

Results and Conclusions from the VLISP Verification*

JOSHUA GUTTMAN

(*guttman@mitre.org*)

*The MITRE Corporation
202 Burlington Road
Bedford, MA 01730-1420*

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 supplements two more detailed studies on VLISP [6, 14]. It summarizes the basic techniques that were used repeatedly throughout the effort. It presents scientific conclusions about the applicability of these techniques as well as engineering conclusions about the crucial choices that allowed the verification to succeed.

Contents

1	Introduction	2
1.1	Refinement Layers	3
1.2	The Emphasis on Rigor	3
1.2.1	Rigorous, but Not Completely Formal Specifications	5
1.2.2	Algorithm-Level Verification.	5
1.3	“Prototype but Verify”	7
2	The Main Techniques	8

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

2.1	Transformational Compilation	9
2.2	Wand-Clinger style Compiler Proof	10
2.3	Faithfulness of an Operational Semantics	11
2.4	Refinement via Storage-Layout Relations	13
2.4.1	A Simple, Inductive Form of Definition	14
2.4.2	Cyclic Structures: A Second-order Form of Definition	15
3	Styles of Semantics	16
3.1	Advantages of the Denotational Approach	16
3.2	Disadvantages of the Denotational Approach	17
3.2.1	The Scheme Denotational Semantics	17
3.2.2	The Denotational Method more Generally	19
3.3	Advantages of the Operational Approach	21
4	Conclusion	22

1. Introduction

The VLISP project showed how to produce a comprehensively verified implementation for a programming language, namely Scheme. The techniques used are described in detail in [6, 14]. In this paper we present a brief overview of:

- The crucial choices that allowed the verification to succeed;
- The small collection of basic techniques that were used repeatedly in the verification process;
- The strengths and weaknesses of operational and denotational styles in various stages of the verification.

This introduction will focus on the underlying choices that made the work feasible, while Sections 2 and 3 will respectively summarize the techniques and appraise the utility of the semantic styles.

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 [8, Appendix A] has been stable for several years, and it has already served as the basis for substantial work, for instance [2]. 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.

Indeed, we also found it a great advantage to start from a well thought out design, as embodied in Scheme48 [9].

1.1. Refinement Layers

The Vlisp implementation 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 soluble verification task.

They are grouped 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 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 [9]. 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.

The verification of each main component is carried out in several steps, and the programs have been organized to respect this division. For instance, the verification of the compiler from Scheme to byte code consists of six separate steps. Each of the proofs connects one programming language equipped with a semantics with another, although in some steps only the language or 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 lower interface meets its semantic definition.

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

1.2. 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 prevents 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 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.

1.2.1. 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 sublanguage 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.

1.2.2. 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 [1].

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, as the level of the verification goes down, the mass of detail increases, 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.

In the case of the Scheme programs that form both the Scheme-to-byte-code compiler and the PreScheme compiler, it would hardly have been tractable to verify the program text directly from the official Scheme semantics. The reason for this is that the Scheme semantics must accurately represent the behavior of all Scheme programs. Many of the compiler phases are implemented by *applicative* Scheme programs. Thus, their behavior may be reliably modeled using a much simpler execution model. To superimpose all the overhead of the official semantics on what was a reasonably complex verification in any case, and with very little increase in the degree of insight to be expected, would be little more than an exercise in mortification of the intellect. By contrast 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. We had tried to make our implementation more efficient by a shortcut not reflected in the specification. When a portion of memory was allocated for a new Scheme pair or vector, we omitted the memory operations to ensure that the newly allocated memory location contained a distinctive value *empty*. We imagined that the program would initialize the memory to the correct useful value before anything could go wrong. In this we departed from our specification, which assumed that newly allocated memory contained *empty*. 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 pointers.¹

¹We later specified and verified a version of allocation and garbage collection which does not require the initialization to *empty*.

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 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 expression 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 sequence of rule applications whatever will be safe.

1.3. “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 [4, 5] 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 good 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 is developed.

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 [6, 14]. 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:

- 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 [14, 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 [17, 11, 10].

- A syntax-directed compiler in the Wand-Clinger style [19, 2]. 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 also establishes an equality (or a strong equivalence) between the denotations of its input and output.
- 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 [20, 6, Section 3.3]. 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 the 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.

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 $\mathcal{P}[[P]] \neq \perp_A$, 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 PreScheme 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 not on the store at the time the procedure was 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 the correctness of the heuristics embodied in the control structure.

2.2. Wand-Clinger style Compiler Proof

The Wand-Clinger style [19, 2] 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 recognizes 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 [19] 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 [14, Theorem 4] takes just this form. Although the corresponding theorem for the Scheme implementation [6, Theorem 10] 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 hypotheses, which are based on [2], spell out how this data is to be used.

2.3. Faithfulness 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 [2] in giving each instruction a denotation that may be regarded as acting on four registers together with a store; in most cases, the denotation

$ \begin{aligned} & \text{make_cont} : \mathbb{P} \rightarrow \mathbb{N} \rightarrow \mathbb{P} \rightarrow \mathbb{P} \\ & \text{make_cont} = \lambda \pi^1 \nu \pi . \lambda \epsilon \epsilon^* \rho_R \psi . \# \epsilon^* = \nu \rightarrow \pi \epsilon \langle \rangle \rho_R (\lambda \epsilon . \pi^1 \epsilon \epsilon^* \rho_R \psi), \\ & \qquad \qquad \qquad \text{wrong "bad stack"} \end{aligned} $
<p>Rule 1: Make Continuation</p> <p>Domain conditions:</p> $b = \langle \text{make-cont } b_1 \# a \rangle :: b_2$ <p>Changes:</p> $b' = b_2; \quad a' = \langle \rangle; \quad k' = \langle \text{CONT } t \ b_1 \ a \ u \ k \rangle$

Table 1: Denotational and operational treatments of `make-cont`

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 1.

In the denotational clause, π^1 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 expected 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. 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. 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 $\langle \text{CONT } t \ b_1 \ a \ u \ k \rangle$ codes the same information as the denotational continuation $(\lambda \epsilon . \pi^1 \epsilon \epsilon^* \rho_R \psi)$, and the two treatments of the `return` instruction,

which may eventually invoke the continuation, arranges to treat them in parallel ways.

Form of the Faithfulness Proof. The main idea of the Scheme faithfulness proof is to associate to each state a denotation in the domain \mathbf{A} of answers. If s is an initial state, then the denotation of s agrees with the denotational value of the program that it contains, when applied to the initial parameters used in the compiler proof. For a halt state s , the denotation is the number contained in its value register (accumulator), and $\perp_{\mathbf{A}}$ if the value register does not contain a number. This choice is compatible with the answer function (or “initial continuation”) used in the 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 preceding 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 *ans* function 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 6 of [14] 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.

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 for Scheme, and for omitting type tags for PreScheme.

To justify these steps we proved *state machine refinement* theorems. 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 [6, Section 3.3], we have developed the technique of *storage layout relations* [20] 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.

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.

2.4.1. 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:

- 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.

In particular, on this approach, when a stored object contains a reference to another store location—for instance, when a cell storing a pair contains a pointer, say to a location in which another pair is stored—then generally the pointers must point to the *same* location.

This approach can also be made to work when the objects are not finite, but are rational trees [3], as in the proof of the linera-data machine in the PreScheme compiler [14, Section 6]. However, there are also cases, such

as garbage collection, when 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.

2.4.2. *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 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 then:

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 s^C represents an abstract machine state s^A , which we will write $s^C \cong s^A$, if there exists a location correlation \sim which extends to a correspondence \simeq such that:

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

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 s^A and concrete states s^C , we must show

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

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 [6, Section 6], but also in one other proof in VLISP where circular structures were at issue.

3. 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 [18, 16]. The official Scheme semantics as presented in [8, 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.

3.1. Advantages of the Denotational Approach

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;

Draft of September 29, 1993

- 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 very difficult challenge to verify the PreScheme Front End, for instance, using the operational style of the Piton compiler proof [13].

3.2. 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.

3.2.1. 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 the next (fifth) revision of the Report on Scheme will incorporate this change. 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} is simply characterized as “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 [6, Section 2.2.1].

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 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, 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.

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 $E^* \rightarrow C$, 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.

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 [6, Sections 2.2.3–2.3.2] was devoted to developing a semantic theory that would justify the obvious intuitive implementation.

The situation here will change in the fifth revision to the Report on Scheme. It specifies a way that the programmer can construct procedures returning zero or several values. The Scheme48/VLISP implementation approach would need to be modified in order to comply with this new aspect of the language.

3.2.2. *The Denotational Method more Generally*

One more general objection to the denotational method as practiced in the tradition exemplified by Stoy [18] 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* [15, 7, 12]. 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 [12], 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 $\mathcal{C} = \mathcal{S} \rightarrow \mathcal{A}$, 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 more difficult to state and to reason about within the denotational semantics than the property of being single valued. This is particularly the case 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. 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 $\text{Ide} \rightarrow \text{L}$. 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 [6, 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 implementer that seem difficult or impossible to specify in the denotational style. For instance, the Scheme standard [8] requires that implementations of Scheme be properly tail recursive. But no plausible denotational definition of this specification has been proposed.

3.3. Advantages of the Operational Approach

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

Stating Requirements for Certain Primitives Sometimes it is more convenient to use an operational style to specify particular primitives for manipulating data, such as input and output operations.

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 too crude a criterion of meaning when I/O behavior is concerned. In particular, it would take substantial surgery to transform a denotational definition like Scheme’s into a form that could express requirements that depend on the exact interleaved sequence of its input and output events.

Moreover, a semantics of the “final computational answer,” which is the normal style of denotational definition, 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 that consumes plaintext characters and key material may be more useful if it 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, 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. Theorem 19 of [6], which justifies the use of storage layout relations to prove refinement, is a paradigm case of this,² as is the faithfulness theorem, Theorem 13. It is difficult to simulate this kind of reasoning denotationally. The traditional approach of using inclusive predicates [18] is notoriously cumbersome.

4. 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 2 and 3. 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 1.2.2). 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

²The induction is hidden in the proof of Lemma 18.

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 1.3). 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 3). The two methods can be combined in a single rigorous development using (for instance) the methods of 2.3.

Requirements at Several Levels Some system-level 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 input-output behavior in the top-level Scheme denotational semantics. It is more natural to represent them 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.

References

1. 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.
2. Will Clinger. The Scheme 311 compiler: An exercise in denotational semantics. In *1984 ACM Symposium on Lisp and Functional Program-*

- ming*, pages 356–364, New York, August 1984. The Association for Computing Machinery, Inc.
3. Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.
 4. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, 1976.
 5. D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
 6. J. D. Guttman and V. Swarup. A verified implementation of scheme. Submitted to LASC.
 7. J. 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.
 8. IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.
 9. Richard Kelsey and Jonathan Rees. Scheme48 progress report. Manuscript in preparation, 1992.
 10. Richard A. Kelsey. Realistic compilation by program transformation. In *Conf. Rec. 16th Ann. ACM Symp. on Principles of Programming Languages*. ACM, 1989.
 11. 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.
 12. 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.
 13. J S Moore. Piton: A verified assembly-level language. Technical Report 22, Computational Logic, Inc., Austin, Texas, 1988.
 14. D. P. Oliva, J. D. Ramsdell, and M. Wand. A verified compiler for VLISP prescheme. Submitted to LASC.

15. G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–256, 1977.
16. David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Wm. C. Brown, Dubuque, IA, 1986.
17. Guy L. Steele. Rabbit: A compiler for Scheme. Technical Report 474, MIT AI Laboratory, 1978.
18. Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, 1977.
19. M. Wand. Semantics-directed machine architecture. In *Conf. Rec. 9th ACM Symp. on Principles of Prog. Lang.*, pages 234–241, 1982.
20. 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.