. What is involved in proving a result like the correctness of garbage collection? In our formulation, there is an abstract machine with an unbounded store, in which objects are never relocated, and there is a concrete machine with a store consisting of two heaps. A copying garbage collector periodically relocates objects in these heaps.

In essence, we would like to "guess" a one-to-one correlation $\ell^C \sim \ell^A$ between a concrete store location ℓ^C and the abstract store location ℓ^A that it represents, if any. Thus it will relate some of the locations in the active heap of the concrete machine and some of the abstract store locations, presumably including all the computationally live ones. We can extend the location correlation $\sim$ by an explicit definition to a correspondence $\simeq$ between all terms representing states or their constituents. If we guessed $\sim$ correctly, then corresponding state components should:

1. Contain equal values, when either contain an concrete atomic value;
2. Reference correlated locations, when either contains a pointer;
3. Be built from corresponding subterms using the same constructor, otherwise.

This train of thought suggests defining a storage layout relation using an existential quantifier over location correlations. A concrete machine state Σ^C represents an abstract machine state Σ^A, which we will write $\Sigma^C \cong \Sigma^A$, if there exists a location correlation $\sim$ which extends to a correspondence $\simeq$ such that:

- $\simeq$ has properties 1–3; and
- $\Sigma^C \simeq \Sigma^A$.

Since $\sim$ is a relation between locations, the existential quantifier here is a second order quantifier.

There are now two main theorems that must be proved to justify a garbage collected implementation.

- First, that $\cong$ is a storage layout relation between the abstract and concrete machines. This establishes that garbage collection is a valid implementation strategy from the abstract machine at all. By contrast, it is also possible that a machine's computations depend directly on the particular locations in which data objects have been stored. This is certainly the case if it is possible to treat a pointer as an integer and apply arithmetic operations to it.
- Second, that a particular garbage collection algorithm preserves $\cong$. If the garbage collector is a function G on concrete states, then for all abstract states Σ^A and concrete states Σ^C, we must show

  $$\Sigma^C \cong \Sigma^A \Rightarrow G(\Sigma^C) \cong \Sigma^A.$$

  This amounts to showing that the garbage collector simply replaces one acceptable $\sim$ with another. It establishes that the particular algorithm G is a valid way to implement garbage collection.

We have used this form of definition not only for justifying garbage collection [13, Section 6.2], but also in one other proof in VLISP where circular structures were at issue.

5. Styles of Semantics

VLISP has extensively used two semantic techniques. The first of these is the denotational approach, in which the meaning of each phrase is given by providing a denotation in a Scott domain [28, 26]. The official Scheme semantics as presented in [15, Appendix A] is of this kind.

Our operational alternative is to define a state machine, in which one state component is the code to be executed. The state machine transition function is defined by cases on the first instruction in that code. The semantics of a program is defined in terms of the computational answer ultimately produced if computation should ever terminate, when the machine is started in a state with that program as the code to be executed. The other components of the initial state are in effect parameters to the semantics.

5.1. Advantages of the Denotational Approach

The main general advantages cited for denotational semantics are:

- Its compositionality;
- 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 proved genuine in reasoning about our main compilation steps. 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 difficult challenge to verify the PreScheme Front End, for instance, using the operational style of the Piton compiler proof [20].

5.2. Disadvantages of the Denotational Approach

There are however some serious limitations to the denotational approach in the traditional form embodied in the official Scheme semantics.

At the prosaic end of this spectrum, there is of course the fact that in some cases one must reason about the "physical" properties of code, for instance in computing offsets for branches in linearizing conditional code. In this case, the meaning of the resulting code depends not only on the meanings of its syntactic constituents, but also on their *widths* in bytes. The denotational approach seems unnatural here.

5.2.1. The Scheme Denotational Semantics

Some of our less shallow objections concern the specifics of the official Scheme semantics, while others are more general objections to the approach as traditionally practiced.

Memory Exhaustion. The official semantics always tests whether the store is out of memory (fully allocated) before it allocates a new location. We removed these tests and introduced the assumption that the store is infinite, and we understand that this change is under consideration for the next (fifth) revision of the Report on Scheme. We made the change 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.

We have chosen to specify the finiteness of memory only at the very lowest level in the refinement of the virtual machine. 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.

Semantics of Constants. The official Scheme semantics contains a semantic function $\mathcal{K}$ which maps constants—certain expressions in the abstract syntax—to denotational values. But $\mathcal{K}$'s definition is "intentionally omitted." In some cases, its behavior seems straightforward: for instance, its behavior for numerals. It is however far from clear how one would define $\mathcal{K}$ for constants that require storage. We have had to treat $\mathcal{K}$ as a parameter to the semantics, and we needed to introduce axiomatic constraints governing it [13, Section 2.1].

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 defined in terms of 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, namely that it should start in a store that already contains many interesting expressed values. But 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 emits the initialization code before the application code generated from the user's program.

Input and Output. The official Scheme semantics is silent on I/O, and in the Scheme paper [13], we have followed it in ignoring I/O. In the PreScheme paper [21], however, we have included a placeholder in the semantics to model I/O, and more recent work on PreScheme has treated it in detail. Our view is that I/O is straightforward to handle, both in the operational and in the denotational framework. This is not to say that it is necessarily easy to prove that particular programs have the intended I/O behavior; rather, the language semantics is easy to characterize. Moreover, a compiler can be proved correct relative to the semantics with I/O without deep changes to the structure of the proofs as presented here.

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. As a consequence, many natural Scheme optimizations are difficult or impossible to verify, as they change the order in which procedure tags are allocated, or make it unnecessary to allocate some of the tags.

In our PreScheme semantics, by contrast, we have avoided tagging procedures. The verification of the PreScheme Front End would have been out of the question otherwise.

Artificial Signature for Expression Continuations. The semantics specifies the type for expression continuations as $\mathtt{E}^* \to \mathtt{C}$, which means that evaluating a Scheme expression may (in theory) pass a finite sequence of "return values," as well as a modified store, to its continuation.

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.

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. A considerable amount of effort [13, Sections 2.3–2.4] was devoted to developing a semantic theory that would justify the obvious intuitive implementation.

The situation here may change in the fifth revision to the Report on Scheme. A way for the programmer to construct procedures returning zero or several values is under consideration. The Scheme48/VLISP implementation approach would need to be modified in order to provide, for this new aspect of the language, an implementation more efficient than representing multiple values by building lists.

5.2.2. *The Denotational Method more Generally*

One more general objection to the denotational method as practiced in the tradition exemplified by Stoy [28] is that the denotational domains are too large. Although any actual implementation represents only recursive functions manipulating a countable class of data objects, the semantic domains are uncountable in the usual approaches to constructing them. Thus, in the most obvious sense, almost all of the denotational objects are not represented in an implementation. Moreover, it is difficult to characterize smaller domains axiomatically, as a class of objects all of which have some property, while ensuring the existence of the fixed points needed to model recursion.

As a consequence, the unrepresented domain elements mean that the denotational theory makes distinctions that cannot be observed in the behavior of the implementation. This phenomenon is called a failure of *full abstraction* [24, 14, 19]. The issue about multiple value returns just mentioned may be regarded as an instance, although we showed how to repair it.

Another familiar example, related to the issues discussed in [19], would be garbage collection. As others have observed, if one takes the official Scheme semantics in the most literal way, garbage collection is a demonstrably unacceptable implementation strategy. Many command continuations, which is to say elements of the domain $\mathtt{C} = \mathtt{S} \rightarrow \mathtt{A}$, are not invariant under garbage collection: for instance, the function which returns one value if location 64 has a cons cell in it and a different value if location 64 has a vector. Thus, we get a different computational result if the implementation garbage collects, and relocates a vector where a cons cell would otherwise have been. However, in any reasonable Scheme implementation, none of these "monstrous" objects is represented; all represented objects are in fact invariant under garbage collection.

Although it is conceivable that one might be able to repair this failure of full abstraction also, the graph isomorphism property that a garbage collector must establish is complex. It would be particularly difficult to state and to reason about, because the denotational theory does not offer logical resources such as the quantifiers that would normally be used in such a definition.

Thus the correctness of a garbage-collected implementation can be stated and proved, as far as we know, only within the operational framework.

A related issue is the difficulty of stating normalcy requirements in the denotational manner (again see [14]). 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. 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 type $\mathrm{Ide} \rightarrow \mathtt{L}$. Thus, the semantics makes no distinction between the environments that respect data structures in the heap and those that do not. In proving the faithfulness of the operational semantics [13, Section 3.2], we needed to introduce a comprehensive list of these "normalcy conditions."

It is not clear how to express the constraints interrelating different denotational domains—in this case, store and environment—in order to define the class of tuples that may reasonably be used together in the semantics.

There are also some intuitive requirements on the implementor that seem difficult or impossible to specify in the denotational style. For instance, the Scheme standard [15] requires that implementations of Scheme be properly tail recursive. But no plausible denotational definition of this specification has been proposed.

5.3. Advantages of the Operational Approach

We would give two main reasons why the operational approach may be uniquely appropriate in some cases.

Stating Special Requirements. Sometimes it is more convenient to use an operational style to specify particular primitives for manipulating data, such as input and output operations. A notorious puzzle is how to express denotationally the stipulation that tail recursion be handled correctly. Complexity constraints are also more naturally expressed in an operational framework.

For these reasons, 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, and in a more operational style, than others.

Induction on Computational Steps. Many of the operational proofs we carried out are essentially proofs by induction on the number of computational steps that a state machines takes. The faithfulness theorem [13, Theorem 6] is a paradigm case of this. It is difficult to simulate this kind of reasoning denotationally. The traditional approach of using inclusive predicates [28] is notoriously cumbersome.

6. Conclusion

The VLISP work has been divided between a scientific portion and an engineering portion. Some of the scientific issues have been summarized in Sections 4.2 and 5. However, to apply those methods effectively on a bulky and complicated program, it was necessary to control them using a number of engineering ideas.

We focused the verification on a mathematically tractable presentation of the algorithm—rather than on concrete program text, interpreted in accordance with the semantics of its programming language—so as to ensure that the rigorous analysis was concentrated on the level at which errors are most likely to occur and can be most effectively found.

We also set out with a sharply defined initial target. Scheme's well thought out semantic definition was a necessary condition, as was the carefully organized Scheme48 implementation that served us as a design model. We found it particularly important to intersperse the formal development with a number of prototype versions of the program; we also came to use a surprising number of separate refinement steps, connected by rigidly defined interfaces.

These are essentially engineering considerations, unlike for instance the choice of refinement methods or of different semantic styles for different portions of the work.

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 2.1.2). 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 5). The two methods can be combined in a single rigorous development using (for instance) the methods of 4.2.3.

Requirements at Several Levels Some requirements cannot be stated at the level of abstraction appropriate for others. For instance, it is not clear how to give a satisfactory specification of being properly tail-recursive in the top-level Scheme denotational semantics. It is more natural to represent these requirements lower down, in an operational framework.

Small Refinement Steps The VLISP proofs separate out a very large number of independent refinement steps. 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. 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.

Acknowledgements. We are deeply indebted to the other participants in the VLISP effort, namely William Farmer, Leonard Monk, and Vipin Swarup of MITRE, as well as Dino Oliva, now at the Oregon Graduate Institute.

We are also grateful to Northrup Fowler III of Rome Laboratory, as well as to Ronald Haggarty and Edward Lafferty of MITRE, for their commitment to enabling us to carry out this work.

Electronic versions of technical reports. The papers in this issue summarize, in updated and much improved form, a set of technical reports. More detailed information may be retrieved from the original technical reports [7, 8, 10, 11, 12, 22, 23, 25, 29, 30], which are available electronically from the Scheme Repository using Universal Resource Locator

`file://ftp.cs.indiana.edu/pub/scheme-repository/txt/vlisp/`

References

1. Joel F. Bartlett. Scheme->C: A portable Scheme-to-C compiler. WRL 89/1, Digital Equipment Corporation Western Research Laboratory, January 1989.
2. Robert S. Boyer and Yuan 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.
3. William 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.
4. William Clinger and Jonathan A. Rees (eds.). Revised^4 report on the algorithmic language Scheme. Technical Report CIS-TR-90-02, University of Oregon, 1990.
5. Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.
6. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, 1976.
7. William M. Farmer, Joshua D. Guttman, Leonard G. Monk, John D. Ramsdell, and Vipin Swarup. The faithfulness of the VLISP operational semantics. M 92B093, The MITRE Corporation, September 1992.
8. William M. Farmer, Joshua D. Guttman, Leonard G. Monk, John D. Ramsdell, and Vipin Swarup. The VLISP linker. M 92B095, The MITRE Corporation, September 1992.
9. David Gries. *The Science of Programming*. Springer-Verlag, 1981.
10. Joshua D. Guttman, Leonard G. Monk, William M. Farmer, John D. Ramsdell, and Vipin Swarup. The VLISP byte-code compiler. M 92B092, The MITRE Corporation, September 1992.
11. Joshua D. Guttman, Leonard G. Monk, William M. Farmer, John D. Ramsdell, and Vipin Swarup. The VLISP flattener. M 92B094, The MITRE Corporation, 1992.
12. Joshua D. Guttman, Leonard G. Monk, John D. Ramsdell, William M. Farmer, and Vipin Swarup. A guide to VLISP, a verified programming language implementation. M 92B091, The MITRE Corporation, September 1992.

13. Joshua D. Guttman, Vipin Swarup, and John D. Ramsdell. The VLISP verified Scheme system. *Lisp and Symbolic Computation*, 8(1/2):???–???, 1995.
14. Joseph Y. Halpern, Albert R. Meyer, and Boris A. Trakhtenbrot. The semantics of local storage, or what makes the free-list free? In *Conference Record of the Eleventh Annual ACM Symposium on the Principles of Programming Languages*, pages 245–257, 1984.
15. IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.
16. Richard A. Kelsey. Realistic compilation by program transformation. In *Conf. Rec. 16th Ann. ACM Symp. on Principles of Programming Languages*. ACM, 1989.
17. Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):???–???, 1994.
18. David Kranz, Richard A. Kelsey, Jonathan A. Rees, Paul Hudak, Jim Philbin, and Norman I. Adams. Orbit: An optimizing compiler for Scheme. *SIGPLAN Notices*, 21(7):219–233, June 1986. Proceedings of the '86 Symposium on Compiler Construction.
19. Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables: Preliminary report. In *Conference Record of the Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, pages 191–203, 1988.
20. J Strother Moore. Piton: A verified assembly-level language. Technical Report 22, Computational Logic, Inc., Austin, Texas, 1988.
21. Dino P. Oliva, John D. Ramsdell, and Mitchell Wand. The VLISP verified PreScheme compiler. *Lisp and Symbolic Computation*, 8(1/2):???–???, 1995.
22. 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.
23. Dino P. Oliva and Mitchell Wand. A verified runtime structure for pure PreScheme. Technical Report NU-CCS-92-27, Northeastern University College of Computer Science, September 1992.
24. Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–256, 1977.
25. John D. Ramsdell, William M. Farmer, Joshua D. Guttman, Leonard G. Monk, and Vipin Swarup. The VLISP PreScheme front end. M 92B098, The MITRE Corporation, September 1992.
26. David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Wm. C. Brown, Dubuque, IA, 1986.
27. Guy L. Steele. Rabbit: A compiler for Scheme. Technical Report 474, MIT AI Laboratory, 1978.
28. Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, 1977.
29. Vipin Swarup, William M. Farmer, Joshua D. Guttman, Leonard G. Monk, and John D. Ramsdell. The VLISP image builder. M 92B096, The MITRE Corporation, September 1992.
30. Vipin Swarup, William M. Farmer, Joshua D. Guttman, Leonard G. Monk, and John D. Ramsdell. The VLISP byte-code interpreter. M 92B097, The MITRE Corporation, September 1992.
31. Mitchell Wand. Semantics-directed machine architecture. In *Conf. Rec. 9th ACM Symp. on Principles of Prog. Lang.*, pages 234–241, 1982.
32. Mitchell Wand and Dino P. Oliva. Proving the correctness of storage representations. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 151–160, New York, 1992. ACM Press.