LISP AND SYMBOLIC COMPUTATION: An International Journal, ?, ??–??, 1993
© 1993 Kluwer Academic Publishers – Manufactured in The Netherlands

VLISP: A Verified Implementation of Scheme*

JOSHUA GUTTMAN (*guttman@mitre.org*)

VIPIN SWARUP (*swarup@mitre.org*)

JOHN RAMSDELL (*ramsdell@mitre.org*)

Abstract. VLISP has produced a rigorously verified compiler from Scheme to byte codes, and a verified interpreter for the resulting byte codes. The official denotational semantics for Scheme provides the main criterion of correctness. The Wand-Clinger technique was used to prove correctness of the main compiler step. Then a state machine operational semantics is proved to be faithful to the denotational semantics. The remainder of the implementation is verified by a succession of state machine refinement proofs. These include proofs that garbage collection is a sound implementation strategy, and that a particular garbage collection algorithm is correct.

Contents

1 Introduction **2**
- 1.1 What is Scheme? 4
- 1.2 Structure of the VLISP Implementation 4
  - 1.2.1 The Extended Byte Code Compiler 4
  - 1.2.2 The Virtual Machine 6

2 The Byte-Code Compiler **8**
- 2.1 The Scheme Semantics 12
- 2.2 The Byte Code Semantics 16
- 2.3 Multiple Values in the Scheme Semantics 19
- 2.4 Faithfulness of the Alternative Semantics 24
- 2.5 Compiler Correctness for Single-valued Semantics 26

3 Operational Semantics: Faithfulness and Refinement **30**
- 3.1 Operational Semantics 30

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

3.2 Proving Faithfulness 32
3.2.1 Denotational Semantics of the Tabular Byte Code 32
3.2.2 Operational Semantics of the Tabular Byte Code 32
3.2.3 Faithfulness: Form of the Proof 36
3.3 Refinement and Storage Layout Relations 40

4 The Flattener 43
4.1 The Tabular Byte Code in More Detail 44
4.2 Flattened Byte Code Syntax 45
4.3 Flattened Byte Code Operational Semantics 46
4.4 The Flattener 47
4.5 Code Correspondence 47
4.6 State Correspondence 51

5 State Observers and Mutators 53

6 Garbage Collection 57
6.1 State Similarity 59
6.2 Legitimacy of Garbage Collection 60
6.3 The Correctness of a Copying Garbage Collector 64

7 The Finite Stored Program Machine 72

8 The VLISP Scheme Implementation 72
8.1 The VLISP Bootstrap Process 72
8.2 VLISP Virtual Machine Performance 74
8.3 Unverified Aspects of VLISP 75
8.4 Conclusion. 76

1. Introduction

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

An implementation for a programming language consists of the ensemble of software required to make a program written in that language execute on a digital computer. A programming implementation may be a compiler, which translates programs in the given language to programs in a lower level language; or it may be an interpreter, which is a program that executes the higher level language directly; or it may be a combination of the two. The VLISP implementation, being modeled on Scheme48 [13], is of this mixed

type. It consists of:

- A simple compiler that translates programs in the source language to target programs in an intermediate level byte code language;
- An interpreter (written in a systems-programming oriented dialect called PreScheme [13]) to execute the resulting byte code programs;
- A second compiler to translate the PreScheme code of the interpreter into assembly language code for commercial workstations (either a Sun 3 with an MC68020 CPU or a Sun 4 with a SPARC processor).

This paper focuses on the first two items, which make up the implementation of Scheme. The third item, which implements PreScheme, is discussed in the accompanying paper [16]. A third paper [11] discusses results and conclusions of the effort as a whole. The present paper is intended to serve has two main purposes:

- To provide a detailed account of the techniques used in the VLISP Scheme verification; and
- To show the decomposition of layers that made the verification process tractable, that is, to display the "formal methods engineering" decisions we have made.

The VLISP effort has focused on algorithm-level rigorous verification. By this we mean to emphasize two things.

First, the proofs we have produced are about algorithms, rather than being about concrete program text. Reasoning about the concrete program text (under its formal semantics) is considerably more cumbersome. Far from providing deeper insight, in our opinion, the mass of detail involved frequently obscures the essential reasons why the program is correct. Naturally, the actual program text must be carefully constructed so that it is a patently faithful embodiment of the algorithm.

Second, our proofs are rigorous mathematical arguments, i.e., what most mathematicians mean by proofs. We have not produced derivations in a particular formal deductive system (which might be assisted by mechanical proof tools). Formal derivations would, again in our opinion, provide less real insight than mathematical proofs, unless the derivations were accompanied by carefully written, humanly understandable mathematical proofs. We return to this point in [11].

1.1. What is Scheme?

The Scheme programming language, "an UnCommon Lisp," is defined by the language Report (currently in its fourth revision [3]), and by an IEEE standard [12]. We have taken [12] 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 abstract 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 [9], as its starting point. We have therefore concentrated on verifying those aspects of the language that it characterizes; some other aspects have only been introduced into our specifications at lower levels in the process of specifying and verifying our implementation.

1.2. Structure of the VLISP Implementation

The VLISP Scheme implementation derives from Scheme48 [13], and is thus a byte-coded implementation. That is, a compiler phase transforms Scheme source programs into an intermediate level language; the resulting target programs are executed by an interpreter, which we refer to as the "virtual machine" VM. The compiler phase is coded in Scheme, while the interpreter is coded in PreScheme.

1.2.1. The Extended Byte Code Compiler

The extended compiler acts on source programs in the Scheme language, and ultimately produces programs in a linearized byte code that we call "stored byte code" [20]. Its input programs consist of abstract syntax trees in a format only slightly different from the format used in the Scheme Standard [12]. The stored byte code output is in a binary form suitable for execution in the VM.

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

- The **byte code compiler** [9] (properly so called, as distinguished from the extended compiler, of which it forms the first stage) rewrites the Scheme source code into a tree-structured byte code based on Clinger's target language [2]. We will refer to this tree-structured byte code as the **basic byte code** or BBC. The BBC makes the flow of control in the Scheme program explicit, including procedure call and return, the order of argument evaluation, and the operations involved in building environment structures for local variables. BBC uses nested subtrees to represent constants, nested lambda expressions, conditionals, and procedure calls (when not in tail recursive position) in the source code. Successive phases of the compiler eliminate these nested structures in favor of linear layouts.
- The **tabulator** [9] transforms BBC procedures into templates in a **tabular byte code** or TBC. A template consists of some code, together with a table listing global variables, constants, and embedded templates (representing locally defined procedures). Instructions using these objects reference them by a numerical index.
- The **flattener** [10] transforms code in the TBC into a form that is linear rather than tree-structured. We call the output language of the linearizer the **flattened byte code** (FBC).
- The next stage of the extended compiler is the **linker** [6]. Its job is to transform a collection of (nested) constants and templates in the FBC into a single linear structure representing the full program.
- Finally the **image builder** [21] writes out the image to a file. The image builder is responsible for transforming the syntax of the linked code into a succession of bytes, with inetrnal references represented by 32-bit pointers.

The verification of the byte code compiler is closely based on Clinger's, although some work, described in Section 2.5, had to be done to massage the official Scheme denotational semantics into a suitable form. The theorem establishes that a Scheme program and its compiled image compute that same computational answer, when started with appropriate initial values. The proof that the tabulator leaves the denotation of a program unchanged is straightforward.

The proofs of the remaining phases are based on an operational semantics, using state machines. Thus, there must be a proof that the operational semantics is faithful to the denotational one. For our purposes, it

is enough to prove what may be regarded as a "partial correctness" assertion, namely that when the operational semantics predicts a computational answer (rather than non-termination, or an error condition), then the denotational semantics predicts the same answer. For this proof we devised a simple and easily mechanized method, described in Section 3.2.

We consider this approach far more tractable than those that have been devised to prove total correctness [14]. Total correctness would also ensure the converse: whenever the denotational semantics predicts a non-bottom answer, the operational semantics delivers that answer. However, the Scheme semantics predicts more answers than any implementation can actually deliver, simply because any implementation will have finite bounds, for instance on the height of the stack, and the semantics does not fully express this.

For the succeeding stages, we prove correctness by showing that one state machine is faithfully *refined* by another. As refinement is defined in Section 3.3, when the relation holds between an abstract state machine and a more concrete one, the latter can be used to compute the same results as the former. In practice, in order to prove refinement, we actually prove something more specific, namely that there is a *storage layout relation* between the machines. A storage layout relation (also as defined in Section 3.3) simplifies the task of proving refinement, because the comparison between state machines involves only states adjacent to one another under the transition relation. We have used this method repeatedly, defining the needed storage layout relations using various different techniques. The flattener (Section 4) provides a clean example of the method.

The other phases of the extended byte code compiler, namely the tabulator, linker and image builder, do not raise essentially new issues, so that we will focus our attention in this paper on the Clinger-style compiler phase, on the faithfulness proof, and on the flattener.

1.2.2. The Virtual Machine

Unlike the compiler phase, which consists of a sequence of essentially separate algorithms, the interpreter comprises a single program. However, the interpreter design is built up by a succession of refinement steps.

These refinement steps are designed to display the successive implementation decisions in a sort of rational reconstruction of the Scheme48 virtual machine. At each stage it must be shown that the new implementation decisions provide a state machine faithful to the behavior of the previous machine. There are three steps, the first two of which are strong refinements, while the last is a weak refinement in the "partial correctness" sense mentioned above. The three refinements interconnect the following four machines, of which the first provides the operational semantics for the out-

put of the extended compiler, and of which the last is implemented directly using the PreScheme primitives.

- The **stored program machine**, or SPM, which executes the byte code images produced by the image builder;
- The **microcoded stored program machine**, or MSPM, which defines the steps of the SPM as compositions of atomic *state observer* and *state modifier* operations;
- The **garbage-collected state machine**, or GCSM, which may nondeterministically stop and copy its live data to a fresh heap. The heaps of this state machine (and its predecessors) are of unbounded size, and each location may store an unbounded integer or pointer (among other objects);
- The **finite garbage-collected state machine**, or FGCSM, which has heaps of bounded size and has a bound on the width of an integer or pointer.

Naturally the FGCSM can not carry out all of the computations of the GCSM; it is correct only in the sense that when it does compute an answer, that is the same value that the GCSM would have computed.

These successive abstract machines are all visible in the code of the VLISP VM. The primitive notions of each machine are shown as procedures or constants, which are in turn implemented in terms of the primitives of the next machine (or primitives of PreScheme, in the case of the last). Thus, each state machine in the sequence of refinements corresponds closely to the program text above a particular abstraction barrier.

There is, however, one complication in matching the program to the specification. It is due to the fact that VLISP PreScheme compiles to a language in which all procedure calls are in tail recursive position. The Front End must substitute the code for a called procedure in line wherever there is a call in the source that is not tail recursive. There are several instructions in the SPM that cause memory to be allocated, and each of them, when coded in the obvious way, contains a call to an allocation routine, which in turn calls the garbage collector if the allocation would exceed a bound. These calls are not tail recursive, as the original procedure must normally do some initializations within the new memory object after it has been allocated. Rather than have the code for the garbage collector duplicated in line at each of the original call sites, we have programmed the calls to the allocation procedure and garbage collector in a partial continuation passing style. Because this programming decision affects only

a small number of call sites, and only one subroutine, the implementation has lost very little lucidity as a consequence of this programming approach.

Of the three main correctness assertions for the virtual machine, the first and third need relatively little scrutiny, provided in the short Sections 5 and 7.

The interesting step (see Section 6) is to show that the garbage collected state machine refines its predecessor, the MSPM. To prove this we provide a storage layout relation $\sim$ between the MSPM and the GCSM. The relation $\sim$ expresses that the stores of the two states, when regarded as directed graphs and restricted to the portion reachable from abstract machine registers, are isomorphic. There are then conceptually two assertions to be proved:

1. When each of the two machines makes an ordinary (non-GC) transition, related states evolve to related states; and
2. When $s_{\mathrm{MSPM}} \sim s_{\mathrm{GCSM}}$ and garbage collection produces s'_{GCSM} from s_{GCSM}, then $s_{\mathrm{MSPM}} \sim s'_{\mathrm{GCSM}}$.

Of these the former establishes that garbage collection is a valid implementation strategy for the MSPM, i.e., that the eventual result of the computation depends only the isomorphism class of the reachable part of the store. The second assertion shows that the particular garbage collection algorithm used is a correct algorithm.

2. The Byte-Code Compiler

The VLISP implementation is a byte-coded one, and the task of the compiler is to produce byte codes from a Scheme source program. The denotation of the resulting byte code program is equivalent to the denotation of the Scheme source program in a sense made precise in Section 2.5.

The byte code compiler itself is based on algorithms used by Clinger [2] and Kelsey and Rees [13]. Its purpose is to analyze the procedural structure of the source code. It distinguishes tail-recursive procedure calls from non-tail-recursive ones; it puts conditional expressions into a more explicit procedural form; and it calculates $\langle over, back \rangle$ lexical addresses for lexical variables, which it distinguishes from global variables.

The algorithm itself is a syntax-directed recursive descent, based on the abstract syntax shown in Table 1. We regard this BNF as determining a set of tree-like abstract syntax objects, rather than as recognising a set of flat character strings. The tree-like abstract syntax objects are represented by nested sequences. Thus for instance, a concrete program of the form `(lambda (x . y)` *body*) has as its abstract syntax:

$$
\begin{array}{rcl}
e, \mathrm{E} & ::= & i \mid c_{sq} \mid \langle \mathtt{quote}\ c \rangle \mid e^{+} \mid \langle \mathtt{begin} \rangle^\frown e^{+} \\
& \mid & \langle \mathtt{lambda}\ i^{*}\ e \rangle \mid \langle \mathtt{dotted_lambda}\ i^{+}\ e \rangle \\
& \mid & \langle \mathtt{if}\ e_1\ e_2\ e_3 \rangle \mid \langle \mathtt{if}\ e_1\ e_2 \rangle \mid \langle \mathtt{set!}\ i\ e \rangle \\
c, \mathrm{K} & ::= & c_{pr} \mid \text{strings} \mid \text{lists, dotted lists, and vectors of } c \\
c_{pr} & ::= & \text{numbers, booleans, characters, symbols and nil} \\
c_{sq} & ::= & \text{numbers, booleans, characters and strings} \\
i, \mathrm{I} & ::= & \text{identifiers (variables)} \\
\Gamma & ::= & e \qquad \text{(commands)}
\end{array}
$$

Table 1: Scheme Abstract Syntax

$$\langle \mathtt{dotted_lambda}\ \langle \mathtt{x\ y} \rangle\ e \rangle$$

where e represents the abstract syntax of the body. This abstract syntax is only slightly more abstract than the result of the normal Scheme reader, when lists are regarded as representing mathematical sequences. Dotted lambda forms are the main case in which the abstract syntax differs; we find this form slightly more explicit.

The compiler's target language is the byte code language abstract syntax defined in Table 2. Expressions of the Basic Byte Code language (BBC) are nested sequences constructed according to the BNF grammar given in Table 2 from natural numbers, Scheme identifiers, Scheme constants, and the keywords shown. We will use n-like variables for natural numbers, i-like variables for identifiers, and c-like variables for constants. Similarly for the classes defined by the grammar, with

t for (BBC) *templates*,
b for (BBC) *closed instruction lists* (conjecturally < Eng. *block*),
z for (BBC) *neutral instructions*,
y for (BBC) *open instruction lists*,
w for (BBC) *(general) instruction lists*, and
m for (BBC) *(machine) instructions.*

This syntax is a little more complex than might have been expected. The distinction between open and closed instruction lists captures a pattern in the code produced by the compiler. When a conditional expression is the last Scheme expression in the block being compiled, the compiler can ar-

t	$::=$	$\langle \texttt{lap}\ c\ b \rangle$
b	$::=$	$\langle\langle \texttt{return} \rangle\rangle \mid \langle\langle \texttt{call}\ n \rangle\rangle \mid \langle\langle \texttt{unless-false}\ b_1\ b_2 \rangle\rangle$
	$\mid$	$\langle \texttt{make-cont}\ b_1\ n \rangle :: b_2 \mid z :: b_1$
z	$::=$	$\langle \texttt{unless-false}\ y_1\ y_2 \rangle$
	$\mid$	$\langle \texttt{literal}\ c \rangle \mid \langle \texttt{closure}\ t \rangle$
	$\mid$	$\langle \texttt{global}\ i \rangle \mid \langle \texttt{local}\ n_1\ n_2 \rangle$
	$\mid$	$\langle \texttt{set-global!}\ i \rangle \mid \langle \texttt{set-local!}\ n_1\ n_2 \rangle$
	$\mid$	$\langle \texttt{push} \rangle \mid \langle \texttt{make-env}\ n \rangle$
	$\mid$	$\langle \texttt{make-rest-list}\ n \rangle \mid \langle \texttt{unspecified} \rangle$
	$\mid$	$\langle \texttt{checkargs=}\ n \rangle \mid \langle \texttt{checkargs>=}\ n \rangle$
	$\mid$	$\langle i \rangle$
y	$::=$	$\langle \texttt{make-cont}\ y_1\ n \rangle :: b \mid \langle \texttt{make-cont}\ \langle\rangle\ n \rangle :: b \mid z :: y_1 \mid \langle z \rangle$
w	$::=$	$b \mid y$
m	$::=$	$z \mid \langle \texttt{return} \rangle \mid \langle \texttt{call}\ n \rangle \mid \langle \texttt{unless-false}\ b_1\ b_2 \rangle$
	$\mid$	$\langle \texttt{make-cont}\ w_1\ n \rangle$

Table 2: Grammar for the Basic Byte Code

range that the code of each branch will contain a `call` or `return` instruction, so that there need be no common code to be executed after either branch has completed. On the other hand, when the conditional expression is not the last Scheme expression in the block, the compiler arranges that neither branch is terminated by a `call` or `return`. Thus, execution "drops into" the following code when the selected branch is completed.

After taking cases based on the abstract syntax class of the expression being compiled, the algorithm combines the code produced by its recursive calls with some additional instructions. The main dispatch and one typical case are shown in slightly simplified form in Figure 1. The parameters are:

- The expression being compiled;
- A "compile-time environment," which associates identifiers serving as lexical variables with their lexical addresses. It is enriched when the algorithm traverses a `lambda`;
- The "after code" to be attached at the end of the code generated from the current expression; it represents the result of a recursive call.

The Scheme code implementing the algorithm can be safely regarded as a presentation of a mathematical algorithm. It is purely applicative; it makes no use of `call/cc`; and it has the form of a primitive recursion

```
(define (comp exp cenv after)
  (cond ((id? exp)
         (compile-id exp cenv after))
        ((self-quoting-constant? exp)
         (compile-constant exp cenv after))
        ((pair? exp)
         (case (car exp)
           ((lambda) (compile-lambda exp cenv after))
           ((if) (compile-if exp cenv after))
           ...
           (else
            (compile-application exp cenv after))))
        (else (compiler-error ...))))

(define (compile-application exp cenv after)
  (if (return? after)
      (let* ((proc (car exp))
             (args (cdr exp))
             (nargs (length args)))
        (comp-args args cenv
                   (comp proc cenv
                         (instruction->code-sequence
                          (make-instruction 'call nargs)))))
      (compiler-prepend-instruction
       (make-instruction 'make-cont after)
       (compile-application exp cenv return-code))))

(define (comp-args args cenv after)
  (if (null? args)
      after
      (comp (car args) cenv
            (compiler-prepend-instruction
             (make-instruction 'push)
             (comp-args (cdr args) cenv after)))))
```

Figure 1: Compiler dispatch procedure, and case for procedure call.

on its expression argument, so that it is guaranteed to terminate. In the procedure to compile a procedure call, the call is tail recursive just in case the the "after code" consists of a bare return instruction. If the call is not tail recursive, then the resulting code will consist of a make-continuation instruction prepended in front of the corresponding tail recursive code. In the tail recursive case, it emits code to evaluate each argument, followed by a push instruction, and followed by code to evaluate the procedure. At the very end, a call instruction applies the procedure value to its evaluated arguments.

Syntactic correctness. If e is a Scheme expression, then $C(e, \rho_C, n, i, w)$ will refer to the result of calling the procedure `comp` on e together with the compile-time environment ρ_C and the after-code w (assumed to be a BBC instruction sequence). We will also use $C(e)$ to refer to the result of calling the procedure `comp` on e with the initial values of the other parameters.

Theorem 1 (Compiler syntactic correctness)

1. *$C(e)$ is a* BBC *template t;*
2. *$C(e, \rho_C, b_0)$ is a* BBC *closed instruction sequence b_1;*
3. *$C(e, \rho_C, y_0)$ is a* BBC *open instruction sequence y_1;*
4. *$C(e, \rho_C, \langle\rangle)$ is a* BBC *open instruction sequence y_1.*

The proof is a bulky but routine simultaneous structural induction on the Scheme syntax.

Since the algorithm is so straightforward a recursive descent, it is natural to use induction on the structure of expressions as a proof technique. Clinger [2] provides us with three suitable induction hypotheses which jointly establish the correctness of the algorithm. They appear below as Lemma 11. However, there is a striking difference between the official Scheme semantics and the simplified (and historically earlier) semantics that Clinger used in his paper, deriving from the fact that the official Scheme semantics makes provision for procedures that may return more than one value.

2.1. The Scheme Semantics

In this section we will briefly describe the official Scheme semantics, in the slightly modified form we have used in the VLISP project. Apart from a few cosmetic changes of notation, our version differs from the standard one in three ways. See [19] for a description of the underlying approach, and [9] for additional details.

$\langle \ldots \rangle$	finite sequence formation, commas optional
$\# s$	length of sequence s
$\langle x \ldots \rangle$	sequence s with $s(0) = x$
$\langle \ldots x \rangle$	sequence s with $s(\#s - 1) = x$
$\mathit{rev}\, s$	reverse of the sequence s
$s \frown t$	concatenation of sequences s and t
$s \dagger k$	drop the first k members of sequence s
$s \ddagger k$	the sequence of only the first k members of s
$p \to a, b$	if p then a else b
$\rho[x/i]$	the function which is the same as ρ except that it takes the value x at i
x in $\mathtt{D}$	injection of x into domain $\mathtt{D}$
$x \mid \mathtt{D}$	projection of x to domain $\mathtt{D}$
$x, y, \ldots : \mathtt{D}$	true if the type of $x, y, \ldots$ is a disjoint sum and $x, y, \ldots$ are injected from elements of $\mathtt{D}$

Table 3: Some Notation

- The domains have been made somewhat more concrete. In particular, $\mathtt{L}$ has been identified with $\mathtt{N}$, and $\mathtt{S}$ has been identified with $\mathtt{E}^*$. Note that this entails that, at the current level of modeling, memory is conceived as unbounded.
- We have removed tests from the semantics to check whether a new storage location can be allocated in $\mathtt{S}$. The official Scheme semantics uses conditionals that raise an "out of memory" error if there is no unallocated location; when memory is conceived as unbounded, this situation will not arise. Moreover, it does not seem that all situations in which a real Scheme interpreter can run out of memory are represented in the official semantics, for instance, overflowing the stack by too many nested, non-tail-recursive procedure calls. Thus, we have chosen to represent all memory exhaustion errors uniformly at a much lower level in the formal specification.
- The constraints on $\mathcal{K}$ given in the section on Semantics of Constants has been added. It was needed in our work on the faithfulness of an operational semantics for the BBC to its denotational semantics.

The domains used in the denotational semantics of Scheme are presented in Table 4. Note that exactly one "domain equation" is actually not an equation. It implicitly introduces an isomorphism between $\mathtt{E}$ and a disjoint

$\alpha \in \mathtt{L}$	$=$	$\mathtt{N}$	locations
$\rho \in \mathtt{U}$	$=$	$\mathrm{Ide} \to \mathtt{L}$	environments
$\nu \in \mathtt{N}$			natural numbers
$\mathtt{T}$	$=$	$\{\mathit{false},\ \mathit{true}\}$	booleans
$\mathtt{T}_{\mathrm{L}}$	$=$	$\{\mathit{mutable}, \mathit{immutable}\}$	mutability flags
$\mathtt{Q}$			symbols
$\mathtt{H}$			characters
$\mathtt{R}$			numbers
$\mathtt{E}_{\mathrm{p}}$	$=$	$\mathtt{L} \times \mathtt{L} \times \mathtt{T}_{\mathrm{L}}$	pairs
$\mathtt{E}_{\mathrm{v}}$	$=$	$\mathtt{L}^* \times \mathtt{T}_{\mathrm{L}}$	vectors
$\mathtt{E}_{\mathrm{s}}$	$=$	$\mathtt{L}^* \times \mathtt{T}_{\mathrm{L}}$	strings
$\mathtt{M}$	$=$	$\mathtt{T} + \mathtt{T}_{\mathrm{L}} + \{\mathit{null},\ \mathit{empty},\ \mathit{unspecified}\}$	miscellaneous
$\phi \in \mathtt{F}$	$=$	$\mathtt{L} \times (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C})$	procedure values
$\epsilon \in \mathtt{E}$	$\cong$	$\mathtt{Q} + \mathtt{H} + \mathtt{R} + \mathtt{E}_{\mathrm{p}} + \mathtt{E}_{\mathrm{v}} + \mathtt{E}_{\mathrm{s}} + \mathtt{M} + \mathtt{F}$	expressed values
$\sigma \in \mathtt{S}$	$=$	$\mathtt{E}^*$	stores
$\theta \in \mathtt{C}$	$=$	$\mathtt{S} \to \mathtt{A}$	command continuations
$\kappa \in \mathtt{K}$	$=$	$\mathtt{E}^* \to \mathtt{C}$	expression continuations
$\mathtt{A}$			answers
$\mathtt{X}$			errors

Table 4: Domains for the Semantics of Scheme

$$\mathcal{K} : \mathrm{Con} \to \mathtt{E}$$
$$\mathcal{E} : \mathrm{Exp} \to \mathtt{U} \to \mathtt{K} \to \mathtt{C}$$
$$\mathcal{E}^* : \mathrm{Exp}^* \to \mathtt{U} \to \mathtt{K} \to \mathtt{C}$$
$$\mathcal{C} : \mathrm{Com}^* \to \mathtt{U} \to \mathtt{C} \to \mathtt{C}$$

Table 5: Scheme Semantic Functions

sum, say $\psi_{\mathtt{E}} : \mathtt{E} \cong \mathtt{E}_{\Sigma}$. Although $\mathtt{A}$ is not explicitly defined, we will stipulate throughout the remainder of this paper that it is some domain containing a $\perp$ and also numbers.

Semantics of Constants. The semantics of constants, given by a function $\mathcal{K}$, will not be completely defined. Rather, we will give constraints on this function. In essence, we consider its actual value to be a parameter to the semantics of Scheme. We do explicitly define its behavior on primitive constants, i.e. constants not requiring storage at run-time.

$c : \mathtt{Q} \vee c : \mathtt{H} \vee c : \mathtt{R} \Rightarrow \mathcal{K}[\![c]\!] = c$

$\mathcal{K}[\![\mathrm{NIL}]\!] = \mathit{null}$ in $\mathtt{E}$; $\mathcal{K}_0[\![\#\mathrm{F}]\!] = \mathit{false}$ in $\mathtt{E}$; and $\mathcal{K}_0[\![\#\mathrm{T}]\!] = \mathit{true}$ in $\mathtt{E}$

In addition, we constrain its behavior on objects, namely strings, vectors, and pairs, that do require storage and can be named by constants. There are three conditions:

1. Types and lengths are correct:
   (a) If c is a string of length n, and $\mathcal{K}[\![c]\!]$ is a non-bottom value ϵ, then $\epsilon : \mathtt{E}_{\mathrm{s}}$, and $\#(\epsilon \mid \mathtt{E}_{\mathrm{s}}\, 0) = n$;
   (b) If c is a vector of length n, and $\mathcal{K}[\![c]\!]$ is a non-bottom value ϵ, then $\epsilon : \mathtt{E}_{\mathrm{v}}$, and $\#(\epsilon \mid \mathtt{E}_{\mathrm{v}}\, 0) = n$;
   (c) If c is a pair (and thus also if it is list or dotted list), and $\mathcal{K}[\![c]\!]$ is a non-bottom value ϵ, then $\epsilon : \mathtt{E}_{\mathrm{p}}$.
2. The immutability bit is set: if c is a string or vector, and $\mathcal{K}[\![c]\!]$ is a non-bottom value ϵ, then $(\epsilon \mid \mathtt{D})\ 1 = \mathit{immutable}$, where $\mathtt{D}$ is either $\mathtt{E}_{\mathrm{s}}$ or $\mathtt{E}_{\mathrm{v}}$. Similarly, if c is a pair, and $\mathcal{K}[\![c]\!]$ is a non-bottom value ϵ, then $(\epsilon \mid \mathtt{E}_{\mathrm{p}})\ 2 = \mathit{immutable}$.
3. Subexpressions are well-defined: if c_0 is a vector or pair, and $\mathcal{K}[\![c_0]\!]$ is a non-bottom value ϵ, and c_1 is a subexpression of c_0, then $\mathcal{K}[\![c_1]\!]$ is also non-bottom.

As an inductive consequence of the first condition, we may infer that in no case does $\mathcal{K}[\![c]\!] : \mathtt{F}$ hold. It also follows that the only real freedom in the definition of $\mathcal{K}$ concerns which locations are occupied by storage-requiring objects such as lists and vectors.

Semantics of Expressions. We give the clauses associated with procedure call as a sample of the Scheme semantics. *Permute* and *unpermute* are functions that determine the order of evaluation; they are fully specified in [9].

$\psi \in \mathtt{K}_1 \;= \mathtt{E} \to \mathtt{C}$	one argument expression continuations
$\rho_R \in \mathtt{U}_R = \mathtt{N} \to \mathtt{N} \to \mathtt{L}$	runtime environments
$\pi \in \mathtt{P} \;= \mathtt{E} \to \mathtt{E}^* \to \mathtt{U}_R \to \mathtt{K}_1 \to \mathtt{C}$	code segments

Table 6: Byte Code Semantics: Additional Domains

$$\begin{aligned}
&\mathcal{E}[\![\mathrm{E}_0 :: \mathrm{E}^*]\!] = \\
&\quad \lambda\rho\kappa\,.\,\mathcal{E}^*(\mathit{permute}(\langle \mathrm{E}_0\rangle \frown \mathrm{E}^*)) \\
&\qquad\qquad \rho \\
&\qquad\qquad (\lambda\epsilon^*\,.\,((\lambda\epsilon^*\,.\,\mathit{applicate}\;(\epsilon^*\;0)\;(\epsilon^* \dagger 1)\;\kappa) \\
&\qquad\qquad\qquad (\mathit{unpermute}\;\epsilon^*)))
\end{aligned}$$

$$\mathcal{E}^*[\![\langle\rangle]\!] = \lambda\rho\kappa\,.\,\kappa\langle\;\rangle$$

$$\begin{aligned}
&\mathcal{E}^*[\![\mathrm{E}_0 :: \mathrm{E}^*]\!] = \\
&\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\;\rho\;(\mathit{single}(\lambda\epsilon_0\,.\,\mathcal{E}^*[\![\mathrm{E}^*]\!]\;\rho\;(\lambda\epsilon^*\,.\,\kappa\;(\langle\epsilon_0\rangle \frown \epsilon^*))))
\end{aligned}$$

$$\begin{aligned}
&\mathit{single} : (\mathtt{E} \to \mathtt{C}) \to \mathtt{K} \\
&\mathit{single} = \\
&\quad \lambda\psi\epsilon^*\,.\,\#\epsilon^* = 1 \to \psi(\epsilon^*\;0), \\
&\qquad\qquad \mathit{wrong}\;\text{“wrong number of return values”}
\end{aligned}$$

$$\begin{aligned}
&\mathit{applicate} : \mathtt{E} \to \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
&\mathit{applicate} = \\
&\quad \lambda\epsilon\epsilon^*\kappa\,.\,\epsilon : \mathtt{F} \to ((\epsilon \mid \mathtt{F})\;1)\epsilon^*\kappa, \mathit{wrong}\;\text{“bad procedure”}
\end{aligned}$$

2.2. The Byte Code Semantics

One can understand a byte code program as operating on a state with four “registers”, so to speak. These are a value register, containing an element of $\mathtt{E}$, an argument stack, an environment register, and a continuation register, where the continuations here take only a single value, unlike the multiple value continuations of the official Scheme semantics. A program, when applied to values of these four kinds, yields a command continuation $\theta \in \mathtt{C}$. This in turn, if given a store σ, determines an answer. Thus, a code segment determines a computational answer if four registers and a store are given.

This view is expressed formally in the signature of the domain π of code segments, as it appears in Table 6. Representative semantic clauses stated

$$\mathcal{B} : b \cup \{\langle\rangle\} \to \mathtt{U} \to \mathtt{P}$$
$$\mathcal{Z} : z \to \mathtt{U} \to \mathtt{P} \to \mathtt{P}$$
$$\mathcal{Y} : y \cup \{\langle\rangle\} \to \mathtt{U} \to \mathtt{P} \to \mathtt{P}$$
$$\mathcal{T} : t \to \mathtt{U} \to \mathtt{P}$$

(The variable y' will range over $y \cup \{\langle\rangle\}$.)

Table 7: Byte Code Semantic Functions

$$\mathcal{B}[\![\langle\rangle]\!] = \lambda\rho\epsilon\epsilon^*\rho_R\psi\sigma \,.\, \epsilon : \mathtt{R} \to \epsilon \,|\, \mathtt{R} \text{ in } \mathtt{A}, \bot$$
$$\mathcal{B}[\![z :: b]\!] = \lambda\rho \,.\, \mathcal{Z}[\![z]\!]\rho \; (\mathcal{B}[\![b]\!]\rho)$$
$$\mathcal{B}[\![\langle \mathtt{make\text{-}cont}\; b_1\; n\rangle :: b_2]\!] = \lambda\rho \,.\, \mathit{make_cont}\; (\mathcal{B}[\![b_1]\!]\rho) n (\mathcal{B}[\![b_2]\!]\rho)$$
$$\mathcal{B}[\![\langle\langle \mathtt{return}\rangle\rangle]\!] = \lambda\rho \,.\, \mathit{return}$$
$$\mathcal{B}[\![\langle\langle \mathtt{call}\; n\rangle\rangle]\!] = \lambda\rho \,.\, \mathit{call}\, n$$
$$\mathcal{Y}[\![\langle \mathtt{make\text{-}cont}\; y'\; n\rangle :: b]\!] = \lambda\rho\pi \,.\, \mathit{make_cont}\; (\mathcal{Y}[\![y']\!]\rho\pi)\; n\; (\mathcal{B}[\![b]\!]\rho)$$
$$\mathcal{Y}[\![z :: y']\!] = \lambda\rho\pi \,.\, \mathcal{Z}[\![z]\!]\rho \; (\mathcal{Y}[\![y']\!]\rho\pi)$$
$$\mathcal{Y}[\![\langle\rangle]\!] = \lambda\rho\pi \,.\, \pi$$
$$\mathcal{Z}[\![\langle \mathtt{literal}\; c\rangle]\!] = \lambda\rho \,.\, \mathit{literal}\, (\mathcal{K}[\![c]\!])$$
$$\mathcal{Z}[\![\langle \mathtt{closure}\; t\rangle]\!] = \lambda\rho \,.\, \mathit{closure}\, (\mathcal{T}[\![t]\!]\rho)$$
$$\mathcal{Z}[\![\langle \mathtt{global}\; i\rangle]\!] = \lambda\rho \,.\, \mathit{global}\, (\mathit{lookup}\; \rho\; i)$$
$$\mathcal{Z}[\![\langle \mathtt{set\text{-}global!}\; i\rangle]\!] = \lambda\rho \,.\, \mathit{set_global}\, (\mathit{lookup}\; \rho\; i)$$
$$\mathcal{T}[\![\langle \mathtt{lap}\; c\; b\rangle]\!] = \mathcal{B}[\![b]\!]$$

Table 8: Byte Code Semantical Clauses

in terms of these semantic functions are presented in Table 8.

The clause for a null code sequence defines the "answer function," that is, the way that a computational answer in `A` is determined when the "program register" contains no more code and the program has thus completed execution.

The auxiliary functions (among them, those in Table 9) follow the Wand-Clinger style of byte code specification. In each case, the auxiliary function takes as arguments:

- the denotations of the arguments to the instruction (if necessary);
- the denotation of the code following the current instruction;

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

$call : \mathtt{N} \to \mathtt{P}$
$call = \lambda\nu \,.\, \lambda\epsilon\epsilon^*\rho_R\psi \,.\, \#\epsilon^* = \nu \to applicate\; \epsilon\epsilon^*(single\; \psi),$
$\qquad wrong$ "bad stack"

$return : \mathtt{P}$
$return = \lambda\epsilon\epsilon^*\rho_R\psi \,.\, \psi\epsilon$

$push : \mathtt{P} \to \mathtt{P}$
$push = \lambda\pi \,.\, \lambda\epsilon\epsilon^*\rho_R\psi \,.\, \pi\epsilon(\epsilon^* \frown \langle\epsilon\rangle)\rho_R\psi$

$lookup : \mathtt{U} \to \mathrm{Ide} \to \mathtt{L}$
$lookup = \lambda\rho\mathrm{I} \,.\, \rho\mathrm{I}$

Table 9: Byte Code Auxiliary Functions

- a sequence of four arguments, $\epsilon, \epsilon^*, \rho_R, \psi$, representing the contents of the value register, the argument stack, the environment register, and the continuation register respectively.

In typical cases, such as for instance *make_cont*, the definition returns a value of the form $\pi\epsilon'\epsilon_1{}^*\rho'_R\psi'$ for non-erroneous arguments. In effect, the instruction has invoked the remainder of the code in a new state, computed from the arguments given. Thus one may view the semantic computation of a final result as consisting of a sequence of terms of this form. The first term in the sequence is determined directly by applying the semantic clauses to the byte code program, together with some initial values. Each successive term is the result of expanding the definition for the auxiliary function used in the head instruction, followed by β-reductions and evaluations of conditionals.

Some auxiliaries, such as *return*, by contrast, do not take this form. However, if the continuation argument ψ has been created by a *make_cont*, then ψ has the form $(\lambda\epsilon \,.\, \pi'\epsilon\epsilon^*\rho_R\psi')$, so that $\psi\epsilon$ β-reduces to a term of the requisite form $\pi'\epsilon\epsilon^*\rho_R\psi'$. A similar relation links *call* to the auxiliary *closure*, which creates procedure values.

These observations were a basic part of the motivation for the Wand-

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

Table 10: Naïve Counterexample to Single-Valued Implementations

Clinger approach, and they served to motivate our approach to proving the faithfulness of the operational semantics. They can also serve as the basis for a proof of adequacy, such as is given in [16, Theorem 6], which would establish that an operational semantics computes all the non-erroneous, non-bottom answers predicted by the denotational semantics.

2.3. Multiple Values in the Scheme Semantics

The official semantics for Scheme allows a procedure to return several values to its caller, as would be needed to model the Common Lisp `values` construct or the **T** `return` form. However, IEEE standard Scheme has no construct that allows a programmer to construct a procedure that would return more than one value. Assuming that a program is started with an initial store that hides no "multiple value returners," then an implementation may assume that there will never be any in the course of execution. So, many implementations of Scheme, among them VLISP, do not support multiple return values. This contrast may be seen at the formal level in the contrast between the domain of expression continuations $\mathtt{K}$, as used in Scheme semantics, and the domain of one argument expression continuations $\mathtt{K}_1$ which plays a corresponding role in the semantics of the byte code.

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

We can make this point clear with an example. Consider the program `(F)`, which calls a procedure with no arguments. Given the values described in Table 10, we can easily see that the semantics predict that the correct computational answer is 3. However, saying that an implementation makes no provision for multiple-value returners is in effect to say that if κ is implemented at all, then it always results in a computational answer of 1.

In the next section, we will formalize the reasoning that justifies an implementation in assuming it need implement only "single-valued" objects,

ϕ_p	$\in$	$\mathtt{F}_p$	$=$	$\mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$	pure (untagged) procedure values

Table 11: Pure Procedure Objects

and need make no provision for multiple value returns.

In essence our approach is to introduce a new, "smaller" semantics for Scheme. In this semantics, it is clear that there is no mechanism for multiple return values. With this alternate semantics, called $(\mathbf{sva}\,\mathcal{E})$, in place, there are two separate facts that must be proved to justify the compiler algorithm.

1. The alternate semantics is faithful to the standard semantics, at least in the intended case in which the initial givens harbor no multiple value returns:
$$\mathcal{E}[\![e]\!]\rho\,\kappa\,\sigma = (\mathbf{sva}\,\mathcal{E})[\![e]\!]\rho\,\kappa\,\sigma,$$
when κ and σ, the halt continuation and the initial store respectively, are unproblematic, single-valued objects in a sense to be defined.

2. The compiled byte code is faithful to the alternate semantics, in the sense that
$$(\mathbf{sva}\,\mathcal{E})[\![e]\!]\rho \quad \text{and} \quad \mathcal{B}[\![C(e)]\!]\rho$$
yield the same answer when applied to suitable initial values.

We will introduce for convenience a new explicitly defined domain of *pure procedure objects*. Unlike the procedure objects in $\mathtt{F}$ (which equals $\mathtt{L} \times (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C})$), those in $\mathtt{F}_p$ contain only the function-like part (namely $\mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$), without the location that serves as a tag (see Table 11). The location tags are used to decide whether two procedure objects are the same in the sense of the Scheme standard procedure `eqv?`. So $\mathtt{F} = \mathtt{L} \times \mathtt{F}_p$. It will also be convenient to define a few auxiliary functions:

Definition 1 (Auxiliaries)

1. *zeroth* $: \mathtt{E}^* \to \mathtt{E}$ *is the strict function returning* $\bot_\mathtt{E}$ *if its argument* ϵ^* *is either* $\bot_{\mathtt{E}^*}$ *or* $\langle\rangle$*, and which returns* $(\epsilon^*\,0)$ *otherwise.*

2. *truncate* $: \mathtt{E}^* \to \mathtt{E}^*$ *is the non-strict function which takes its argument* ϵ^* *to* $\langle \mathit{zeroth}\,\epsilon^*\rangle$*. Hence* $\#(\mathit{truncate}\,\epsilon^*) = 1$*, even for* $\epsilon^* = \bot_{\mathtt{E}^*}$*.*

3. *one_arg* $: \mathtt{K} \to \mathtt{K}_1$ *is defined to equal* $\lambda\kappa\epsilon\,.\,\kappa\langle\epsilon\rangle$

4. *multiple* : $\mathtt{K}_1 \to \mathtt{K}$ *is defined to equal* $\lambda\psi\epsilon^* \,.\, \psi(zeroth\,\epsilon^*)$

The auxiliaries *one_arg* and *multiple* coerce from $\mathtt{K}$ to $\mathtt{K}_1$ and back. The name *truncate* is quite long, and we sometimes prefer to truncate it as *trunc*. Expanding the definitions, we have:

Lemma 2 1. $one_arg(multiple\,\psi) = \psi$;

2. $multiple(one_arg\,\kappa) = \lambda\epsilon^* \,.\, \kappa(truncate\,\epsilon^*)$.

We define next the function **sva**, which, given any object in Ω, returns the single-valued object that approximates it. As we will see, **sva** ω always belongs to the same summand of Ω as ω. Moreover, **sva** is idempotent, so that we can think of its range as determining the single-valued objects. We will extend **sva** immediately after defining it so that it may be applied to elements of the disjoint summands of Ω, in addition to the elements of Ω itself.

The heart of **sva** is its behavior on pure procedure objects ϕ_p. A pure procedure object ϕ_p is altered to another function ϕ'_p, such that ϕ'_p calls ϕ_p with the same expressed value arguments and the same store. Only the continuation argument is altered, to ensure that the modified continuation can access only one expressed value, and only a single-valued approximation to that. Similarly, only a single-valued approximation to the contents of the new store is made available to the continuation. Crudely put, ϕ'_p cannot pass anything infected with multiple values to its continuation, whether in the store, or by making available more than one return value itself, or by hiding it in a procedure value that would later, when applied, make more than one value available to *its* continuation.

As for the uncontroversially given domains that do not involve $\mathtt{E}$, **sva** is the identity. For all other domains, **sva** commutes in the obvious way with the constructor of the domain.

Recall that we write $\mathtt{E}_\Sigma$ for $\mathtt{Q} + \mathtt{H} + \mathtt{R} + \mathtt{E}_\mathrm{p} + \mathtt{E}_\mathrm{v} + \mathtt{E}_\mathrm{s} + \mathtt{M} + \mathtt{F}$, and $\psi_\mathtt{E}$ for the isomorphism from $\mathtt{E}$ to $\mathtt{E}_\Sigma$. We assume $\mathtt{E}$ to be indecomposable in the sense that that it is not produced by any of the domain-constructing operators $\to$, $+$, *, and $\times$, which produce function domains, disjoint union domains, sequence domains, and product domains respectively. We also assume these operators are formalized so that given a domain, we can read off at most one operator and one list of argument domains that produced it.

We define Ω to be the disjoint sum (in any order) of a set C_Ω of domains, where C_Ω is the least set containing all of the Scheme denotational domains (including E_Σ) and such that

1. every argument of every element of C_Ω is an element of C_Ω, and
2. if D is constructed by a basic constructor from a finite sequence of arguments in C_Ω, then D is an element of C_Ω.

Given $f : \Omega \to \Omega$, say that f *respects types* if f is strict and for every D in C_Ω and every x in Ω such that $x : D$, we have $fx : D$; also, for each D in C_Ω, let $f^D : D \to D$ be defined, for $d \in D$, to be

$$f^D d = (f(d \text{ in } \Omega))\,|D.$$

As a preliminary step towards single-valued approximations, define an operator $\alpha : (\Omega \to \Omega) \to (\Omega \to \Omega)$ as follows. Let $f : \Omega \to \Omega$ and $\omega \in \Omega$ be arbitrary. Then $\alpha f \omega$ is

(a) if $\omega = \bot_\Omega$, then ω

(b) if $\omega : D$ for an indecomposable D other than $\mathtt{E}$, then ω,

(c) if $\omega : \mathtt{E}$, then

$$(\psi_{\mathtt{E}}^{-1}(f(\psi_{\mathtt{E}}(\omega|\mathtt{E}) \text{ in } \Omega)|\mathtt{E}_\Sigma)) \text{ in } \Omega,$$

(d) if $\omega : D$ for a decomposable D other than $\mathtt{F}_p$, then associate with each argument D_i of D the functional $f^{D_i} : D_i \to D_i$, lift these to $g : D \to D$, and take $g(\omega|D)$ in Ω, and

(e) if $\omega : \mathtt{F}_p$, then

$$(\lambda\epsilon^*\kappa\,.\,(\omega \mid \mathtt{F}_p)\,(f^{\mathtt{E}^*}\,\epsilon^*)\,(\lambda\epsilon^*\,.\,(f^{\mathtt{C}}(\kappa\,(\mathit{truncate}\,(f^{\mathtt{E}} \circ \epsilon^*))))))\text{ in } \Omega,$$

(here $(f^{\mathtt{E}} \circ \epsilon^*)$ is taken to be $\bot_{\mathtt{E}^*}$ for $\epsilon^* = \bot_{\mathtt{E}^*}$)

All of the real work of α is done by the truncation in the last case. In all cases, if $\omega : D$, then $\alpha f \omega : D$; because in addition αf is strict, for all $f : \Omega \to \Omega$, αf respects types.

Definition 2 (Single-valued approximation) sva $: \Omega \to \Omega$ *is the least fixed point of the above* α.

For $D \in C_\Omega$ and $d \in D$, we will abuse notation by writing **sva** d for $\mathbf{sva}^D\,d$, i.e., for $(f(d \text{ in } \Omega)|D)$. The following lemma provides the crucial justification for regarding **sva** as providing an alternative semantics.

Lemma 3 sva *is idempotent.*

Proof: If we let $f_0 = \bot_{\Omega\to\Omega}$, and $f_{n+1} = \alpha f_n$, then **sva** is the supremum of the f_n for $n \in N$. Define a strict $g_0 : \Omega \to \Omega$ by letting $g_0\,\omega = \bot_D$ in Ω whenever $\omega : D$ (so each $g_0^D = \bot_{D\to D}$). Then g_0 is least type-respecting element of $\Omega \to \Omega$. Let $g_{n+1} = \alpha g_n$. As g_1 respects types, $g_0 \sqsubseteq g_1$ and by the monotonicity of α, $g_n \sqsubseteq g_{n+1}$. Since f_1 respects types, $f_0 \sqsubseteq g_0 \sqsubseteq f_1$, so that the supremum of the g_n's is also **sva**.

For $h : \Omega \to \Omega$, let $P(h)$ mean that

(i) h respects types;
(ii) each h^D is strict; and
(iii) h is idempotent.

Since the supremum of idempotent elements of $\Omega \to \Omega$ is idempotent, and $P(g_0)$ holds, it suffices to establish that α preserves the property P. Take an arbitrary f such that $P(f)$ and let h be αf. We already know that (i) holds, and (ii) follows easily taking cases on D. For idempotence of h, pick an arbitrary $\omega \in \Omega$; the only interesting case is (e). Let $\omega : \mathtt{F}_p$; we must show that $(\alpha\ f)(\alpha\ f\ \omega) = (\alpha\ f\ \omega)$. By (e), the former (projected into $\mathtt{F}_p$ for convenience) equals:

$$\begin{aligned}
&\lambda\epsilon^*\kappa\,.\,(\lambda\epsilon^*\kappa\,.\,(\omega \mid \mathtt{F}_p)(f^{\mathtt{E}^*}\epsilon^*)(\lambda\epsilon_0{}^*\,.\,f^{\mathtt{C}}(\kappa(\mathit{trunc}(f^{\mathtt{E}} \circ \epsilon_0{}^*)))))\\
&\qquad(f^{\mathtt{E}^*}\epsilon^*)(\lambda\epsilon_1{}^*\,.\,f^{\mathtt{C}}(\kappa(\mathit{trunc}(f^{\mathtt{E}} \circ \epsilon_1{}^*))))\\
=\ &\lambda\epsilon^*\kappa\,.\,(\omega \mid \mathtt{F}_p)(f^{\mathtt{E}^*}(f^{\mathtt{E}^*}\epsilon^*))\\
&\qquad(\lambda\epsilon_0{}^*\,.\,f^{\mathtt{C}}(f^{\mathtt{C}}(\kappa(\mathit{trunc}(f^{\mathtt{E}} \circ (\mathit{trunc}(f^{\mathtt{E}} \circ \epsilon_0{}^*)))))))\\
=\ &\lambda\epsilon^*\kappa\,.\,(\omega \mid \mathtt{F}_p)(f^{\mathtt{E}^*}\epsilon^*)\\
&\qquad(\lambda\epsilon_0{}^*\,.\,f^{\mathtt{C}}(\kappa(\mathit{trunc}(f^{\mathtt{E}} \circ (\mathit{trunc}(f^{\mathtt{E}} \circ \epsilon_0{}^*)))))),
\end{aligned}$$

We have used β-reduction twice in the first step, and the idempotence of f and therefore also f^D in the second. So it suffices to prove that

$$\mathit{trunc}(f^{\mathtt{E}} \circ (\mathit{trunc}(f^{\mathtt{E}} \circ \epsilon_0{}^*))) = \mathit{trunc}(f^{\mathtt{E}} \circ \epsilon_0{}^*),$$

which follows easily taking cases on whether $0 < \#\epsilon_0{}^*$. ∎

By the convention that $(\mathbf{sva}\,x) = x$ when x belongs to a syntactic domain such as the Scheme expressions, we may apply **sva** to the semantic functions $\mathcal{K}$, $\mathcal{E}$, $\mathcal{E}^*$, and $\mathcal{C}$. Since Scheme has no syntax for constants denoting procedure objects, both c and $\mathcal{K}[\![c]\!]$ are necessarily single-valued, so that $(\mathbf{sva}\,\mathcal{K}) = \mathcal{K}$. However, the remaining semantic functions are not single valued, and the alternative semantics consists in replacing them with their single-valued approximations $(\mathbf{sva}\,\mathcal{E})$, $(\mathbf{sva}\,\mathcal{E}^*)$, and $(\mathbf{sva}\,\mathcal{C})$.

2.4. Faithfulness of the Alternative Semantics

The alternative semantics is faithful in the sense that, for every Scheme expression e, it delivers the same computational answer as the official semantics, provided that a single-valued initial continuation and store are supplied. As mentioned previously, it is reasonable for a Scheme implementation to provide a single-valued store. Moreover, many natural initial continuations (which, intuitively, say how to extract the computational answer if the program finally halts) are single-valued. For instance, the VLISP operational semantics for the Tabular Byte Code Machine is justified against the denotational semantics using the initial continuation:

$$halt = \lambda\epsilon^*\sigma\,.\,(\epsilon^*\,0) : \mathtt{R} \rightarrow (\epsilon^*\,0) \mid \mathtt{R} \text{ in } \mathtt{A}, \bot$$

which is single-valued.

Theorem 4 (Faithfulness of Alternative Semantics)

1. *For all Scheme expressions e, environments ρ, expression continuations κ, and stores σ,*

$$(\mathbf{sva}\,\mathcal{E})[\![e]\!]\rho\,\kappa\,\sigma = \mathcal{E}[\![e]\!]\rho\,(\mathbf{sva}\,\kappa)\,(\mathbf{sva}\,\sigma) \tag{1}$$

2. *Let $\kappa = \mathbf{sva}\,\kappa$ and $\sigma = \mathbf{sva}\,\sigma$. Then*

$$\mathcal{E}[\![e]\!]\rho\,\kappa\,\sigma = (\mathbf{sva}\,\mathcal{E})[\![e]\!]\rho\,\kappa\,\sigma$$

Proof: 1. We simply use the definition of **sva**, together with the fact that the domain of Scheme expressions, the domain of environments, and the domain of answers do not involve $\mathtt{E}$:

$$\begin{aligned}
(\mathbf{sva}\,\mathcal{E})[\![e]\!]\rho\,\kappa\,\sigma &= (\mathbf{sva}(\mathcal{E}[\![\mathbf{sva}\,e]\!]))\rho\,\kappa\,\sigma \\
&= (\mathbf{sva}(\mathcal{E}[\![e]\!](\mathbf{sva}\,\rho)))\,\kappa\,\sigma \\
&= (\mathbf{sva}(\mathcal{E}[\![e]\!]\rho\,(\mathbf{sva}\,\kappa)))\,\sigma \\
&= (\mathbf{sva}(\mathcal{E}[\![e]\!]\rho\,(\mathbf{sva}\,\kappa)\,(\mathbf{sva}\,\sigma))) \\
&= (\mathcal{E}[\![e]\!]\rho\,(\mathbf{sva}\,\kappa)\,(\mathbf{sva}\,\sigma))
\end{aligned}$$

2. We use the assumption that κ and σ are single-valued, followed by Equation 1, from right to left:

$$\begin{aligned}
\mathcal{E}[\![e]\!]\rho\,\kappa\,\sigma &= \mathcal{E}[\![e]\!]\rho\,(\mathbf{sva}\,\kappa)\,(\mathbf{sva}\,\sigma) \\
&= (\mathbf{sva}\,\mathcal{E})[\![e]\!]\rho\,\kappa\,\sigma
\end{aligned}$$

∎

Corollary 5 *For all values of* e, ρ, κ, *and* σ*:*

$$\begin{array}{rcl}(\mathbf{sva}\,\mathcal{E})[\![e]\!]\rho\,\kappa\,\sigma & = & (\mathbf{sva}\,\mathcal{E})[\![e]\!]\rho\,(\mathbf{sva}\,\kappa)\,\sigma \\ & = & (\mathbf{sva}\,\mathcal{E})[\![e]\!]\rho\,\kappa\,(\mathbf{sva}\,\sigma) \\ & = & (\mathbf{sva}\,\mathcal{E})[\![e]\!]\rho\,(\mathbf{sva}\,\kappa)\,(\mathbf{sva}\,\sigma)\end{array}$$

Proof: Use Equation 1 and the idempotence of **sva** repeatedly. ∎

A further partial justification for the single-valued semantics that we have introduced is that it allows us to eliminate the operator *single* from the semantic clauses in which it occurs, and to truncate any continuation. These two facts are intuitively significant, as they amount to saying that the meaning of a Scheme expression, if it invokes its expression continuation at all, applies it to a sequence $\langle\epsilon\rangle$ of length 1. Hence they justify the claim that the alternate semantics ensures that procedures never return multiple values.

Theorem 6 (Truncate Continuations)
$(\mathbf{sva}\,\mathcal{E})[\![\mathrm{E}]\!]\rho\,\kappa = (\mathbf{sva}\,\mathcal{E})[\![\mathrm{E}]\!]\rho\,(\lambda\epsilon^* \,.\, \kappa\,(\mathit{trunc}\,\epsilon^*))$

Proof: The proof is by induction on E. The cases where E is a constant, an identifier, a lambda expression, a dotted lambda expression, or an assignment are immediate from the definitions of *send* and *hold*. The cases for two- and three-expression `if`, and for `begin`, are immediate from the induction hypothesis. Hence the only case of interest is for procedure call, which is as it should be. Using the semantics of procedure call, we must show

$$(\mathbf{sva}\,\mathcal{E}^*)(\mathit{permute}(\langle \mathrm{E}_0\rangle ^\frown \mathrm{E}^*))\rho\,\kappa_1 = (\mathbf{sva}\,\mathcal{E}^*)(\mathit{permute}(\langle \mathrm{E}_0\rangle ^\frown \mathrm{E}^*))\rho\,\kappa_2$$

where

$$\begin{array}{l}\kappa_1 = \lambda\epsilon^* \,.\, ((\lambda\epsilon^* \,.\, \mathit{applicate}\,(\epsilon^*\,0)\,(\epsilon^* \dagger 1)\,\kappa)(\mathit{unpermute}\,\epsilon^*)) \\ \kappa_2 = \lambda\epsilon^* \,.\, ((\lambda\epsilon^* \,.\, \mathit{applicate}\,(\epsilon^*\,0)\,(\epsilon^* \dagger 1)\,(\lambda\epsilon^* \,.\, \kappa\,(\mathit{trunc}\,\epsilon^*)))(\mathit{unpermute}\,\epsilon^*))\end{array}$$

Pushing **sva**s through κ_1, we obtain:

$$\begin{array}{l}\lambda\epsilon^* \,.\, ((\lambda\epsilon^* \,.\, \mathbf{sva}(\mathit{applicate}\,(\mathbf{sva}(\epsilon^*\,0))\,(\mathbf{sva}(\epsilon^* \dagger 1)) \\ \qquad\qquad\qquad\qquad (\lambda\epsilon^* \,.\, (\mathbf{sva}\kappa)(\mathit{trunc}\,\epsilon^*)))) \\ \qquad (\mathbf{sva}(\mathit{unpermute}\,(\mathbf{sva}\epsilon^*))))\end{array}$$

Pushing **sva**s throughκ_2 yields:

$$\begin{array}{l}\lambda\epsilon^* \,.\, ((\lambda\epsilon^* \,.\, \mathbf{sva}(\mathit{applicate}\,(\mathbf{sva}(\epsilon^*\,0))\,(\mathbf{sva}(\epsilon^* \dagger 1)) \\ \qquad\qquad\qquad\qquad (\lambda\epsilon^* \,.\, (\lambda\epsilon^* \,.\, \mathbf{sva}(\kappa\,(\mathbf{sva}(\mathit{trunc}\,\epsilon^*)))) \\ \qquad\qquad\qquad\qquad\qquad (\mathit{trunc}\,\epsilon^*)))) \\ \qquad (\mathbf{sva}(\mathit{unpermute}\,(\mathbf{sva}\epsilon^*)))).\end{array}$$

For these two expressions to be equal, it certainly suffices that for all κ,

$$\lambda\epsilon^* \,.\, (\mathbf{sva}\kappa)(\mathit{trunc}\,\epsilon^*) = \lambda\epsilon^* \,.\, (\lambda\epsilon^* \,.\, \mathbf{sva}(\kappa\,(\mathbf{sva}(\mathit{trunc}\,\epsilon^*))))(\mathit{trunc}\,\epsilon^*)$$

Using the definition of **sva** and the idempotence of *trunc*, we have:

$$\begin{aligned} &\lambda\epsilon^* \,.\, (\mathbf{sva}\kappa)(\mathit{trunc}\,\epsilon^*) \\ &\quad = \quad \lambda\epsilon^* \,.\, \mathbf{sva}(\kappa\,(\mathbf{sva}(\mathit{trunc}\,\epsilon^*))) \\ &\quad = \quad \lambda\epsilon^* \,.\, \mathbf{sva}(\kappa\,(\mathbf{sva}(\mathit{trunc}\,(\mathit{trunc}\,\epsilon^*)))) \end{aligned}$$

∎

Theorem 7 ("Single" eliminable)
$(\mathbf{sva}\,\mathcal{E})[\![\mathrm{E}]\!]\rho\,(\mathit{single}\,\psi) = (\mathbf{sva}\,\mathcal{E})[\![\mathrm{E}]\!]\rho\,(\lambda\epsilon^* \,.\, \psi\,(\epsilon^*\,0))$

The very similar proof is also by induction on E.

2.5. Compiler Correctness for Single-valued Semantics

We will prove that if e is any Scheme program and b is the result of compiling it, then e and b are equivalent in the following sense:

Definition 3 (Computational equivalence $\equiv$)
A Scheme program e and a byte code program b are computationally equivalent if, for every environment ρ and expression continuation κ,

$$(\mathbf{sva}\,\mathcal{E})[\![e]\!]\rho\kappa = (\mathbf{sva}\mathcal{B})[\![b]\!]\rho\ \mathit{unspecified}\ \langle\rangle\ (\lambda\nu_1\nu_2 \,.\, \bot)\ (\mathit{one_arg}\,\kappa).$$

Definition 4 (Environment Composition) *Consider $\rho : \mathit{Ide} \to \mathtt{L}$, $\rho_C : \mathit{Ide} \to (\mathtt{N} \times \mathtt{N} \cup \{\mathit{not_lexical}\})$, and $\rho_R : \mathtt{N} \to \mathtt{N} \to \mathtt{L}$. Their* environment composition *$\rho \triangleleft^{\rho_R}_{\rho_C}$ is the environment:*

$$\lambda i \,.\, \rho_C\, i = \mathit{not_lexical} \to \rho\, i,\ (\lambda p \,.\, \rho_R\,(p\,0)(p\,1))(\rho_C\, i)$$

We will refer to objects like ρ_C and ρ_R as "compile-time environments" and "run-time environment" respectively.

Lemma 8 $\rho \triangleleft^{\rho_R}_{\mathrm{MT}_C} = \rho$.

Proof: For all i, $\mathrm{MT}_C\, i = \mathit{not_lexical}$. Hence, $\rho \triangleleft^{\rho_R}_{\mathrm{MT}_C} i = \rho\, i$, and, by extensionality, $\rho \triangleleft^{\rho_R}_{\mathrm{MT}_C} = \rho$. ∎

We state the main theorem next, and we will give its (short) proof using Lemma 11, which is due to Will Clinger [2]. A portion of the (longer) proof of Clinger's lemma, for the VLISP compiler, is given below.

Theorem 9 (Compiler correctness)
For any Scheme expression e and environment ρ,

$$e \equiv C(e).$$

Proof: The theorem is a direct consequence of Clinger's lemma (Lemma 11 below), Clause 1. Since

$$C(e) = C(e, \mathrm{MT}_C, 0, \mathrm{FALSE}, \langle\langle \texttt{return} \rangle\rangle),$$

we let $(\mathbf{sva}\mathcal{B})[\![C(e)]\!]\rho = \pi_2$, $(\mathbf{sva}\mathcal{B})[\![\langle\langle \texttt{return} \rangle\rangle]\!]\rho = \mathit{return} = \pi_1$, and $\psi = \mathit{one_arg}\,\kappa$. Substituting the initial values in the right hand side, we infer:

$$\begin{array}{l} (\mathbf{sva}\mathcal{E})[\![e]\!]\,(\rho \triangleleft^{\rho_R}_{\mathrm{MT}_C})\,(\mathit{multiple}\,\lambda\epsilon\,.\,\mathit{return}\,\epsilon\langle\rangle\rho_R(\mathit{one_arg}\,\kappa)) \\ \quad = \quad \pi_2\,\mathit{unspecified}\,\langle\rangle\,(\lambda\nu_1\nu_2\,.\,\bot)\,(\mathit{one_arg}\,\kappa). \end{array}$$

Since $\mathit{return} = \lambda\epsilon\epsilon^*\rho_R\psi\,.\,\psi\epsilon$,

$$\begin{array}{rcl} \lambda\epsilon\,.\,\mathit{return}\,\epsilon\langle\rangle\rho_R(\mathit{one_arg}\,\kappa) & = & \lambda\epsilon\,.\,(\mathit{one_arg}\,\kappa)\epsilon \\ & = & (\mathit{one_arg}\,\kappa), \end{array}$$

and it follows that

$$\begin{array}{l} (\mathbf{sva}\mathcal{E})[\![e]\!]\,(\rho \triangleleft^{\rho_R}_{\mathrm{MT}_C})\,(\mathit{multiple}\,(\mathit{one_arg}\,\kappa)) \\ \quad = \quad \pi_2\,\mathit{unspecified}\,\langle\rangle\,(\lambda\nu_1\nu_2\,.\,\bot)\,(\mathit{one_arg}\,\kappa). \end{array}$$

By Lemmas 2 and 8, the left hand side equals:

$$(\mathbf{sva}\mathcal{E})[\![e]\!]\,\rho\,(\lambda\epsilon^*\,.\,\kappa(\mathit{trunc}\,\epsilon^*))$$

By Theorem 6, the latter equals $(\mathbf{sva}\mathcal{E})[\![e]\!]\,\rho\,\kappa$. ∎

Lemma 10 (Compiled expressions set value register)

1. *Let* $\pi = (\mathbf{sva}\mathcal{B})[\![C(e, \rho_C, n, i, b)]\!]\rho$. *For all* ϵ, ϵ', $\pi\epsilon = \pi\epsilon'$.
2. *Let* $\pi = (\mathbf{sva}\mathcal{Y})[\![C(e, \rho_C, n, i, y)]\!]\rho\pi_0$. *For all* ϵ, ϵ', $\pi\epsilon = \pi\epsilon'$.
3. *Let* $\pi = (\mathbf{sva}\mathcal{B})[\![C_{args}(e^*, \rho_C, n, i, b)]\!]\rho$, *and let* $\pi_1(\mathbf{sva}\mathcal{B})[\![b]\!]\rho$. *If for all* ϵ, ϵ', $\pi_1\epsilon = \pi_1\epsilon'$, *then for all* ϵ, ϵ', $\pi\epsilon = \pi\epsilon'$.

Proof: The proof is by simultaneous induction on e and $e*$. We show one case from the (very routine) induction.

Clause 3: If $e* = \langle\rangle$, then $C_{args}(e*, \rho_C, n, i, b) = b$. Otherwise, it is of the form:

$$C(e, \rho_C, n, i, \langle \texttt{push} \rangle :: C_{args}(e_1*, \rho_C, n+1, i, b)),$$

so that the result follows by Clause 1. ∎

Lemma 11 (Clinger's Lemma)

1. *Consider any Scheme expression e,* BBC *closed instruction list b, environment ρ, compile-time environment ρ_C, value sequence $\epsilon*$, run-time environment ρ_R, and initial value ϵ.*

   Let $\pi_1 = (\mathbf{sva}\mathcal{B})[\![b]\!]\rho$ and $\pi_2 = (\mathbf{sva}\mathcal{B})[\![C(e, \rho_C, \#\epsilon*, i, b)]\!]\rho$. Then

   $$(\mathbf{sva}\mathcal{E})[\![e]\!]\ (\rho \triangleleft^{\rho_R}_{\rho_C})\ (\mathit{multiple}\ \lambda\epsilon\ .\ \pi_1\epsilon\epsilon*\rho_R\psi) = \pi_2\epsilon\epsilon*\rho_R\psi.$$

2. *Consider any Scheme expression e,* BBC *open instruction list y, segment π, environment ρ, compile-time environment ρ_C, value sequence $\epsilon*$, run-time environment ρ_R, and initial value ϵ.*

   Let $\pi_1 = (\mathbf{sva}\mathcal{Y})[\![y]\!]\rho\pi$ and $\pi_2 = (\mathbf{sva}\mathcal{Y})[\![C(e, \rho_C, \#\epsilon*, i, y)]\!]\rho\pi$. Then

   $$(\mathbf{sva}\mathcal{E})[\![e]\!]\ (\rho \triangleleft^{\rho_R}_{\rho_C})\ (\mathit{multiple}\ \lambda\epsilon\ .\ \pi_1\epsilon\epsilon*\rho_R\psi) = \pi_2\epsilon\epsilon*\rho_R\psi.$$

3. *Consider any sequence of Scheme expressions $e*$,* BBC *closed instruction list b, environment ρ, compile-time environment ρ_C, value sequence $\epsilon*$, run-time environment ρ_R, and initial values ϵ and ϵ_1.*

   Let $\pi_1 = (\mathbf{sva}\mathcal{B})[\![b]\!]\rho$ and $\pi_2 = (\mathbf{sva}\mathcal{B})[\![C_{args}(e*, \rho_C, \#\epsilon*, i, b)]\!]\rho$.

   Suppose that the after code sets the value register before it reads it, in the sense that, for all values ϵ', $\pi_1\ \epsilon = \pi_1\ \epsilon'$. Then

   $$(\mathbf{sva}\mathcal{E}*)[\![e*]\!]\ (\rho \triangleleft^{\rho_R}_{\rho_C})\ (\lambda\epsilon_1*\ .\ \pi_1\epsilon_1(\epsilon*{}^\frown\epsilon_1*)\rho_R\psi) = \pi_2\epsilon\epsilon*\rho_R\psi.$$

Proof: The proof is by a simultaneous induction on e and $e*$. That is, in proving 1 and 2, we assume that 1 and 2 hold for any proper subexpression of e, and that 3 holds when each expression in $e*$ is a proper subexpression of e. When proving 3, we assume that 1 and 2 hold for each e occurring in $e*$, and that 3 holds of any proper subsequence of $e*$. To emphasize the treatment of the single-valued semantics, we give the proof of clause 3 in detail; the proofs of clauses 1 and 2 are very similar in manner to Clinger's original proof [2].

3. Here we argue by induction on $e*$, assuming that 1 and 2 hold of the expressions occurring in $e*$.

Base Case Suppose that $e^* = \langle\rangle$. Then, by the semantic clause for $\mathcal{E}^*$,

$$\begin{array}{rl} & (\mathbf{sva}\mathcal{E}^*)[\![\langle\rangle]\!](\rho \triangleleft^{\rho_R}_{\rho_C})\ (\lambda \epsilon_1{}^* \,.\, \pi_1 \epsilon_1 (\epsilon^* \frown \epsilon_1{}^*) \rho_R \psi) \\ = & \mathbf{sva}((\lambda \epsilon_1{}^* \,.\, \pi_1 \epsilon_1 (\epsilon^* \frown \epsilon_1{}^*) \rho_R \psi)\langle\rangle) \\ = & \mathbf{sva}(\pi_1 \epsilon_1 (\epsilon^* \frown \langle\rangle) \rho_R \psi) \\ = & \mathbf{sva}(\pi_1 \epsilon_1 \epsilon^* \rho_R \psi). \end{array}$$

Since π_1 is single-valued, the latter equals $\pi_1 \epsilon_1 \epsilon^* \rho_R \psi$. Examining the compiler's code, $C_{args}(\langle\rangle, \rho_C, \#\epsilon^*, i, b) = b$, so that $\pi_1 = \pi_2$. Since, by assumption, $\pi_1\ \epsilon = \pi_1\ \epsilon_1$, the case is complete.

Induction Step Suppose that 3 holds for e^* and and 1 holds for e, and consider $e :: e^*$. Using the semantic clause for $\mathcal{E}^*$, and writing κ for $\lambda \epsilon_1{}^* \,.\, \pi_1 \epsilon_1 (\epsilon^* \frown \epsilon_1{}^*) \rho_R \psi$, the left hand side of our goal takes the form:

$$(\mathbf{sva}\,\mathcal{E})[\![e]\!](\rho \triangleleft^{\rho_R}_{\rho_C})(\mathit{multiple}(\lambda \epsilon_0 . (\mathbf{sva}\,\mathcal{E})^*[\![e^*]\!](\rho \triangleleft^{\rho_R}_{\rho_C})(\lambda \epsilon^* . \kappa(\langle \epsilon_0 \rangle \frown \epsilon^*))))$$

Applying κ,

$$\begin{array}{rl} & (\lambda \epsilon_1{}^* \,.\, \pi_1 \epsilon_1 (\epsilon^* \frown \epsilon_1{}^*) \rho_R \psi)(\langle \epsilon_0 \rangle \frown \epsilon^*) \\ = & \pi_1 \epsilon_1 (\epsilon^* \frown (\langle \epsilon_0 \rangle \frown \epsilon^*)) \rho_R \psi \\ = & \pi_1 \epsilon_1 ((\epsilon^* \frown \langle \epsilon_0 \rangle) \frown \epsilon^*) \rho_R \psi \end{array}$$

Hence, the left hand side equals:

$$\begin{array}{l} (\mathbf{sva}\,\mathcal{E})[\![e]\!]\ (\rho \triangleleft^{\rho_R}_{\rho_C})(\mathit{multiple} \\ \quad (\lambda \epsilon_0 \,.\, (\mathbf{sva}\,\mathcal{E}^*)[\![e^*]\!]\ (\rho \triangleleft^{\rho_R}_{\rho_C})\ (\lambda \epsilon^* \,.\, \pi_1 \epsilon_1 ((\epsilon^* \frown \langle \epsilon_0 \rangle) \frown \epsilon^*) \rho_R \psi))) \end{array}$$

Applying the induction hypothesis with $(\epsilon^* \frown \langle \epsilon_0 \rangle)$ in place of ϵ^*, with the other variables unchanged, and letting

$$\pi_3 = (\mathbf{sva}\mathcal{B})[\![C_{args}(e^*, \rho_C, \#\epsilon^* + 1, i, b)]\!]\rho,$$

Lemma 10 shows (for all ϵ_0):

$$(\mathbf{sva}\,\mathcal{E}^*)[\![e^*]\!](\rho \triangleleft^{\rho_R}_{\rho_C})(\lambda \epsilon^* . \pi_1 \epsilon_1 ((\epsilon^* \frown \langle \epsilon_0 \rangle) \frown \epsilon^*) \rho_R \psi) = \pi_3 \epsilon (\epsilon^* \frown \langle \epsilon_0 \rangle) \rho_R \psi$$

Plugging this in this, followed by Lemma 10, then the definition of *push*, and finally the semantics of `push`, we have:

$$\begin{array}{rl} & (\mathbf{sva}\,\mathcal{E})[\![e]\!]\ (\rho \triangleleft^{\rho_R}_{\rho_C})(\mathit{multiple}(\lambda \epsilon_0 \,.\, \pi_3\ \epsilon\ (\epsilon^* \frown \langle \epsilon_0 \rangle)\ \rho_R\ \psi)) \\ = & (\mathbf{sva}\,\mathcal{E})[\![e]\!]\ (\rho \triangleleft^{\rho_R}_{\rho_C})(\mathit{multiple}(\lambda \epsilon_0 \,.\, \pi_3\ \epsilon_0\ (\epsilon^* \frown \langle \epsilon_0 \rangle)\ \rho_R\ \psi)) \\ = & (\mathbf{sva}\,\mathcal{E})[\![e]\!]\ (\rho \triangleleft^{\rho_R}_{\rho_C})(\mathit{multiple}(\lambda \epsilon_0 \,.\, (\mathit{push}\,\pi_3)\ \epsilon_0\ \epsilon^*\ \rho_R\ \psi)) \\ = & (\mathbf{sva}\,\mathcal{E})[\![e]\!]\ (\rho \triangleleft^{\rho_R}_{\rho_C})(\mathit{multiple}(\lambda \epsilon_0\ . \\ & \qquad (\mathbf{sva}\mathcal{B})[\![\langle \mathtt{push} \rangle :: C_{args}(e^*, \rho_C, \#\epsilon^* + 1, i, b)]\!]\rho \\ & \qquad\quad \epsilon_0\ \epsilon^*\ \rho_R\ \psi)) \end{array}$$

We will apply Clause 1 with

$$b = \langle \texttt{push} \rangle :: C_{args}(e^*, \rho_C, \#\epsilon^* + 1, i, b),$$

and thus with

$$\pi_2 = (\mathbf{sva}\mathcal{B})[\![C(e, \rho_C, \#\epsilon^*, i, \langle \texttt{push} \rangle :: C_{args}(e^*, \rho_C, \#\epsilon^*{+}1, i, b))]\!]\rho.$$

Hence,

$$\begin{aligned} & (\mathbf{sva}\,\mathcal{E})[\![e]\!]\ (\rho \triangleleft^{\rho_R}_{\rho_C}) \\ & \quad (\mathit{multiple}(\lambda \epsilon_0 \,.\, (\mathbf{sva}\mathcal{B})[\![\langle \texttt{push} \rangle :: C_{args}(e^*, \rho_C, \#\epsilon^* + 1, i, b)]\!]\rho \\ & \qquad \epsilon_0\ \epsilon^*\ \rho_R\ \psi)) \\ = \quad & \pi_2\ \epsilon\ \epsilon^*\ \rho_R\ \psi \end{aligned}$$

But, by the code for C_{args}, π_2 is in fact equal to

$$(\mathbf{sva}\mathcal{B})[\![C_{args}(e :: e^*, \rho_C, \#\epsilon^*, i, b)]\!],$$

as the latter is the code:

$$C(e, \rho_C, \#\epsilon^*, i, \langle \texttt{push} \rangle :: C_{args}(e^*, \rho_C, \#\epsilon^* + 1, i, b)).$$

∎

3. Operational Semantics: Faithfulness and Refinement

In this section we describe and justify the transition from the denotational method of the Wand-Clinger style proof of the byte code compiler to the operational framework that will be used for the state machine refinement proofs of the succeeding layers.

3.1. Operational Semantics

To equip a language with an operational semantics, we define a particular state machine and a function that maps programs p, possibly with additional parameters $\vec{x}$, to initial states. The machine may compute a computational answer of type $\mathtt{A}$ from this initial state. We will continue to assume that $\mathtt{A}$ is a fixed domain of which $\mathtt{R}$ is a summand. The denotation of p may then be taken to be the function which maps $\vec{x}$ to the unique value that results, if there is a unique value, and to $\bot_{\mathtt{A}}$ otherwise.[1]

[1] For some kinds of non-deterministic state machines this definition would not be subtle enough. However, the state machine we shall use in our proof of faithfulness is deterministic. In a later stage of our work, we will need a state machine which is non-deterministic but "answer deterministic" as described below. An answer deterministic machine may have different computations starting from a particular state, but they all eventually produce the same computational result.

We would expect that only answer deterministic state machines would play a role as faithful operational refinements of a language with a deterministic denotational semantics, as Scheme's essentially is.

We choose one convenient way of pinning down details among many which would suffice. First we define (possibly non-deterministic) *state transition machines*, these are 9-tuples

$$\langle \mathit{states},\ \mathit{inits},\ \mathit{halts},\ \mathit{inputs},\ \mathit{null\text{-}in},\ \mathit{outputs},\ \mathit{null\text{-}out},\ \mathit{acts},\ \mathit{ans}\rangle$$

such that

(a) *states*, *inputs*, and *outputs* are disjoint sets with $\mathit{inits} \subseteq \mathit{states}$, $\mathit{halts} \subseteq \mathit{states}$, $\mathit{null\text{-}in} \in \mathit{inputs}$, and $\mathit{null\text{-}out} \in \mathit{outputs}$;

(b) $\mathit{acts} \subseteq (\mathit{states} \times \mathit{inputs}) \times (\mathit{states} \times \mathit{outputs})$;

(c) for no pair $\langle s,\ i\rangle$ in the domain of actions is $s \in \mathit{halts}$; and

(d) $\mathit{ans} : \mathit{halts} \rightarrow \mathtt{A}$.

A state s is called a halt state if $s \in \mathit{halts}$, which is equivalent to $\mathit{ans}(s)$ being well defined. If *acts* is actually a function, then the machine is *deterministic.*

Given such a machine M, let C be a non-empty, finite or infinite sequence of elements of $\mathit{states} \times \mathit{inputs} \times \mathit{outputs}$. Then C_S (the *state history*), C_I (the *input history*), and C_O (the *output history*), of C are, respectively, the derived sequences of *states*, *inputs*, and *outputs* components of C. C is a *computation* if $C_S(0) \in \mathit{inits}$, $C_O(0) = \mathit{null\text{-}out}$, and, for every i such that $C(i+1)$ is defined,

$$\langle\langle C_S(i), C_I(i)\rangle,\ \langle C_S(i+1), C_O(i+1)\rangle\rangle \in \mathit{acts}.$$

A transition takes a pair of a current state and a current input and produces a next state and a next output. A state is *accessible* if it is $C_S(i)$ for some computation C and integer i.

Call a state-input pair *proceedable* if it is in the domain of *acts*. A computation is *maximal* if it is infinite or its last state and last input comprise a non-proceedable pair. A finite, maximal computation is *successful* if its last state is a halt state; other finite, maximal computations are *erroneous.*

A computation is *inputless* if every element of its input history is null; it is *outputless* if every element of its output history is null; it is *pure* if it is both inputless and outputless. We will focus on pure computations, partly for simplicity, and partly because the official Scheme semantics does not impose a particular treatment of I/O on us. In the case of a pure computation, we will not distinguish between the computation C and the derived sequence of states C_S, as the former contains no real additional information.

3.2. Proving Faithfulness

The VLISP faithfulness proof works at the level of the tabular byte code TBC. This differs from the BBC only in that constants, embedded closures obtained from lambda expressions, and global variables have been gathered, along with disambiguating tags, into a sequence of entries (each of syntactic type d) called a "template table" (of syntactic type e). An index into the template table appears in place of their occurrences in the byte code.

The syntax of the TBC is presented in Table 12. The tokens of the TBC are the same as those of the BBC, with the addition of `constant`, and `global-variable`, and with `lap` being replaced by `template`. The defining productions are very similar to the ones given for BBC, but note that:

- in a template, the block now precedes an important table, instead of following an unimportant constant, and
- `literal`, `closure`, `global`, and `set-global!` take integer arguments here.

We will use n-like variables for natural numbers, i-like variables for identifiers, and c-like variables for constants.

3.2.1. *Denotational Semantics of the Tabular Byte Code*

The denotational semantics of the TBC, which is given in full in [9], is very similar to the semantics of the BBC. The main difference is that the semantic functions take an additional parameter—namely the template table e to be used—and the semantics of instructions that use the template table make reference to it. We expect the relevant template table entry to be of one of the forms:

$$\langle \texttt{constant}\ c\rangle,\ \langle \texttt{global-variable}\ i\rangle,\ \text{or}\ \langle \texttt{template}\ b\ e'\rangle.$$

Naturally, in the first two cases, it is the value c or i that interests us, which we extract by applying the entry (a sequence) to the argument 1. The clause for closures takes the whole entry, which is an embedded template. The relevant instructions are shown in Table 13.

3.2.2. *Operational Semantics of the Tabular Byte Code*

The operational semantics of the TBC, which is given in full in [10], is determined by a state machine. That state machine uses, as the components of its states, objects from a syntax extending the TBC; we will call this the *augmented byte code* or ABC.

The new ABC tokens are locations (which are actually the same as natural numbers, but which for clarity we indicate by l-like variables when used as

$$
\begin{array}{lll}
z & ::= & \langle \texttt{unless-false}\ y_1\ y_2 \rangle \\
 & \mid & \langle \texttt{literal}\ n \rangle \mid \langle \texttt{closure}\ n \rangle \\
 & \mid & \langle \texttt{global}\ n \rangle \mid \langle \texttt{local}\ n_1\ n_2 \rangle \\
 & \mid & \langle \texttt{set-global!}\ n \rangle \mid \langle \texttt{set-local!}\ n_1\ n_2 \rangle \\
 & \mid & \langle \texttt{push} \rangle \mid \langle \texttt{make-env}\ n \rangle \\
 & \mid & \langle \texttt{make-rest-list}\ n \rangle \mid \langle \texttt{unspecified} \rangle \\
 & \mid & \langle \texttt{checkargs=}\ n \rangle \mid \langle \texttt{checkargs>=}\ n \rangle \\
 & \mid & \langle i \rangle \\
m & ::= & z \mid \langle \texttt{return} \rangle \mid \langle \texttt{call}\ n \rangle \mid \langle \texttt{unless-false}\ b_1\ b_2 \rangle \\
 & \mid & \langle \texttt{make-cont}\ w_1\ n \rangle \\
b & ::= & \langle \langle \texttt{return} \rangle \rangle \mid \langle \langle \texttt{call}\ n \rangle \rangle \mid \langle \langle \texttt{unless-false}\ b_1\ b_2 \rangle \rangle \\
 & \mid & \langle \texttt{make-cont}\ b_1\ n \rangle :: b_2 \mid z :: b_1 \\
y & ::= & \langle \texttt{make-cont}\ y_1\ n \rangle :: b \mid \langle \texttt{make-cont}\ \langle\rangle\ n \rangle :: b \mid z :: y_1 \mid \langle z \rangle \\
w & ::= & b \mid y \\
d & ::= & \langle \texttt{constant}\ c \rangle \mid \langle \texttt{global-variable}\ i \rangle \mid t \\
e & ::= & d^* \\
t & ::= & \langle \texttt{template}\ b\ e \rangle
\end{array}
$$

Table 12: Grammar for the Tabular Byte Code

$$
\begin{array}{l}
\mathcal{B}_\tau : b \cup \{\langle\rangle\} \to e \to \texttt{U} \to \texttt{P} \\
\mathcal{Z}_\tau : z \to e \to \texttt{U} \to \texttt{P} \to \texttt{P} \\
\mathcal{Y}_\tau : y \cup \{\langle\rangle\} \to \texttt{U} \to \texttt{P} \to \texttt{P} \qquad \text{(The variable } y' \text{ ranges over } y \cup \{\langle\rangle\}) \\
\mathcal{T}_\tau : t \to \texttt{U} \to \texttt{P} \\
\\
\mathcal{T}_\tau[\![\langle \texttt{template}\ b\ e \rangle]\!] = \mathcal{B}_\tau[\![b]\!]e \\
\mathcal{Z}_\tau[\![\langle \texttt{literal}\ n \rangle]\!] = \lambda e\rho\ .\ \mathit{literal}\,(\mathcal{K}[\![(e(n))(1)]\!]) \\
\mathcal{Z}_\tau[\![\langle \texttt{closure}\ n \rangle]\!] = \lambda e\rho\ .\ \mathit{closure}\,(\mathcal{T}_\tau[\![e(n)]\!]\rho) \\
\mathcal{Z}_\tau[\![\langle \texttt{global}\ n \rangle]\!] = \lambda e\rho\ .\ \mathit{global}\,(\mathit{lookup}\ \rho\ ((e(n))(1)))
\end{array}
$$

Table 13: Partial Semantics for the Tabular Byte Code

$$
\begin{array}{rcl}
v & ::= & c \mid \langle \text{CLOSURE}\ t\ u\ l \rangle \mid \langle \text{ESCAPE}\ k\ l \rangle \\
 & \mid & \langle \text{MUTABLE-PAIR}\ l_1\ l_2 \rangle \mid \langle \text{STRING}\ l^* \rangle \mid \langle \text{VECTOR}\ l^* \rangle \\
 & \mid & \text{UNSPECIFIED} \mid \text{UNDEFINED} \\
a & ::= & v^* \\
u & ::= & \text{EMPTY-ENV} \mid \langle \text{ENV}\ u\ l^* \rangle \\
k & ::= & \text{HALT} \mid \langle \text{CONT}\ t\ b\ a\ u\ k \rangle \\
s & ::= & v^*
\end{array}
$$

Table 14: ABC Syntactic Definition

locations) and the constructors appearing in small capotals in Table 14. The new ABC syntactic categories, which will be defined by BNF, are

v for (ABC) *values*,
a for (ABC) *argument stacks*
u for (ABC) *environments*,
k for (ABC) *continuations*, and
s for (ABC) *stores*.

All of the categories (templates, blocks etc.) and productions given for TBC are included here. However, although we are extending the syntax, the additional productions do not feed back into the recursive construction of the old categories. Thus, the meanings of the old categories (as sets of tree-like objects) in the ABC are exactly the same as in the TBC. The new productions are given in Table 14.

The states of a byte code state machine are the sequences of the form

$$\langle t,\ b,\ v,\ a,\ u,\ k,\ s \rangle,$$

with the abuse of notation that b is allowed to be $\langle\rangle$. The components of a state are called, in order, its *template*, *code*, *value*, *argument stack*, *environment*, *continuation*, and *store*, and we may informally speak of them as being held in registers. The halt states in this machines are the states such that $b = \langle\rangle$. The answer function *ans* is defined for $s \in \mathit{halts}$ by

$$\mathit{ans}(\langle t,\ \langle\rangle,\ v,\ a,\ u,\ k,\ s \rangle) = v : \mathtt{R}\ \rightarrow\ v \,|\, \mathtt{R},\ \bot_{\mathtt{A}}.$$

Form of the Rules. We present *acts* as the union of subfunctions called (*action*) *rules*. The action rules are functions from disjoint subsets of *states* × *inputs* into *states* × *outputs*. Because their domains are disjoint,

the union is in fact a function, and the resulting machine is a deterministic one.

For each rule we give a name, one or more conditions determining when the rule is applicable (and possibly introducing new locally bound variables for later use), and a specification of the new values of some state components ("registers"). Often the domain is specified by equations giving "the form" of certain registers, especially the code. In all specifications the original values of the various registers are designated by conventional variables used exactly as in the above definition of a state: t, b, v, a, u, k, and s. Call these the original register variables. The new values of the registers are indicated by the same variables with primes attached: t', b', v', a', u', k', and s'. Call these the new register variables. New register variables occur only as the left hand sides of equations specifiying new register values. Registers for which no new value is given are tacitly asserted to remain unchanged. Additionally, we use e for $t(2)$; thus e must always be a template table no matter what the original state is, just because it is a valid state.

It may help to be more precise about the use of local bindings derived from pattern matching. The domain conditions may involve the original register variables and may introduce new variables (other than e and not among the new or old register variables). If we call these new, "auxiliary" variables $x_1, \ldots, x_j$, then the domain conditions define a relation of $j + 7$ places

$$(\dagger)\ R(t, b, v, a, u, k, s, x_1, \ldots, x_j).$$

The domain condition really is this: the rule can be applied in a given state if there exist $x_1, \ldots, x_j$ such that (†). Furthermore, in the change specifications we assume for these auxiliary variables a local binding such that (†). Independence of the new values on the exact choice (if there is any choice) of the local bindings will be unproblematic.

Some Rules of the Tabular Byte Code Machine. We will present the rules connected with `make-cont`, `call`, and `return`. We will refer to all the rules R given in [10] as the ABC rules. Each of them is a partial function from states to states; the domains of the rules are pairwise disjoint; hence, they may be combined to form a unique smallest common extension *acts*.

Rules 1–2	Rules 3–4
Rule 1: **Return-Halt**	Rule 3: **Call**
Domain conditions:	Domain conditions:
$b = \langle\langle\texttt{return}\rangle\rangle$	$b = \langle\langle\texttt{call}\ \#a\rangle\rangle$
$k = \text{HALT}$	$v = \langle\text{CLOSURE}\ t_1\ u_1\ l_1\rangle$
Changes:	$t_1 = \langle\texttt{template}\ b_1\ e_1\rangle$
$b' = \langle\rangle$	Changes:
	$t' = t_1 \quad u' = u_1$
Rule 2: **Return**	$b' = b_1$
Domain conditions:	
$b = \langle\langle\texttt{return}\rangle\rangle$	Rule 4: **Make Continuation**
$k = \langle\text{CONT}\ t_1\ b_1\ a_1\ u_1\ k_1\rangle$	Domain conditions:
Changes:	$b = \langle\texttt{make-cont}\ b_1\ \#a\rangle :: b_2$
$t' = t_1 \quad u' = u_1$	Changes:
$b' = b_1 \quad k' = k_1$	$b' = b_2 \quad a' = \langle\rangle$
$a' = a_1$	$k' = \langle\text{CONT}\ t\ b_1\ a\ u\ k\rangle$

3.2.3. *Faithfulness: Form of the Proof*

The main idea of the faithfulness proof, which is contained in [5], is to associate a denotation in the domain $\mathtt{A}$ with each state. If s is an initial state with template $t = \langle\texttt{template}\ b\ e\rangle$, then the denotation of s agrees with the denotational value of

$$\mathcal{T}_\tau[\![t]\!]\rho\ \mathit{unspecified}\ \langle\rangle\ (\lambda\nu_1\nu_2\,.\,\bot)\ (\mathit{one_arg}\ \kappa_0)$$

where $\kappa_0 = \lambda\epsilon^*\,.\,(\epsilon^*\ 0) : \mathtt{R} \to (\epsilon^*\ 0) \mid \mathtt{R}\ \text{in}\ \mathtt{A}, \bot$. This ensures that the denotation of an initial state containing t matches its value as used in the compiler proof. For a halt state s, the denotation is equal to its computational answer $\mathit{ans}(s)$, which has been defined to be compatible with κ_0.

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 *ans* function is compatible with κ_0, the computational answer matches the denotational answer, so that faithfulness is assured.

Denotational and Operational Answers. We define in this section the notions of the denotational and operational answers associated with an ABC state. Our definition requires the new semantic functions shown in Table 15

$\mathcal{D}_v : v \to \mathtt{E}$
$\mathcal{D}_a : v^* \to \mathtt{E}^*$
$\mathcal{D}_u : u \to \mathtt{N} \to \mathtt{N} \to \mathtt{L}$
$\mathcal{D}_k : k \to \mathtt{E} \to \mathtt{C}$
$\mathcal{D}_s : v^* \to \mathtt{E}^*$

Table 15: Semantic Functions for the Faithfulness Proof

The definitions of the functions take a parameter, namely a function $\rho_G : \text{Ide} \to \mathtt{L}$ known as *globals*. Moreover, for $f : A \to B$, $f^{(*)} : A^* \to B^*$ is defined by

$$f^{(*)}(\langle a_1, \ldots, a_n \rangle) = \langle f(a_1), \ldots, f(a_n) \rangle,$$

and $f^{[*]} : A^* \to B^*$ is defined by

$$f^{[*]}(\langle a_1, \ldots, a_n \rangle) = \langle f(a_n), \ldots, f(a_1) \rangle.$$

The semantic functions are defined by the equations shown in Table 16.

Definition 5 (Denotational and Operational Answer) *The* denotational answer *of an* ABC *state* $\Sigma = \langle t, b, v, a, u, k, s \rangle$, *denoted* $\mathcal{D}[\![\Sigma]\!]$, *is the value of*

$$\mathcal{B}_\tau[\![b]\!] e \rho_G \mathcal{D}_v[\![v]\!] \mathcal{D}_a[\![a]\!] \mathcal{D}_u[\![u]\!] \mathcal{D}_k[\![k]\!] \mathcal{D}_s[\![s]\!].$$

The operational answer *of* Σ, *denoted* $\mathcal{O}[\![\Sigma]\!]$, *is defined inductively to be the smallest partial function such that:*

1. *If* Σ *is a halt state, then* $\mathcal{O}[\![\Sigma]\!] = ans(\Sigma) \quad [= \mathcal{D}_v[\![v]\!] | \mathtt{R}\ in\ \mathtt{A}]$*;*
2. *Otherwise,* $\mathcal{O}[\![\Sigma]\!]$ *is* $\mathcal{O}[\![acts(\Sigma)]\!]$.

Note that, by the definition of $\mathcal{K}$ on $\mathtt{R}$ (see Section 2.1), if $v \in \mathtt{R}$, then $\mathcal{D}_v[\![v]\!] : \mathtt{R}$.

Let $\Sigma = \langle \langle \mathtt{template}\ b_0\ e \rangle, b, v, a, u, k, s \rangle$ be an ABC state. The functions $L_{\text{glo}}, L_{\text{env}}, L_{\text{mp}}, L_{\text{ip}}$, which map ABC states to finite sets of locations, are defined by the following equations:

$$L_{\text{glo}}(\Sigma) = ran(\rho_G).$$

$$L_{\text{env}}(\Sigma) = \{ env\text{-}reference(u', n_0, n_1) : u' \text{ occurs in } \Sigma,\ n_0, n_1 \in \mathtt{N}, \text{ and } env\text{-}reference(u', n_0, n_1) \text{ is defined.} \}.$$

$\mathcal{D}_v[\![c]\!] = \mathcal{K}[\![c]\!]$. $\mathcal{D}_v[\![\langle \text{CLOSURE } t\ u\ l\rangle]\!] = \mathit{fix}(\lambda\epsilon.\langle l, (\lambda\epsilon^*\kappa.\mathcal{T}_\tau[\![t]\!]\rho_G\epsilon\epsilon^*\mathcal{D}_u[\![u]\!](\lambda\epsilon.\kappa\langle\epsilon\rangle))\rangle)$ in $\mathtt{E}$. $\mathcal{D}_v[\![\langle \text{ESCAPE } k\ l\rangle]\!] = \langle l, \mathit{single_arg}(\lambda\epsilon\kappa.\mathcal{D}_k[\![k]\!]\epsilon)\rangle$ in $\mathtt{E}$. $\mathcal{D}_v[\![\langle \text{MUTABLE-PAIR } l_1\ l_2\rangle]\!] = \langle l_1, l_2, \mathit{mutable}\rangle$ in $\mathtt{E}$. $\mathcal{D}_v[\![\langle \text{STRING } l^*\rangle]\!] = \langle l^*, \mathit{mutable}\rangle$ in $\mathtt{E}$. $\mathcal{D}_v[\![\langle \text{VECTOR } l^*\rangle]\!] = \langle l^*, \mathit{mutable}\rangle$ in $\mathtt{E}$. $\mathcal{D}_v[\![\text{NOT-SPECIFIED}]\!] = \mathit{unspecified}$ in $\mathtt{E}$. $\mathcal{D}_v[\![\text{UNDEFINED}]\!] = \mathit{empty}$ in $\mathtt{E}$. $\mathcal{D}_a = \mathcal{D}_v^{[*]}$. $\mathcal{D}_u[\![\text{EMPTY-ENV}]\!] = (\lambda\nu_1\nu_2.\bot)$. $\mathcal{D}_u[\![\langle \text{ENV } u\ l^*\rangle]\!] = \mathit{extend}_R\,\mathcal{D}_u[\![u]\!](\mathit{rev}\ l^*)$. $\mathcal{D}_k[\![\text{HALT}]\!] = (\lambda\epsilon\sigma.\epsilon : \mathtt{R} \to \epsilon\vert\mathtt{R} \text{ in } \mathtt{A}, \bot)$. $\mathcal{D}_k[\![\langle \text{CONT } \langle \mathtt{template}\ b_0\ e\rangle\ b\ a\ u\ k\rangle]\!] = \lambda\epsilon.\mathcal{B}_\tau[\![b]\!]e\rho_G\epsilon\mathcal{D}_a[\![a]\!]\mathcal{D}_u[\![u]\!]\mathcal{D}_k[\![k]\!]$. $\mathcal{D}_s = \mathcal{D}_v^{(*)}$.

Table 16: Semantic Clauses for the Faithfulness Proof

$$L_{\mathrm{mp}}(\Sigma) = \{l_1, l_2 \in \mathtt{L} : \langle \text{MUTABLE-PAIR } l_1\ l_2\rangle \text{ occurs in } \Sigma\}.$$

$$L_{\mathrm{ip}}(\Sigma) = \{l_1, l_2 \in \mathtt{L} : p = \langle \text{IMMUTABLE-PAIR } c_1\ c_2\rangle \text{ occurs } \Sigma \text{ and } (\mathcal{K}[\![p]\!]\vert\mathtt{L} \times \mathtt{L} \times \{\mathit{immutable}\}) = \langle l_1, l_2, \mathit{immutable}\rangle\}.$$

Definition 6 (Normal States, Initial States)
Σ is normal if the following conditions hold:

(1) $L_{\mathrm{glo}}(\Sigma) \cup L_{\mathrm{env}}(\Sigma) \cup L_{\mathrm{ip}}(\Sigma) \cup L_{\mathrm{mp}}(\Sigma) \subseteq \mathit{dom}(s)$.

(2) $L_{\mathrm{glo}}(\Sigma)$, $L_{\mathrm{env}}(\Sigma)$, $L_{\mathrm{ip}}(\Sigma)$, $L_{\mathrm{mp}}(\Sigma)$ *are pairwise disjoint.*

(3) For each $p = \langle \text{IMMUTABLE-PAIR } c_1\ c_2\rangle$ *occurring in* Σ*, there are* $l_1, l_2 \in \mathtt{L}$ *such that* $(\mathcal{K}'[\![p]\!]\vert\mathtt{L} \times \mathtt{L} \times \{\mathit{immutable}\}) = \langle l_1, l_2, \mathit{immutable}\rangle$, $s(l_1) = c_1$*, and* $s(l_2) = c_2$.

A store s is codeless if it contains no value v of the form $\langle \text{CLOSURE } t\ u\ l\rangle$ or the form $\langle \text{ESCAPE } k\ l\rangle$.

A initial state for a TBC *template $t = \langle \mathtt{template}\ b\ e\rangle$ is any normal* ABC *state of the form*

$$\langle t, b, \text{NOT-SPECIFIED}, \langle\rangle, \text{EMPTY-ENV}, \text{HALT}, s\rangle$$

where s is codeless.

The restriction of initial states to those containing codeless stores plays no role in proving faithfulness, but simplifies the proof of the flattener below.

Theorem 12 *The operational semantics for* TBC *is faithful: for all* TBC *templates t and all initial states Σ for t, if $\mathcal{O}[\![\Sigma]\!]$ is defined, then $\mathcal{D}[\![\Sigma]\!] = \mathcal{O}[\![\Sigma]\!]$.*

> **Proof:** The proof is a straightforward induction on the length of the computation sequence determining $\mathcal{O}[\![\Sigma]\!]$. It uses the following lemmas. ∎

Lemma 13 *If Σ is a normal* ABC *state, R is an* ABC *rule, and $\Sigma' = R(\Sigma)$, then Σ' is also a normal* ABC *state.*

The proof examines the individual rules.

Lemma 14 *If $\Sigma = \langle t, \langle\rangle, v, a, u, k, s\rangle$ is a normal* ABC *state with $v \in \mathtt{R}$, then $\mathcal{D}[\![\Sigma]\!] = \mathcal{D}_v[\![v]\!]|\mathtt{R}$ in $\mathtt{A}$.*

The proof is a simple calculation.

Lemma 15 *If Σ and Σ' are normal* ABC *states such that $\mathcal{O}[\![\Sigma]\!]$ is defined and $\Sigma' = R(\Sigma)$ for some* ABC *rule R, then $\mathcal{D}[\![\Sigma]\!] = \mathcal{D}[\![\Sigma']\!]$.*

> **Proof:** (Selected Case) The full proof is a lengthy examination of 34 cases, one for each rule. Let Σ and Σ' be normal ABC states such that $\mathcal{O}[\![\Sigma]\!]$ is defined and $\Sigma' = R(\Sigma)$ for some special rule R. To prove Lemma 15, we must show that, for each special rule R, if Σ satisfies the domain conditions of R, then $\mathcal{D}[\![\Sigma]\!] = \mathcal{D}[\![\Sigma']\!]$.
>
> The individual cases require, almost exclusively, definitional expansion and β-reduction. A typical case concerns the rule for a *return* instruction: *Case 2:* $R =$ Return.
>
> Let $\Sigma = \langle\langle\mathtt{template}\ b\ e\rangle, \langle\langle\mathtt{return}\rangle\rangle, v, a, u, k, s\rangle$,
> where $k = \langle\textsc{cont}\ t_1\ b_1\ a_1\ u_1\ k_1\rangle$ and $t_1 = \langle\mathtt{template}\ b'\ e_1\rangle$.
>
> Then $R(\Sigma) = \Sigma' = \langle t_1, b_1, v, a_1, u_1, k_1, s\rangle$.

$$
\begin{array}{rcl}
\mathcal{D}[\![\Sigma]\!] & = & \mathcal{B}_\tau[\![\langle\langle\mathtt{return}\rangle\rangle]\!]e\rho_G\mathcal{D}_v[\![v]\!]\mathcal{D}_a[\![a]\!]\mathcal{D}_u[\![u]\!]\mathcal{D}_k[\![k]\!]\mathcal{D}_s[\![s]\!] \\
& = & (\lambda e\rho.return)e\rho_G\mathcal{D}_v[\![v]\!]\mathcal{D}_a[\![a]\!]\mathcal{D}_u[\![u]\!]\mathcal{D}_k[\![k]\!]\mathcal{D}_s[\![s]\!] \\
& = & return\mathcal{D}_v[\![v]\!]\mathcal{D}_a[\![a]\!]\mathcal{D}_u[\![u]\!]\mathcal{D}_k[\![k]\!]\mathcal{D}_s[\![s]\!] \\
& = & (\lambda\epsilon\epsilon^*\rho_R\psi.\psi\epsilon)\mathcal{D}_v[\![v]\!]\mathcal{D}_a[\![a]\!]\mathcal{D}_u[\![u]\!]\mathcal{D}_k[\![k]\!]\mathcal{D}_s[\![s]\!] \\
& = & \mathcal{D}_k[\![k]\!]\mathcal{D}_v[\![v]\!]\mathcal{D}_s[\![s]\!] \\
& = & (\lambda\epsilon.\mathcal{B}_\tau[\![b_1]\!]e_1\rho_G\epsilon\mathcal{D}_a[\![a_1]\!]\mathcal{D}_u[\![u_1]\!]\mathcal{D}_k[\![k_1]\!])\mathcal{D}_v[\![v]\!]\mathcal{D}_s[\![s]\!] \\
& = & \mathcal{B}_\tau[\![b_1]\!]e_1\rho_G\mathcal{D}_v[\![v]\!]\mathcal{D}_a[\![a_1]\!]\mathcal{D}_u[\![u_1]\!]\mathcal{D}_k[\![k_1]\!]\mathcal{D}_s[\![s]\!] \\
& = & \mathcal{D}[\![\Sigma']\!]
\end{array}
$$

▮

Although not all of the cases are as straightforward as this (see [5] for all the details), the proof is a perfect illustration of the observation that much formal verification requires long but exquisitely tedious arguments. A proof tool in which it is possible to reason effectively about denotational domains would ease the burden of constructing these proofs, and would increase confidence that no error is hidden within the bulky LaTeX source.[2]

Theorem 12 is a conditional: it asserts on that *if* the machine computes an answer, then that is the correct answer. This property also holds of less useful machines than the TBCSM, for instance a machine whose transition relation is empty. However, the technique of [16, Theorem 6] is available to prove the converse. The converse establishes the adequacy of the operational semantics, asserting that the machine succeeds in computing an answer when the denotational semantics predicts a non-$\perp$, non-erroneous answer.

3.3. Refinement and Storage Layout Relations

In the remainder of this paper, we will show how some state machines *refine* others. Refinement allows us to substitute a more easily implemented state machine (a more "concrete" machine) in place of another (a more "abstract" machine) which we already knew would be acceptable. We will give the definitions in the context of pure computations (those involving neither input nor output), although it is not difficult to modify them for impure computations. We will however state our definitions for non-deterministic machines. The garbage collected state machine is in fact non-deterministic, although it does have the weaker property of *answer determinism*, meaning that any two successful computations that start at the same initial state produce the same answer.

When C is a successful computation, that is, a finite computation terminating with a halt state s, we will write $ans(C)$ to mean $ans(s)$. Otherwise, $ans(C)$ is undefined. We will say that M *can compute* a *from* s if there exists a C such that $C(0) = s$ and $ans(C) = a$.[3]

---

[2] In fact the actual VLISP faithfulness proof as described in [5] is somewhat more complex. The operational semantics and the denotational semantics differ in the order in which they copy the values on the argument stack into the store when making a new environment rib after a procedure call. One order simplified the denotational reasoning, while the other order is more natural to implement. For this reason we have split the faithfulness proof into two parts. The first part has exactly the form described in this section, but uses a state machine with a reversed variant of the Make Environment rule. The second part uses a simple storage layout relation to demonstrate that the machine with the implemented Make Environment rule refines the one used in the first part.

[3] The reader may want to pronounce the relations $\sim_0$, $\sim$, and $\approx$ in this section using

Definition 7 (Refinement) *Suppose that* M^A *and* M^C *are state machines and* $\sim_0 \subseteq inits^C \times inits^A$.

M^C weakly refines M^A via $\sim_0$ *iff, whenever* $s^C \sim_0 s^A$ *and* M^C *can compute* a *from* s^C, *then* M^A *can compute* a *from* s^A.

M^C strongly refines M^A via $\sim_0$ *iff:*

1. *Every abstract initial state corresponds to at least one concrete initial state:*
$$\forall s^A : inits^A \,.\, \exists s^C : inits^C \,.\, s^C \sim_0 s^A$$
2. *Whenever* $s^C \sim_0 s^A$ *and* M^C *can compute* a *from* s^C, *then* M^A *can compute* a *from* s^A;
3. *Whenever* $s^C \sim_0 s^A$ *and* M^A *can compute* a *from* s^A, *then* M^C *can compute* a *from* s^C.

Naturally, weak refinement is an interesting relation between state machines only when *enough* abstract initial states correspond to concrete initial states, and when *enough* abstract computations can be simulated on the concrete machine. As we use weak refinement, these are the states and computations in which all values can be stored in a machine word of fixed width.

Our standard approach [22] for proving that one state machine refines another is to prove that there is a *storage layout relation* between them.

Definition 8 (Storage layout relation) *Suppose that* M^A *and* M^C *are state machines.*

1. *If* $\sim \subseteq states^C \times states^A$, *then the* sequential extension *of* $\sim$, *written here as* $\approx$, *is the relation between sequences of states of* M^C *and* M^A *defined inductively by the conditions:*
   (a) *If* $s^C \sim s^A$, *then* $\langle s^C \rangle \approx \langle s^A \rangle$;
   (b) *If* $\gamma_1 \approx \langle s^A \rangle$ *and* $\gamma_2 \approx \langle s^A \rangle$, *then* $\gamma_1 {}^\frown \gamma_2 \approx \langle s^A \rangle$;
   (c) *If* $\gamma_1 \approx \alpha_1$ *and* $\gamma_2 \approx \alpha_2$, *then* $\gamma_1 {}^\frown \gamma_2 \approx \alpha_1 {}^\frown \alpha_2$.
2. $\sim \subseteq states^C \times states^A$ *is a* storage layout relation *for* M^C *and* M^A *iff:*
   (a) $\forall s^A : inits^A \,.\, \exists s^C : inits^C \,.\, s^C \sim s^A$;

terms such as "refines" or "implements." This motivates the order of the arguments, with the more concrete argument always appearing on the left.

(b) $\forall C^C, s^A \,.\, C^C$ *is finite* $\wedge\, C^C(0) \sim s^A \Rightarrow$
$\exists C^A \,.\, C^A(0) = s^A \wedge C^C \approx C^A$;

(c) $\forall C^A, s^C \,.\, C^A$ *is finite* $\wedge\, C^A(0) \sim s^C \Rightarrow$
$\exists C^C \,.\, C^C(0) = s^C \wedge C^C \approx C^A$.

(d) $\forall s^C, s^A \,.\, s^C \sim s^A \Rightarrow ans^C(s^C){=}{=}ans^A(s^A)$;[4]

3. $\sim \subseteq states^C \times states^A$ *is a* weak storage layout relation *for* M^C *and* M^A *iff conditions 2b and 2d above hold.*

Lemma 16 *The sequential extension of* $\sim$, *restricted to* computations *of* M^C *and* M^A *(rather than arbitrary sequences of states), is the least relation* $\approx$ *satisfying:*

1. *If* s^C *and* s^A *are initial states and* $s^C \sim s^A$, *then* $\langle s^C \rangle \approx \langle s^A \rangle$;
2. *If* $\gamma^\frown\langle s^C_0 \rangle \approx \alpha^\frown\langle s^A \rangle$, s^C_0 *can proceed to* s^C_1, *and* $s^C_1 \sim s^A$, *then* $\gamma^\frown\langle s^C_0, s^C_1 \rangle \approx \alpha^\frown\langle s^A \rangle$;
3. *If* $\gamma^\frown\langle s^C_0 \rangle \approx \alpha^\frown\langle s^A_0 \rangle$, s^C_0 *can proceed to* s^C_1, s^A_0 *can proceed to* s^A_1, *and* $s^C_1 \sim s^A_1$, *then* $\gamma^\frown\langle s^C_0, s^C_1 \rangle \approx \alpha^\frown\langle s^A_0, s^A_1 \rangle$.

Consequently, a relation $\sim$ *satisfying 2a and 2d is a storage layout relation if for all accessible states* s^C_0, s^C_1, *and* s^A_0, *if* $s^C_0 \sim s^A_0$ *and* s^C_0 *can proceed to* s^C_1, *then either*

1. $s^C_1 \sim s^A_0$, *or*
2. *there exists an* s^A_1 *such that* $s^C_1 \sim s^A_1$.

These latter conditions amount to saying, for every pair of similar accessible states, that any transition of the concrete machine corresponds either to a no-op or to a transition of the abstract machine.

Storage layout relations provide a way of proving refinement, as we can easily show with the help of a lemma; it requires only a routine induction on the relation $\approx$:

Lemma 17 *Suppose* $\gamma \approx \alpha$.

1. $\forall i < \#\alpha \,.\, \exists j \,.\, i \leq j \wedge \gamma(j) \sim \alpha(i)$.
2. $\forall j < \#\gamma \,.\, \exists i \,.\, i \leq j \wedge \gamma(j) \sim \alpha(i)$.

[4] We write $s{=}{=}t$ to mean that s and t are equal if either of them is well defined: $(s\downarrow \vee t\downarrow) \Rightarrow s = t$. In particular, $ans^C(s^C){=}{=}ans^A(s^A)$ entails that either both s^C and s^A are halt states, or else neither of them is.

3. $\gamma(0) \sim \alpha(0)$.

Theorem 18 (Storage layout relations guarantee refinement) *Suppose that M^A and M^C are state machines sharing the same inputs, outputs, null-in, and null-out.*

1. *If $\sim$ is a weak storage layout relation for M^C and M^A, then M^C weakly refines M^A via $\sim_0 = \sim \cap \mathit{inits}^C \times \mathit{inits}^A$.*
2. *If $\sim$ is a storage layout relation $\sim$ for M^C and M^A, then M^C strongly refines M^A via $\sim_0 = \sim \cap \mathit{inits}^C \times \mathit{inits}^A$.*

Proof: We prove in turn that each of the three clauses in Definition 7 holds. Note that in proving clause 2 we use only clauses 2b and 2d of Definition 8.

1. If $s^A \in \mathit{inits}$, then $\langle s^A \rangle$ is a computation, and by clause 2c, there is a C^C with $C^C \approx \langle s^A \rangle$. By the definition of $\approx$, $C^C(0) \sim \langle s^A \rangle(0) = s^A$.
2. Suppose that $s^C \sim_0 s^A$; that $C^C(0) = s^C$; and that $\mathit{ans}(C^C) = a$. Then by clause 2b, there is a C^A with $C^A(0) = s^A$ and $C^C \approx C^A$. By Lemma 17, there is an i such that $C^A(i) \sim C^C(\#C^C - 1)$. By clause 2d, $\mathit{ans}(C^A(i)) = a$; moreover, as C^A is a computation and $C^A(i) \in \mathit{halts}$, $\#C^A = i + 1$, so $\mathit{ans}(C^A) = \mathit{ans}(C^A(i))$.
3. Suppose that $s^C \sim_0 s^A$; that $C^A(0) = s^A$; and that $\mathit{ans}(C^A) = a$. Then by clause 2b, there is a C^C with $C^C(0) = s^C$ and $C^C \approx C^A$. By the other half of Lemma 17, there is a j such that $C^C(j) \sim C^A(\#C^A - 1)$. As before, $\mathit{ans}(C^C) = \mathit{ans}(C^C(j)) = a$.

▮

4. The Flattener

The **flattener** [10] transforms byte coded procedures in the TBC into a linear rather than a tree-structured form. Conditionals are represented in a familiar way using `jump` and `jump-if-false` instructions that involve a numeric offset to the target code. Similarly, the target code to which a procedure should return is indicated by a numeric offset from the instruction. We call the output language of the flattener the **flattened byte code** (FBC).

4.1. The Tabular Byte Code in More Detail

In order to describe the operational semantics of conditionals in the TBCSM, we will need to refer to the *open-adjoin* of an open instruction list y and any instruction list w, written $y \smile w$ and defined as follows:

Definition 9 (Open-adjoin)

$$\langle z \rangle \smile w = z :: w;$$

$$(z :: y) \smile w = z :: (y \smile w);$$

$$(\langle \texttt{make-cont}\ y\ n \rangle :: b) \smile w = \langle \texttt{make-cont}\ y \smile w\ n \rangle :: b; \text{ and}$$

$$(\langle \texttt{make-cont}\ \langle\rangle\ n \rangle :: b) \smile w = \langle \texttt{make-cont}\ w\ n \rangle :: b.$$

It may help to note, first, that in the grammar presented in Table 12, a closed TBC instruction list can contain (at top level) no instruction of the form $\langle \texttt{make-cont}\ y\ n \rangle$ or $\langle \texttt{make-cont}\ \langle\rangle\ n \rangle$, and, hence, that an open instruction list is either a list of neutral instructions or contains exactly one instruction of this form. Open-adjoin proceeds recursively inwards from its first argument through the code lists of such instructions until it reaches one which is just a (possibly empty) list of neutral instructions and then it concatenates w to the "open" end of that list.

Lemma 19 Properties of open-adjoin. *For all y, w, y_1, and b,*

1. *$y \smile w$ is well-defined by the above recursion and is an instruction list;*
2. *$y \smile y_1$ is open, and $y \smile b$ is closed; and*
3. *$y \smile (y_1 \smile w) = (y \smile y_1) \smile w$ (associativity).*

The rules for conditionals in open instruction lists use $\smile$:

Rule 5: **Open Branch/True**	Rule 6: **Open Branch/False**
Domain conditions:	Domain conditions:
$b = \langle \texttt{unless-false}\ y_1\ y_2 \rangle :: b_1$	$b = \langle \texttt{unless-false}\ y_1\ y_2 \rangle :: b_1$
$v \neq \mathit{false}$	$v = \mathit{false}$
Changes:	Changes:
$b' = y_1 \smile b_1$	$b' = y_2 \smile b_1$

In the remainder of this section, we will to present the FBC and give it an operational semantics in a style very similar to the presentation of

$$
\begin{array}{rcl}
t & ::= & \langle \texttt{template}\ w\ e \rangle \\
e & ::= & o* \\
o & ::= & \langle \texttt{constant}\ c \rangle \mid \langle \texttt{global-variable}\ i \rangle \mid t \\
w & ::= & \langle\rangle \mid m^\frown w \\
m & ::= & \langle \texttt{call}\ n \rangle \mid \langle \texttt{return} \rangle \mid \langle \texttt{make-cont}\ n_0\ n_1\ n_2 \rangle \mid \langle \texttt{literal}\ n \rangle \\
 & \mid & \langle \texttt{closure}\ n \rangle \mid \langle \texttt{global}\ n \rangle \mid \langle \texttt{local}\ n_0\ n_1 \rangle \mid \langle \texttt{set-global!}\ n \rangle \\
 & \mid & \langle \texttt{set-local!}\ n_0\ n_1 \rangle \mid \langle \texttt{push} \rangle \mid \langle \texttt{make-env}\ n \rangle \\
 & \mid & \langle \texttt{make-rest-list}\ n \rangle \mid \langle \texttt{unspecified} \rangle \mid \langle \texttt{jump}\ n_0\ n_1 \rangle \\
 & \mid & \langle \texttt{jumpf}\ n_0\ n_1 \rangle \mid \langle \texttt{checkargs=}\ n \rangle \mid \langle \texttt{checkargs>=}\ n \rangle \mid \langle i \rangle
\end{array}
$$

Table 17: Flattened Byte Code: Syntax

the TBC semantics in Section 3.2.2. We will then briefly describe the flattener algorithm. To prove the correctness of this algorithm, we develop a correspondence relation between flattened and unflattened code, and prove that the flattener produces code bearing this relation to its input. Finally, we extend the code correspondence relation to a storage layout relation between states of the newly introduced flattened byte code state machine (FBCSM) and the TBCSM. This establishes that the FBCSM is a refinement of the TBCSM.

4.2. Flattened Byte Code Syntax

The Flattened Byte Code (FBC) is a modification of the Tabular Byte Code that uses exclusively unnested linear sequences of tokens for the code part of its templates. Instead of the TBC conditional instruction `unless-false`, FBC has `jumpf` and `jump`, both of which take a pair of numeric operands that together indicate an offset to another location in the instruction sequence. The syntax of `make-cont` has been altered so that it too can use numeric offsets. The BNF definitions of these classes appears as Table 17.

Note that, here in FBC, every member of w is a sequence, of which each element is either a token representing the name of a byte code operation or a number. In the VLISP implementation, these numbers are all small enough unsigned integers to be stored in a single byte (i.e., less than 256).

4.3. Flattened Byte Code Operational Semantics

FBC states differ from TBC states in that they contain a template belonging to the FBC language rather than the TBC language. In addition, instead of having a "code register" they have a numerical "program counter." It is interpreted as an offset into the code sequence of the template.

Officially, the states of an FBC state machine are the sequences of the form

$$\langle t,\ n,\ v,\ a,\ u,\ k,\ s\rangle.$$

We will use the variable w to refer to the code sequence (or just the *code*) of a state, i.e., $w = t(1)$. With our conventions, $w(n)$ is the first element of $w \dagger n$, if $0 \leq n < \#w$. The *halts* are the states such that $n = \#w$ (so $w\dagger n = \langle\rangle$). The answer function, normal states, and initial states are defined as before.

The presentation of the pure rules for the FBC will use the same conventions as were used for the TBC, except that here n stands for the offset of the ingoing state, n' for the offset of the state produced, and w for $t(1)$, the code sequence of the template of the ingoing state. The operator $\oplus$ takes two integers and returns an integer representing an offset they are together coding; it is defined:

$$n_0 \oplus n_1 = (256 * n_0) + n_1.$$

<table>
<tr>
<td>
Rule 1: Call
Domain conditions:
$w\dagger n = \langle \texttt{call}\ \#a\rangle ^\frown w_1$
$v = \langle \text{CLOSURE}\ t_1\ u_1\ l_1\rangle$
Changes:
$t' = t_1$
$n' = 0$
$u' = u_1$

Rule 2: Jumpf/False
Domain conditions:
$w\dagger n =$
$\langle \texttt{jumpf}\ n_1\ n_2\rangle ^\frown w_1$
$v = false$
Changes:
$n' = n + 3 + (n_1 \oplus n_2)$
</td>
<td>
Rule 3: Make Continuation
Domain conditions:
$w\dagger n =$
$\langle \texttt{make-cont}\ n_1\ n_2\ \#a\rangle ^\frown w_1$
Changes:
$n' = n + 4$
$a' = \langle\rangle$
$k' =$
$\langle \text{CONT}\ t\ (n + 4 + (n_1 \oplus n_2))\ a\ u\ k\rangle$

Rule 4: Jumpf/True
Domain conditions:
$w\dagger n = \langle \texttt{jumpf}\ n_1\ n_2\rangle ^\frown w_1$
$v \neq false$
Changes:
$n' = n + 3$
</td>
</tr>
</table>

4.4. The Flattener

The flattener is implemented by a purely applicative Scheme program; it is a straightforward recursive descent. It is however sensitive to the syntactic category of the code that it is flattening, whether it is a *closed* TBC instruction list b or an *open* TBC instruction list y.

An obvious induction shows that the algorithm terminates, and produces a syntactically well-formed FBC template from any TBC template.

We will use $F(w)$ to abbreviate the result of applying `flatten-code` to w and *closed*, if w is of the form b, and for the result of applying `flatten-code` to w and *open*, if w is of the form y. Similarly, we will use $F(e)$ as an abbreviation for the result of applying `flatten-table` to e, and we will use $F(t)$ to mean the result of applying `flatten-template` to t. Observe from the code that $F(\langle \texttt{template}\ b\ e\rangle) = \langle \texttt{template}\ F(b)\ F(e)\rangle$.

4.5. Code Correspondence

We will define the storage layout relation between states justifying this refinement in terms of a preliminary "code correspondence" relation $\simeq$. This is really a four-place relation on a TBC instruction list, a TBC table, a FBC code sequence, and a FBC table. We think of it as a binary correspondence between the pair of its first two arguments and the pair of its last two arguments, meaning that the former pair represents or implements the latter, and we will write, for instance, $(w_F, e_F) \simeq (w, e)$. It is defined as the least relation satisfying the conditions summarized in Table 19. One fine point in the definition concerns expressions of the form $w\dagger n$: when we make an atomic assertion about $w\dagger n$, we are implicitly asserting that it is well defined, so that $n \leq \#w$. We will repeatedly use a fact about sequences:

Lemma 20 drop/adjoin. *When* $n \leq \#w_0$, $(w_0\dagger n)^\frown w_1 = (w_0{}^\frown w_1)\dagger n$.

For *closed* instruction lists, adding code to the end of a flattened version does not destroy the $\simeq$ relation. Since the proofs in this section are all similar, we present only the first in detail.

Lemma 21 $\simeq$/adjoin. *If* $(w_0, e^F) \simeq (b_0, e)$, *then for any* w_1*:*

$$(w_0{}^\frown w_1, e^F) \simeq (b_0, e).$$

Proof: We will assume that w_0 is not of the form $\langle \texttt{jump}\ n_0\ n_1\rangle^\frown w_2$, and thus that $(w_0{}^\frown w_1, e^F) \simeq (b_0, e)$ is not true in virtue of clause 6. For otherwise, $(w_2\dagger(n_0 \oplus n_1), e^F) \simeq (b_0, e)$, and we may apply the

```
(define (flatten-template tem)
  `(template ,(raw-code (flatten-code (cadr tem) 'closed))
             ,(flatten-table (caddr tem))))

(define (flatten-table table)
  (map (lambda (entry) (if (tbc-template? entry)
                           (flatten-template entry)
                           entry))
       table))

(define (flatten-code code category)
  (if (null? code) empty-code-sequence
      (let ((instr (car code)) (after-code (cdr code)))
        (case (operator instr)
          ((make-cont)
           (flatten-make-cont (rand1 instr) (rand2 instr)
                              after-code category))
          ((unless-false) ...)
          (else
           (flatten-default instr after-code category))))))

(define (flatten-make-cont saved nargs after category)
  (let ((after-code (flatten-code after 'closed))
        (saved-code (flatten-code saved category)))
    (add-offset+byte-instruction
     'make-cont (code-length after-code) nargs
     (adjoin-code-sequences after-code saved-code))))

(define (flatten-default instr after category)
  (let ((after-code (flatten-code after category)))
    (prepend-instruction instr after-code)))
```

Table 18: Flattener Algorithm

1. For $m = \langle \texttt{return} \rangle$ or $\langle \texttt{call}\; n \rangle$, $(m ^\frown w_F, e_F) \;\simeq\; (\langle m \rangle, e)$.

2. For atomic z, $(z ^\frown w_F, e_F) \;\simeq\; (z :: m^*, e)$ if, first, either $m^* = w_F = \langle\rangle$ or $(w_F, e_F) \;\simeq\; (m^*, e)$, and second, depending on the form of z:
   (a) if $z = \langle \texttt{literal}\; n \rangle$, $\langle \texttt{global}\; n \rangle$, or $\langle \texttt{set-global!}\; n \rangle$, then $e(n) = e_F(n)$;
   (b) if $z = \langle \texttt{closure}\; n \rangle$, then:
      i. $e(n)$ is of the form $\langle \texttt{template}\; b_1\; e_1 \rangle$, and
      ii. $e_F(n)$ is of the form $\langle \texttt{template}\; w_2\; e_2 \rangle$, where
      iii. $(w_2, e_2) \;\simeq\; (b_1, e_1)$.

3. $(\langle \texttt{make-cont}\; n_0\; n_1\; n \rangle ^\frown w_F, e_F) \;\simeq\; (\langle \texttt{make-cont}\; w\; n \rangle :: b, e)$ if:
   (a) $(w_F, e_F) \;\simeq\; (b, e)$; and
   (b) $(w_F \dagger (n_0 \oplus n_1), e_F) \;\simeq\; (w, e)$.

3′. $(\langle \texttt{make-cont}\; n_0\; n_1\; n \rangle ^\frown w_F, e_F) \;\simeq\; (\langle \texttt{make-cont}\; \langle\rangle\; n \rangle :: b, e)$ if:
   (a) $(w_F, e_F) \;\simeq\; (b, e)$; and
   (b) $\# w_F = n_0 \oplus n_1$.

4. $(\langle \texttt{jumpf}\; n_0\; n_1 \rangle ^\frown w_F, e_F) \;\simeq\; (\langle \langle \texttt{unless-false}\; b_1\; b_2 \rangle \rangle, e)$ if:
   (a) $(w_F, e_F) \;\simeq\; (b_1, e)$; and
   (b) $(w_F \dagger (n_0 \oplus n_1), e_F) \;\simeq\; (b_2, e)$.

5. $(\langle \texttt{jumpf}\; n_0\; n_1 \rangle ^\frown w_F, e_F) \;\simeq\; (\langle \texttt{unless-false}\; y_1\; y_2 \rangle :: w, e)$ if:
   (a) $(w_F, e_F) \;\simeq\; (y_1 {}^\smile w, e)$;
   (b) $(w_F \dagger (n_0 \oplus n_1), e_F) \;\simeq\; (y_2 {}^\smile w, e)$.

6. $(\langle \texttt{jump}\; n_0\; n_1 \rangle ^\frown w_F, e_F) \;\simeq\; (w, e)$ if $(w_F \dagger (n_0 \oplus n_1), e_F) \;\simeq\; (w, e)$.

Table 19: Recursive Conditions for $\simeq$.

lemma[5] to infer that

$$(w_2 \dagger (n_0 \oplus n_1)^\frown w_1, e^F) \;\simeq\; (b_0, e).$$

Since $w_2 \dagger (n_0 \oplus n_1)^\frown w_1 = w_2{}^\frown w_1 \dagger (n_0 \oplus n_1)$, we may apply clause 6 to infer:

$$(\langle \texttt{jump}\; n_0\; n_1 \rangle^\frown (w_2{}^\frown w_1), e^F) \;\simeq\; (b_0, e).$$

By the associativity of $^\frown$, the desired conclusion holds.

The proof is by induction mirroring the inductive definition of $\simeq$.

1. Suppose $b_0 = \langle m \rangle = \langle\langle \texttt{return} \rangle\rangle$ or $\langle\langle \texttt{call}\; n \rangle\rangle$, and $w_0 = \langle m \rangle^\frown w$. By the associativity of $^\frown$, $(\langle m \rangle^\frown w)^\frown w_1 = \langle m \rangle^\frown (w^\frown w_1)$, which is also an instance of clause 1.
2. Suppose that $b_0 = z :: b$, $w_0 = z^\frown w$. (The syntax ensures that m^* really is of the form b.) Since, inductively, $(w^\frown w_1, e_F) \;\simeq\; (b, e)$, we simply apply clause 2 to $w^\frown w_1$. The subclauses dealing with e and e_F are unaffected.
3. If $b_0 = \langle \texttt{make-cont}\; b_1\; n \rangle :: b$, and $w_0 = \langle \texttt{make-cont}\; n_0\; n_1\; n_2 \rangle^\frown w$: Assume inductively:

   $$(w^\frown w_1, e_F) \;\simeq\; (b, e)$$

   and

   $$(w \dagger (n_0 \oplus n_1)^\frown w_1, e_F) \;\simeq\; (b_1, e).$$

   By the drop/adjoin lemma, we may apply clause 3 with $w^\frown w_1$ in place of w.

3′. A closed instruction sequence b_0 is not of this form.

4, 5, 6. Similar to 3.

∎

On the other hand, for *open* instruction lists, adding code to the end of a flattened version corresponds to $^\smile$:

Lemma 22 $\simeq$/open-adjoin. *If $(w_0, e^F) \;\simeq\; (y, e)$ and $(w_1, e^F) \;\simeq\; (w, e)$, then $(w_0{}^\frown w_1, e^F) \;\simeq\; (y^\smile w, e)$.*

Lemma 23 No initial jumps. *$F(w)$ is not of the form $\langle \texttt{jump}\; n_0\; n_1 \rangle^\frown w_1$.*

Theorem 24 Flattener establishes $\simeq$.
Suppose that w respects the template table e.

A. $(F(w), F(e)) \;\simeq\; (w, e)$;

B. *If n occurs in w and $e(n)$ is of the form $\langle \texttt{template}\; b_1\; e_1 \rangle$, then*

$$(F(w_1), F(e_1)) \;\simeq\; (b_1, e_1).$$

[5] Naturally, $w_2 \dagger (n_0 \oplus n_1)$ might begin with a `jump`, but this may be repeated only a finite number of times.

4.6. State Correspondence

Based on the underlying "code correspondence" relation $\simeq$, we will define a binary "miscellaneous correspondence" relation $\sim$ between various kinds of syntactic objects of AFBC on the left and ones of ABC on the right, building up to a notion of correspondence between FBC states and TBC states that is preserved by state transitions. The main result of this section is that the state correspondence is a storage layout relation; hence by Theorem 18 the FBCSM is a faithful refinement of the TBCSM.

We take advantage of the inessential fact that the two languages share some syntactic objects to simplify the definition. Thus some of the syntactic objects that are shared by the two languages are to be called *self corresponding*, namely, the HALT continuation; all environments; UNDEFINED; UNSPECIFIED; all constants; and all values beginning with MUTABLE-PAIR, STRING, or VECTOR.

The relation $\sim$ is then defined recursively, as the least set of pairs whose right element is a member of one of the ABC classes t, v, a, u, k or s, or is an ABC state, and which satisfies the closure conditions given in Table 20. Note especially clauses 6 and 7, which reflect differences in how, and even in which registers, the two machines store "active" code. Also, clause 4 applies to both argument stacks and stores.

From this definition and Theorem 24, it immediately follows that for any initial TBC state $\langle t, t(1), \text{UNSPECIFIED}, \langle\rangle, \text{EMPTY-ENV}, \text{HALT}, s\rangle$, that the FBC initial state $\langle F(t), 0, \text{UNSPECIFIED}, \langle\rangle, \text{EMPTY-ENV}, \text{HALT}, s\rangle$ corresponds to it. Thus condition 2a in Definition 8 (storage layout relations) is satisfied. Moreover, condition 2d follows immediately from clause 8 for $\sim$.

Corollary 25 *$\sim$ satisfies the initial condition (2a) and the halt condition (2d) for being a storage layout relation.*

The following easy lemma shortens the proof of the preservation theorem somewhat and justifies a looser way of thinking about computations of TBC and FBC state machines. One can talk about what they *do* (or *would do* under given circumstances), not just about what they *might* do.

Lemma 26 (Determinacy of Pure Transitions)
If M is either a TBC *machine or an* FBC *machine, and M is in state S, then there is at most one state S' such that M can proceed by one rule transition from S to S'.*

> **Proof:** The domain conditions for the different rules (of the same machine) are pairwise mutually incompatible, as can be seen by looking just at the required form of the code register, except for a few

1. $x \sim x$, if x is self corresponding
2. $\langle \text{CLOSURE } t_1 \ u \ l \rangle \sim \langle \text{CLOSURE } t_2 \ u \ l \rangle$, if $t_1 \sim t_2$
3. $\langle \text{ESCAPE } k_1 \ l \rangle \sim \langle \text{ESCAPE } k_2 \ l \rangle$, if $k_1 \sim k_2$
4. $v_1^* \sim v_2^*$, if $\#v_1^* = \#v_2^*$ and, for every $n < \#v_1^*$, $v_1^*(n) \sim v_2^*(n)$
5. $\langle \texttt{template} \ w \ e_2 \rangle \sim \langle \texttt{template} \ b \ e_1 \rangle$, if $(w, e_2) \ \simeq \ (b, e_1)$
6. $\langle \text{CONT } t_2 \ n \ a_2 \ u \ k_2 \rangle \sim \langle \text{CONT } t_1 \ b \ a_1 \ u \ k_1 \rangle$, if
   (a) $(t_2(1) \dagger n, t_2(2)) \ \simeq \ (b, t_1(2))$, and
   (b) $a_2 \sim a_1$ and $k_2 \sim k_1$
7. $\langle t_2, \ n, \ v_2, \ a_2, \ u, \ k_2, \ s_2 \rangle \sim \langle t_1, \ b, \ v_1, \ a_1, \ u, \ k_1, \ s_1 \rangle$, if
   (a) $(t_2(1) \dagger n, t_2(2)) \ \simeq \ (b, t_1(2))$, and
   (b) $v_2 \sim v_1$, $a_2 \sim a_1$, $k_2 \sim k_1$, and $s_2 \sim s_1$
8. $\langle t_2, \ n, \ v_2, \ a_2, \ u, \ k_2, \ s_2 \rangle \sim \langle t_1, \ b, \ v_1, \ a_1, \ u, \ k_1, \ s_1 \rangle$, if each is a halt state and $v_1 = v_2$.

Table 20: Closure Conditions for $\sim$.

cases when one must also consider the value register or the argument register. Given a particular pure rule and a state, there is always at most one way of binding the variables of the domain equations so that they are satisfied. The next state is clearly determined by the rule, the ingoing state, and these bindings. ∎

Clearly the TBC rules are closely paralleled by the FBC rules – let us say that a TBC rule A *corresponds* to an FBC rule B if A and B have the same name, also that Closed Branch/True and Open Branch/True both *correspond* to Jumpf/True, and finally that Closed Branch/False and Open Branch/False both *correspond* to Jumpf/False. No TBC rule corresponds to Jump. A *branch* rule means one whose name ends in /True or /False – four for TBC and two for FBC.

Theorem 27 (Preservation of State Correspondence)
Let M be a TBC *machine and M_F be a* FBC *machine with the same globals as M. Let S be the state of M and S_F the state of M_F, and assume that $S_F \sim S$. Then*

1. *if M_F proceeds by the Jump rule to state S'_F, then $S'_F \sim S$, and*

2. *if M_F cannot proceed by the Jump rule, then, if either machine proceeds, then the other machine proceeds by a corresponding rule, and the resulting states correspond.*

The proof consists of a careful analysis of the rules of the two machines. Using Lemma 16 and Theorem 18, we have:

Corollary 28 *$\sim$ is a storage layout relation for the* FBCSM *and the* TBCSM. *The* FBCSM *refines the* TBCSM.

5. State Observers and Mutators

Microcode functions are classified into two categories.

State Observers These are functions that return an atomic piece of information about the state. They may be composed when the return type of one state observer matches the type of a parameter to another observer; we will call a (well-formed) expression built from variables, constants, and names for state observers an "observer expression."

State Modifiers These are operations, possibly taking parameters, that modify an atomic component of a state. A "state modifier expression" consists of a state modifier name applied to observer expressions.

We use “modifier” and “mutator” interchangeably. MSPM observers are specified in Table 21 and MSPM mutators are specified in Table 22.

The stores in MSPM accessible states[6] satisfy a representation invariant, namely they contain a collection of *stored objects.* A stored object consists of a header and a tuple of values. The header contains a type tag for the stored object and the length of the tuple of values. The header and the values are stored in consecutive locations in the store. The address at which the first value is stored is called the address of the stored object.

An important property of MSPM-accessible states is that if an accessible state contains a term of the form $\langle \text{PTR}\ \ell \rangle$, then ℓ is the address of a stored object in the store. The representation invariant and this property are established in the correctness proof of the MSPM.

Definition 10 (Stored Objects) *s contains a* stored object *at ℓ if:*

1. *$s(\ell - 1) = \langle \text{HEADER}\ h\ p\ m \rangle$, for some h, p, and m;*
2. *$s(j)$ is a well-defined non-header value for each j for $\ell \leq j < \ell + m$.*

A state Σ is composed of stored objects *if, whenever $\langle \text{PTR}\ \ell \rangle$ occurs in any (transitive) component of Σ, then the store of Σ contains a stored object at location ℓ.*

Action Rules. MSPM action rules are specified as programs composed from state modifier expressions and observer expressions using the constructs `while`, `if-then-else`, and sequential composition. Tests are equations between observer expressions or boolean combinations of equations. The atomic statements are the modifier expressions. The rules are exceedingly simple programs, so there need be no procedure call mechanism. In fact, only the rule to make a `rest`-argument list—when calling a Scheme procedure with a dotted argument list—requires the `while` construct; the others could be expressed using only a primitive recursive `for`-loop. The majority of rules require no iteration whatever.

For our purposes it suffices to regard this language as having a simple Floyd-style [8] operational semantics. To describe the execution of a program, we regard that program as a flow graph. The execution state consists of an MSPM state together with a “program counter” selecting a node in the flow graph, representing the phrase of the program currently being executed. Evaluating a test expression selects between two next nodes; executing a modifier expression alters the MSPM state and advances the current node.

[6] An accessible state is a state in a computation from an initial state.

Let $\Sigma = \langle t, n, v, a, u, k, s, r \rangle$ be an MSPM state. Then,

Register Observers

`template-ref` Σ	$=$	t
`offset-ref` Σ	$=$	n
`value-ref` Σ	$=$	v
`env-ref` Σ	$=$	u
`cont-ref` Σ	$=$	k
`spare-ref` Σ	$=$	r

Stack Observers

`stack-top` Σ	$=$	$a(0) \qquad \text{if } \#a > 0$
`stack-empty?` Σ	$=$	$(\#a = 0)$
`stack-length` Σ	$=$	$\#a$

Stored Object Observers

Let $d = \langle \text{PTR } l \rangle$ and $s(l-1) = \langle \text{HEADER } h\ p\ m \rangle$. Then,

`stob-ref` $d\ i\ \Sigma$	$=$	$s(l+i)$
`stob-tag` $d\ \Sigma$	$=$	h
`stob-mutable?` $d\ \Sigma$	$=$	p
`stob-size` $d\ \Sigma$	$=$	m

Table 21: MSPM State Observers.

Let $\Sigma = \langle t, n, v, a, u, k, s, r \rangle$ be an MSPM state. Then,

Register Mutators

`template-set` $vd\ \Sigma$	$=$	$\Sigma[t' = vd]$	
`offset-set` $i\ \Sigma$	$=$	$\Sigma[n' = i]$	
`value-set` $vd\ \Sigma$	$=$	$\Sigma[v' = vd]$	
`env-set` $vd\ \Sigma$	$=$	$\Sigma[u' = vd]$	
`cont-set` $vd\ \Sigma$	$=$	$\Sigma[k' = vd]$	
`spare-set` $vd\ \Sigma$	$=$	$\Sigma[r' = vd]$	

Stack Mutators

`stack-push` $vd\ \Sigma$	$=$	$\Sigma[a' = vd :: a]$	
`stack-pop` Σ	$=$	$\Sigma[a' = a \dagger 1]$	if $\#a > 0$
`stack-clear` Σ	$=$	$\Sigma[a' = \langle\rangle]$	

Miscellaneous Mutators

`assert` $p\ \Sigma$ $=$ if p then Σ else `abort`

Stored Object Mutators

`make-stob` $h\ p\ m\ \Sigma$ $=$

$$\Sigma[s' = s \frown \langle \text{HEADER}\ h\ p\ m\rangle \frown v_1 \frown \ldots \frown v_m][r' = \langle \text{PTR}\ \#s + 1\rangle]$$

where $v_1 = \ldots = v_m = \langle \text{IMM UNDEFINED}\rangle$

`stob-set` $\langle \text{PTR}\ l\rangle\ i\ vd\ \Sigma$ $=$ $\Sigma[s' = s[(l+i) \mapsto vd]]$
if $s(l-1) = \langle \text{HEADER}\ h\ p\ m\rangle$
and $0 \leq i < m$

Table 22: MSPM State Mutators.

We will refer to the programs in this language as "rule programs." The states in the operational semantics for a rule program P we will call "P-states," so each P-state is a pair consisting of a node in the flow graph for P together with an MSPM state. We will refer to these as the "control node" of the state and the "underlying MSPM state." An initial P-state is a pair whose control node is at the start of P, and whose underlying state may be any MSPM state, while a halt state is one whose control node has reached the end of the program. A computation of the state machine corresponding to P will be a "P-computation." Since rule programs are deterministic, we can freely refer to the maximal computation starting from any initial P-state.

Theorem 29 *The implementation of the action rules of the* MSPM *is correct with respect to the specification of the* SPM *action rules.*

The proof is simple, but highly tedious. A detailed examination of the correctness of all the action rules was carried out.

As with the FBCSM (Lemma 26), the MSPM is determinate: for every state of the MSPM, there is at most one rule that can be applied in that state, and there is at most one next state that the rule may produce.

6. Garbage Collection

In this section we will prove that a garbage collected state machine GCSM is a faithful refinement of the MSPM. The two machines share the same set of states, the same initial states, halt states, and answer function. The only difference between the GCSM and the MSPM is that the GCSM may perform a *garbage collection* before executing the "microcode" mutator `make-stob`, which allocates a new stored object.[7]

We will define a relation $\cong$ of similarity between GCSM states and MSPM states. The initial states of the GCSM are the same as the initial states of the MSPM. Thus, because identity suffices for similarity, the initial state condition 2a for a storage layout relation (Definition 8) is trivial. Moreover, the definition of the answer function on halt states ensures that similar halt states yield identical answers (condition 2d). Thus the proof reduces to showing that execution of the two machines preserves the similarity relation, i.e., that each "microcode" operation, including garbage collection in the GCSM, preserves $\cong$.

[7] In the form of the proof given here, we will not use the fact that garbage collection occurs only immediately before `make-stob`; we shall regard it simply as a new state transition. However, the fact that garbage collection does occur only before this one mutator can be used to justify an optimization to `make-stob`, so that it need not initialize the newly allocated memory to a distinctive value *empty* [20].

The proof is partitioned into two parts. First, we partially specify GCSM, stipulating that garbage collection is some function g from states to states with the property that $g(\Sigma) \cong \Sigma$ (for all accessible states Σ). That the remaining rules preserve $\cong$ establishes that garbage collection is a legitimate implementation strategy, independent of the selection of any particular garbage collection algorithm. Naturally, there are many choices of rules for which this is false, for instance most kinds of pointer arithmetic.

We then complete the specification of GCSM by defining a specific garbage collection algorithm gc. We prove that it satisfies its correctness requirement, which is to say that $gc(\Sigma) \cong \Sigma$ for accessible states Σ.

We define the state similarity relation using a variant of the storage-layout relations notation. The treatment in [22] assumes that terms are tree-structured, and storage layout relations are defined by structural induction over such terms. However, states can effectively contain cycles, since a Scheme program, during execution, can use operations with side-effects to construct circular data structures. A garbage collector must faithfully relocate these data structures also. Thus the storage-layout relations cannot be defined by structural induction over the states regarded as terms. Instead, we use an existential quantifier. Two states[8] s and s^M are similar if *there exists* some correlation between locations in their respective stores, such that corresponding values can be found in correlated locations, as well as in registers. Because we formalize the correlation as a binary predicate relating concrete and abstract store locations, this is a *second-order* existential quantifier.

Our experience is that this technique is intuitively understand and that it leads to straightforward proof obligations. By contrast, although the technique of taking *greatest fixed points* is sometimes suggested for building up a representation relation suited to circular structures, it seems not to be easily applicable to this problem. It is intuitively important that the representation relation between concrete and abstract locations should be one-one on the active parts of the stores, so that an assignment to a concrete location can matched an assignment to the corresponding abstract location. There is, however, generally no single greatest relation $\sim$ between concrete and abstract store locations which is one-one on the active locations. For instance, if different locations in the abstract store contain the same values, then there are arbitrary but incompatible choices about which concrete locations should correspond to each. Moreover, a theory of *maximal* fixed

[8] We will typically indicate objects from the more abstract MSPM by variables with a superscript M, leaving variables representing GCSM objects unadorned. Since the MSPM and GCSM agree in their sets of states, however, as well as in their other components apart from *acts*, the objects involved are generally drawn from the same sets. Rather, the superscript connects the object with the machine we regard it as arising in.

points among one-one relations would be theoretically unfamiliar, and by no means guaranteed to work.

6.1. State Similarity

We formalize locations as natural numbers, but use ℓ-like variables when we are regarding the numbers as store locations.

Definition 11 (Store Correspondence) *When $\sim$ is a binary relation on natural numbers (regarded as representing concrete and abstract store locations), we say that $\sim$ is a* store correspondence between *concrete store* s and *abstract store* s^M *if the following two conditions are met:*

1. $\ell \sim \ell_0^M$ *and* $\ell \sim \ell_1^M$ *implies* $\ell_0^M = \ell_1^M$*;*
2. *if* $\ell \sim \ell^M$ *then* ℓ *and* ℓ^M *are in the domain of* s *and* s^M *respectively.*

An explicit condition extends a store correspondence to a correspondence between values of all kinds:

Definition 12 (Term Correspondence) *Let $\sim$ be a store correspondence between stores s and s^M.*

The term correspondence generated by $\sim$ *is a relation* $\simeq$ *between values* cell *and* cell^M*, defined by taking cases on* cell^M*.* $\text{cell} \simeq \text{cell}^M$ *holds just in case one of the following holds:*

1. $\langle \text{PTR}\ l^M \rangle = \text{cell}^M$ *and*
   - (a) $\langle \text{PTR}\ l \rangle = \text{cell}$*;*
   - (b) $s^M(l^M - 1) = s(l - 1) = \langle \text{HEADER}\ h\ p\ m \rangle$*, for some header type* h*, mutability bit* p *and stored object length* m*; and*
   - (c) *for all* $0 \leq i < m$*,* $(l + i) \sim (l^M + i)$*.*
2. $\langle \text{FIXNUM}\ m \rangle = \text{cell}^M = \text{cell}$*, for some number* m*.*
3. $imm = \text{cell}^M = \text{cell}$*, for some immediate value* imm*.*
4. $b^* = \text{cell}^M = \text{cell}$*, for some sequence of bytes* b^**.*

If x and x^M are not cells, then for convenience we will say that $x \simeq x^M$ just in case $x = x^M$.

Finally, we define the state correspondence to hold if *there exists* a suitable store correspondence:

Definition 13 (State Similarity) *Let $\sim$ be a store correspondence between concrete* GCSM *store s and abstract* MSPM *store s^M, with generated term correspondence $\simeq$. Σ, of the form $\langle t, n, v, a, u, k, s, r\rangle$, and Σ^M, of the form $\langle t^M, n^M, v^M, a^M, u^M, k^M, s^M, r^M\rangle$, are* similar with respect to $\sim$, *written $\Sigma \approx \Sigma^M$, just in case:*

1. $\#a^M = \#a \quad \text{and} \quad \forall 0 \leq i < \#a \,.\, a(i) \simeq a^M(i)$;
2. *for all locations l^M and l,* $\quad l \sim l^M \Rightarrow s(l) \simeq s^M(l^M)$;
3. $x \simeq x^M$, *where x and x^M are any of the pairs of state components $\langle t, t^M\rangle$, $\langle n, n^M\rangle$, $\langle v, v^M\rangle$, $\langle u, u^M\rangle$, $\langle k, k^M\rangle$, and $\langle r, r^M\rangle$.*

Σ and Σ^M are similar, *written $\Sigma \cong \Sigma^M$, just in case there exists a store correspondence $\sim$ such that Σ and Σ^M are similar with respect to $\sim$.*

We will slightly abuse notation by sometimes not showing $\sim$ when using $\simeq$ and $\approx$. When both $\simeq$ and $\approx$ are used in the same assertion, then $\simeq$ is the term correspondence generated by the omitted $\sim$. By inspecting the structure of states we can verify:

Lemma 30 (Properties of $\cong$)

1. $\Sigma \cong \Sigma$;
2. $\Sigma \cong \Sigma^M \Rightarrow \Sigma \in \mathit{halts}$ *iff* $\Sigma^M \in \mathit{halts}$;
3. $\Sigma \cong \Sigma^M \Rightarrow \mathit{ans}(\Sigma) {=}{=} \mathit{ans}(\Sigma^M)$.

Thus, by Lemma 16 and Theorem 18, to show that GCSM refines MSPM it suffices to show that for every pair of similar accessible states, any transition of GCSM corresponds to either a transition of MSPM or a no-op. The garbage collections will correspond to no-ops. The proof itself must show how to extend a given $\sim$ for any transition of the abstract state machine that allocates new memory. And it must also show that the similarity is preserved for transitions that do not allocate new storage. In our proof, the similarity holds with respect to the same $\sim$ in all non-allocating transitions.

6.2. Legitimacy of Garbage Collection

We now *partially specify* the GCSM to be a state machine differing from the MSPM only in the tuple component *acts*. We will use g as a function symbol of type *states* $\rightarrow$ *states*, subject to the (axiomatic) assumptions:

1. $\Sigma \cong \Sigma^M \Rightarrow g(\Sigma) \cong \Sigma^M$;

2. If Σ consists of stored objects, then so does $g(\Sigma)$.

We now define *acts* to be

$$acts^M \cup \{\langle \Sigma, g(\Sigma)\rangle \mid \Sigma \in states\},$$

so that the transitions of GCSM are those of MSPM together with garbage collection steps. We will "instantiate" g in the next section with a particular function *gc*; to do so we will need to establish that axioms 1 and 2 hold of *gc*. This may be regarded as a form of theory interpretation [7].

To prove that $\cong$ is a storage layout relation, we will want to show that every modifier expression preserves it. To do so, we will segregate observers and modifiers into the following lemmas, each of which is proved [20] by perfectly straightforward (but lengthy) applications of the definitions:

Lemma 31 (Observers preserve $\approx$) *Let $e(s,\vec{x})$ be any of the state observers listed in Table 21, taking state parameter s and possibly other parameters $\vec{x}$. Suppose:*

1. $\Sigma \approx \Sigma^M$; and

2. the respective components of $\vec{a}$ and $\vec{a}^M$ satisfy $\simeq$.

Then $e(\Sigma,\vec{a}) \simeq e(\Sigma^M,\vec{a}^M)$.

Lemma 32 (State modifiers preserve $\cong$) *Let $e(s,\vec{x})$ be any of the state modifiers listed in Table 22, taking state parameter s and possibly other parameters $\vec{x}$. Suppose:*

1. $\Sigma \approx \Sigma^M$; and

2. the respective components of $\vec{a}$ and $\vec{a}^M$ satisfy $\simeq$.

Then $e(\Sigma,\vec{a}) \cong e(\Sigma^M,\vec{a}^M)$.

In the proof of Lemma 32, the underlying store correspondence $\sim$ is used unaltered for every mutator except `make-stob`, and in this case the new correspondence $\sim'$ is the extension of $\sim'$ which associates the newly allocated locations.

The following lemma ensures that test expressions in rule programs are invariant with respect to similar states:

Lemma 33 *Suppose:*

1. *$e(s,\vec{x})$ and $e'(s,\vec{y})$ are observer expressions;*
2. *$\Sigma \approx \Sigma^M$; and*
3. *the respective components of $\vec{a}$ and $\vec{a}^M$ satisfy $\simeq$;*
4. *the respective components of $\vec{b}$ and $\vec{b}^M$ satisfy $\simeq$;*

Then $e(\Sigma,\vec{a}) = e'(\Sigma,\vec{b})$ iff $e(\Sigma^M,\vec{a}^M) = e'(\Sigma^M,\vec{b}^M)$.

Proof. By structural induction on observer expressions, using Lemma 31 and the definition of $\approx$. QED.

Lemma 34 (Rule programs $\cong$-invariant) *Suppose that* `P` *is a rule program, and that $\Sigma_0 \cong \Sigma_0^M$.*

1. *Suppose* `P` *terminates in a state Σ_1, when started from Σ_0. Then there exists a state Σ_1^M such that* `P` *terminates in Σ_1^M when started from Σ_0^M and $\Sigma_1 \cong \Sigma_1^M$.*
2. *Conversely, suppose* `P` *terminates in a state Σ_1^M, when started from Σ_0^M. Then there exists a state Σ_1 such that* `P` *terminates in Σ_1 when started from Σ_0 and $\Sigma_1 \cong \Sigma_1^M$.*

Proof: Consider C and C^M, the maximal computations starting with underlying MSPM states Σ_0 and Σ_0^M respectively. We show by induction on i that the control nodes of $C(i)$ and $C^M(i)$ are equal, and that the underlying MSPM states are in $\cong$. The base case is immediate. If the control node of $C(i)$ is a choice point (i.e. an `if` or `while` test), then we apply Lemma 33 to show that the next pair of control nodes are equal; there is no change in the underlying MSPM states. Otherwise, we apply Lemma 32 to show that the next pair of underlying MSPM states remain in $\cong$; there is no choice about the next control node.

So if $C(j)$ is a halt state, then, as only the control node matters, so is $C^M(j)$, and conversely. Moreover, we may take Σ_1^M (for assertion 1) and Σ_1 (for assertion 2) to be the underlying states of $C^M(j)$ and $C(j)$ respectively. ∎

Lemma 35 *Suppose:*

1. *R is any* MSPM *rule with auxiliary variables $\vec{x}$;*
2. *The domain condition for R is satisfied in Σ using the witnessing values $\vec{a}$ in place of $\vec{x}$;*

3. $\Sigma \approx \Sigma^M$;
4. Σ *consists of stored objects.*

Then there is a unique $\vec{a}^M$ *such that:*

1. $\#\vec{a} = \#\vec{a}^M$;
2. $\vec{a}(i) \simeq \vec{a}^M(i)$, *for each* i *where* $0 \leq i < \#\vec{a}$;
3. *The domain condition for* R *is satisfied in* Σ^M *using the witnessing values* $\vec{a}^M$ *in place of* $\vec{x}$.

Proof: Let $\vec{a}^M(i)$ be $\vec{a}(i)$ if the latter is not of the form $\langle \text{PTR } \ell\rangle$. Otherwise, let $\vec{a}^M(i)$ be $\langle \text{PTR } \ell^M\rangle$ where ℓ^M is the unique ℓ' such that $\ell \sim \ell'$.

If this $\vec{a}^M$ is well-defined, then it is clear that conditions 1 and 2 are satisfied, and condition 3 follows from Lemma 33, given that the domain conditions are always equations between observer expressions. Moreover, for non-pointer $\vec{a}(i)$, $\simeq$ requires equality, so there is no alternative candidate. When $\vec{a}(i)$ is a pointer, the definedness of the definite description operator entails that there is no alternative candidate.

Thus, it suffices to show that there is a unique ℓ' such that $\ell \sim \ell'$, whenever $\langle \text{PTR } \ell\rangle$ is a witnessing value for a domain condition auxiliary variable. By Definition 11 clause 1, there is at most one such ℓ'.

Inspecting the domain conditions of the rules, we find three cases:

1. The value of some state register r_0 is $\langle \text{PTR } \ell\rangle$;
2. The value of some state register r_0 is $\langle \text{PTR } \ell_0\rangle$, and $\langle \text{PTR } \ell\rangle$ is a component of the object stored at ℓ_0;
3. The value of some state register r_0 is $\langle \text{PTR } \ell_0\rangle$; $\langle \text{PTR } \ell_1\rangle$ is a component of the object stored at ℓ_0; and $\langle \text{PTR } \ell\rangle$ is a component of the object stored at ℓ_1.

The third case occurs in the rules to reference or set the value of a global variable, as the store location for the variable is determined from the current template table.

In the first case, we use Definition 13 clause 3 to infer that, if v is the value of r_0 in Σ^M, then $\langle \text{PTR } \ell\rangle \simeq v$. By Definition 12, v is of the form $\langle \text{PTR } \ell^M\rangle$, and by clause 1c, $\ell \sim \ell^M$. The remaining cases are similar, except that Definitions 13 and 12 must be used repeatedly. ∎

Theorem 36 (Transition preserves $\cong$) *Suppose that* $\Sigma_0 \cong \Sigma_0^M$ *and that* GCSM *can proceed from* Σ_0 *to* Σ_1. *Then either:*

1. *$\Sigma_1 \cong \Sigma_0^M$ or*
2. *there exists a Σ_1^M such that* MSPM *can proceed from Σ_0^M to Σ_1^M and $\Sigma_1 \cong \Sigma_1^M$.*

Proof: If $\Sigma_1 \cong \Sigma_0^M$, then we're done, so assume otherwise. Since this is not a g-transition, there is an MSPM rule R such that $\langle \Sigma_0, \Sigma_1 \rangle \in R$, and by the determinacy of MSPM, there is just one such R.

Since R is applicable in Σ_0, there is a sequence $\vec{a}$ of witnesses for its auxiliary variables. By Lemma 35, R is also applicable in Σ_0^M.

Thus, we may apply Lemma 34 clause 1 to the rule program P implementing R, which completes the proof. ▮

6.3. The Correctness of a Copying Garbage Collector

In the remainder of this section, we present a specific garbage collection algorithm *gc* and we prove that it satisfies the above property specified of g. The algorithm uses a simple semi-space garbage collection technique where two stores are maintained, one active and one inactive. During garbage collection, "live" data from the active store is copied to the inactive store and then the roles of the two stores are swapped. Live data refers to data that can be accessed either directly or indirectly (via pointers) from the registers (or stack) of the state. The garbage collection algorithm is presented in Figure 2. In this presentation, the new (not-yet-active) store is regarded as the set of locations beyond the end of the old (previously active) store. Thus, if s is the old store, and $\#s \leq \ell$, then $\langle \text{PTR } \ell \rangle$ points at the location $\ell - \#s$ words beyond the beginning of new space. To model the change of active store, we use the auxiliary function *shift*, a partial function from states and integers to states.

Definition 14 (Shift_store, Shift) *If s is a store, then*

$$\text{shift_store}(s,j)(i) == \begin{cases} \langle \text{PTR } \ell - j \rangle & \text{if } s(j+i) = \langle \text{PTR } \ell \rangle \\ s(j+i) & \text{otherwise} \end{cases}$$

If Σ is a state, then $\text{shift}(\Sigma, j)$ *is the result of replacing the store s of Σ by* $\text{shift_store}(s,j)$*, and replacing every component $\langle \text{PTR } \ell \rangle$ of Σ with $\langle \text{PTR } \ell - j \rangle$.*

Then $\mathit{shift}(\Sigma, j)$ is a well-formed state if, whenever $\langle \text{PTR } \ell \rangle$ is a component of Σ, or occurs at a location ℓ_0 in its store, then $j \leq \ell_0$ implies $j \leq \ell$. This condition ensures that a pointer always has a nonnegative argument.

The function *gc-convert* takes a pointer to a stored object in the active store as argument. If the object has not yet been copied from the old store

Let $\Sigma = \langle t, n, v, a, u, k, s, r\rangle$ be a GCSM state. Then,

$$
\begin{array}{l}
\textit{gc-convert}(\langle \text{PTR } l\rangle, s) = \\
\quad \langle\langle \text{PTR } l'\rangle\ s\rangle \quad \text{if } s(l-1) = \langle \text{PTR } l'\rangle \quad \text{(Object already relocated)} \\
\quad \langle\langle \text{PTR } l'\rangle\ s''\rangle \\
\qquad \text{if } s(l-1) = \langle \text{HEADER } h\ p\ m\rangle \\
\qquad \text{and } l' = \#s + 1 \\
\qquad \text{and } s' = s \frown s(l-1) \frown s(l) \frown \ldots \frown s(l+m-1) \\
\qquad \text{and } s'' = s'[l-1 \mapsto \langle \text{PTR } l'\rangle]
\end{array}
$$

$$
\textit{gc-trace}(d, s) \quad = \quad \begin{array}{ll} \textit{gc-convert}(d, s) & \text{if } d \text{ is a pointer} \\ \langle d\ s\rangle & \text{otherwise} \end{array}
$$

$$
\begin{array}{l}
\textit{gc-trace-stack}(a, s) = \langle a'\ s_m\rangle \\
\quad \text{where } \textit{gc-trace}(a(0), s) = \langle a'_0\ s_1\rangle \\
\quad \text{and } \textit{gc-trace}(a(1), s_1) = \langle a'_1\ s_2\rangle \\
\quad \text{and } \textit{gc-trace}(a(m-1), s_{m-1}) = \langle a'_{m-1}\ s_m\rangle \\
\quad \text{and } m = \#a \\
\quad \text{and } a' = \langle a'_0\ a'_1\ \ldots a'_{m-1}\rangle
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textit{gc-scan-heap}(s, m) =} \\
\quad s & \text{if } m = \#s \\
\quad \textit{gc-scan-heap}(s'[m \mapsto d], m+1), & \\
\qquad \text{where } \langle d, s'\rangle = \textit{gc-trace}(s(m), s) & \text{otherwise}
\end{array}
$$

$$
\begin{array}{l}
gc_0(\Sigma, n) = \langle t', n, v', a', u', k', s_7, r\rangle \\
\quad \text{where } \textit{gc-trace}(t, s) = \langle t'\ s_1\rangle \\
\quad \text{and } \textit{gc-trace}(v, s_1) = \langle v'\ s_2\rangle \\
\quad \text{and } \textit{gc-trace}(u, s_2) = \langle u'\ s_3\rangle \\
\quad \text{and } \textit{gc-trace}(k, s_3) = \langle k'\ s_4\rangle \\
\quad \text{and } \textit{gc-trace-stack}(a, s_4) = \langle a'\ s_5\rangle \\
\quad \text{and } \textit{gc-trace}(r, s_5) = \langle r'\ s_6\rangle \\
\quad \text{and } \textit{gc-scan-heap}(s_6, n) = s_7
\end{array}
$$

$$
gc(\Sigma) = \textit{shift}(gc_0(\Sigma, \#s))
$$

Figure 2: Garbage Collection Algorithm

to the new store, it copies the stored object and returns a pointer to the newly allocated stored object. It also replaces the header of the old stored object with a "broken heart" pointing to the new object; all other pointers to the old stored object can be modified to point to the new object. If the object has already been copied to the new store, *gc-convert* uses the broken heart. The result of this function is always a pointer to an object in the new store.

The function *gc-trace* takes an arbitrary value d as argument. If the value d is not a pointer, it returns the argument value d. Otherwise it invokes *gc-convert* on the pointer d and returns the result.

The function *gc-trace-stack* successively calls *gc-trace* on each element of the argument stack a. It accumulates the result values into a new argument stack a'.

The function *gc-scan-heap* successively calls *gc-trace* on each value in the new store. Note that each call to *gc-trace* may potentially extend the new store by copying a stored object to it.

Finally, the function *gc* invokes *gc-trace* on all registers, on all components of the stack and on the store. All resulting pointers reference the new store, so we may apply *shift* to discard the old store.

Since garbage collection is a non-trivial algorithm that transforms the state through a succession of intermediate states, the proof requires the use of a relation similar to $\cong$ to express an invariant maintained by the algorithm. That is, this technique is used to prove that each intermediate GCSM state encountered during garbage collection is similar (by a state correspondence relation $\cong^{gc}$ still to be specified) to the GCSM state prior to garbage collection. Since $\cong^{gc}$ will be defined to coincide with $\cong$ when a store contains no broken hearts, it will follow that garbage collection preserves $\cong$.

Definition 15 (GC Term Correspondence, GC State Similarity)

Let $\sim$ be a store correspondence between s and abstract MSPM *store s'.*

The GC term correspondence generated by $\sim$ *is a relation $\simeq^{gc}$ between values* cell *and* cell$'$, *defined by taking cases on* cell$'$. cell $\simeq$ cell$'$ *holds just in case one of the following holds:*

1. $\langle \text{PTR } l' \rangle = \text{cell}'$ *and, letting $a' = b'$ if $s'(l'-1) = \langle \text{PTR } b' \rangle$ and letting $a' = l'$ otherwise,*

    (a) $\langle \text{PTR } l \rangle = \text{cell}$, *and let $a = b$ if $s(l-1) = \langle \text{PTR } b \rangle$ and $a = l$ otherwise;*

    (b) $s'(a'-1) = s(a-1) = \langle \text{HEADER } h\ p\ m \rangle$, *for some header type h, mutability bit p and stored object length m; and*

   (c) *for all* $0 \leq i < m$, $(a+i) \sim (a'+i)$.

2. $\langle \text{FIXNUM } m \rangle = \text{cell}' = \text{cell}$, *for some number* m.
3. $imm = \text{cell}' = \text{cell}$, *for some immediate value* imm.
4. $b^* = \text{cell}' = \text{cell}$, *for some sequence of bytes* b^*.

Σ *and* Σ', *of the forms* $\langle t, n, v, a, u, k, s, r \rangle$ *and* $\langle t', n', v', a', u', k', s', r' \rangle$, *respectively, are* GC-similar with respect to $\sim$, *written* $\Sigma \approx^{gc} \Sigma'$, *just in case:*

1. $\#a' = \#a \quad and \quad \forall 0 \leq i < \#a \,.\, a(i) \simeq^{gc} a'(i)$;
2. *for all locations* l' *and* l, $\quad l \sim l' \Rightarrow s(l) \simeq^{gc} s'(l')$;
3. $x \simeq^{gc} x'$, *where* x *and* x' *are any of the pairs of state components* $\langle t, t' \rangle$, $\langle n, n' \rangle$, $\langle v, v' \rangle$, $\langle u, u' \rangle$, $\langle k, k' \rangle$, *and* $\langle r, r' \rangle$.

Σ *and* Σ' *are* GC-similar, *written* $\Sigma \cong^{gc} \Sigma'$, *just in case there exists a store correspondence* $\sim$ *such that* Σ *and* Σ' *are GC-similar with respect to* $\sim$.

The proof of the following lemma simply composes the witness store correspondences.

Lemma 37 $\cong^{gc}$ *is a transitive relation. That is,*

$$\Sigma_2 \cong^{gc} \Sigma_1 \text{ and } \Sigma_3 \cong^{gc} \Sigma_2 \Rightarrow \Sigma_3 \cong^{gc} \Sigma_1$$

GC stored objects are like stored objects, except that we may dereference a broken heart for free:

Definition 16 (GC Stored Objects) *Let* $a = b$ *if* $s(\ell - 1) = \langle \text{PTR } b \rangle$ *and let* $a = \ell$ *otherwise. Then* s *contains a* GC stored object *at* ℓ *if:*

1. $s(a-1) = \langle \text{HEADER } h\ p\ m \rangle$, *for some* h, p, *and* m;
2. $s(j)$ *is a well-defined non-header value for each* j *for* $a \leq j < a+m$.

A state Σ *is* composed of GC stored objects *if, whenever* $\langle \text{PTR } \ell \rangle$ *occurs in any (transitive) component of* Σ, *then the store of* Σ *contains a GC stored object at location* ℓ.

Lemma 38 *Suppose that* $\Sigma = \langle t, n, v, a, u, k, s, r \rangle$ *be a state encountered during garbage collection; then* Σ *is composed of GC stored objects.*

Proof: This holds of every accessible GCSM state prior to garbage collection. During garbage collection, all mutations of the state are made via the function *gc-trace* which preserves this property. ∎

We may now discharge the instance of axiom 2 on g (page 61), as instantiated for our garbage collector.

Theorem 39 (GC preserves stored objects) *If* Σ *consists of stored objects and* $\Sigma' = \mathrm{gc}(\Sigma)$*, then* Σ' *consists of stored objects.*

Proof: Since Σ consists of stored objects, it consists (*a fortiori*) of GC stored objects. Using Lemma 38 inductively, the same is true of $gc_0(\Sigma)$. The property of consisting of GC stored objects is preserved by *shift*, so $\Sigma' = \mathit{shift}(gc_0(\Sigma), \#s)$ consists of GC stored objects. Finally, Σ' contains no broken hearts, so it also consists of stored objects. ∎

The following key lemma asserts that the function *gc-trace* respects similarity, i.e., it maps values and states to similar values and states.

Lemma 40 *Let* $\Sigma = \langle t, n, v, a, u, k, s, r\rangle$ *be accessible in garbage collection; let* $\mathrm{gc\text{-}trace}(d, s) = \langle d'\ s'\rangle$ *for some* d', s'*. If* d *is not a pointer, or if* $d = \langle \textsc{ptr}\ l\rangle$ *and* s *contains a GC stored object at* l*, then there exists a store correspondence* $\sim$ *with generated GC term correspondence* $\simeq^{gc}$ *such that*

1. $d' \simeq^{gc} d$
2. $\Sigma \cong^{gc} \langle t, n, v, a, u, k, s', r\rangle$ *with* $\sim$ *as the witness store correspondence.*

Proof: The proof is by cases. We show the crucial case in which d is a pointer $\langle \textsc{ptr}\ l\rangle$ and $s(l-1)$ is a header, rather than a broken heart, so that the object must be copied.

Case: $d = \langle \textsc{ptr}\ l\rangle$ and $s(l-1) = \langle \textsc{header}\ h\ p\ m\rangle$, so, by definition of *gc-trace*, $d' = \langle \textsc{ptr}\ a\rangle$, $a = \#s + 1$, and

$$\begin{array}{l} s' = s[a-1 \mapsto s(l-1)][a \mapsto s(l)]\ldots[a+m-1 \mapsto s(l+m-1)] \\ \qquad [l-1 \mapsto \langle \textsc{ptr}\ a\rangle] \end{array}$$

Define $\sim$ as follows:

$$\begin{array}{rl} a+i \sim l+i & \text{for } 0 \leq i < m \\ x \sim x & \text{for all } x \in \mathit{dom}(s) \text{ such that } s(x) \text{ is not a header} \\ & \text{and } x \neq l+i \text{ for } 0 \leq i < m. \end{array}$$

1. $s(l-1) = \langle \text{HEADER } h\ p\ m \rangle$ by case condition. $s'(a-1) = s(l-1) = \langle \text{HEADER } h\ p\ m \rangle$ by definition of s'. For all $0 \leq i < m$, $a+i \sim l+i$ holds by definition of $\sim$. Thus, by definition of a term correspondence,

$$\langle \text{PTR } a \rangle \simeq^{gc} \langle \text{PTR } l \rangle.$$

2. We show that if vd is a value in one of the registers, or a component of the stack, then $vd \simeq^{gc} vd$.

   If vd is not a pointer, $vd \simeq^{gc} vd$ holds by definition of $\simeq^{gc}$. If $vd = \langle \text{PTR } x \rangle$ where $x \neq l$, then $s(x-1) = s'(x-1) = \langle \text{HEADER } h'\ p'\ m' \rangle$ for some h', p', m'. Further, $x+i \sim x+i$ for all $0 \leq i < m'$. Thus $\langle \text{PTR } x \rangle \simeq^{gc} \langle \text{PTR } x \rangle$ holds by definition of $\simeq^{gc}$.

   If $vd = \langle \text{PTR } l \rangle$, then we know that $s(l-1) = \langle \text{HEADER } h\ p\ m \rangle$ by case condition. Further, $s'(l-1) = \langle \text{PTR } a \rangle$ and $s'(a-1) = \langle \text{HEADER } h\ p\ m \rangle$ by definition of s'. Finally, for all $0 \leq i < m$, $a+i \sim l+i$ holds by definition of $\sim$. Thus, $\langle \text{PTR } l \rangle \simeq^{gc} \langle \text{PTR } l \rangle$ holds by definition of $\simeq^{gc}$.

3. If vd is a component of the store other than a header, then the condition to verify is:

$$\text{for all } x, x',\ x' \sim x \Rightarrow s'(x') \simeq^{gc} s(x)$$

   So assume $x' \sim x$.

   If $x \neq l+i$ for $0 \leq i < m$, then $x = x'$ by definition of $\sim$. Since x is in the domain of $\sim$, $s(x)$ is not a header (by definition of $\sim$). Thus $x \neq l-1$, and so $s'(x') = s'(x) = s(x)$. The result then follows since $vd \simeq^{gc} vd$ was shown above for any component of the state.

   Otherwise, $x = l+i$ for some $0 \leq i < m$. Then, $x' = a+i$ by definition of $\sim$. Now $s'(x') = s'(a+i) = s(l+i) = s(x)$ by definition of s'. The result then follows since $vd \simeq^{gc} vd$ was shown above for any component of the state.

∎

Lemma 41 *Let* $\Sigma = \langle t, n, v, a, u, k, s, r \rangle$*, where* s *consists of stored objects, and let* gc-trace$(s(l), s) = \langle d'\ s' \rangle$*. Then* $\langle t, n, v, a, u, k, s'[l \mapsto d'], r \rangle \cong^{gc} \Sigma$*. Hence, inductively,* gc-scan-heap$(s, l) \cong^{gc} \Sigma$.

Proof: By Lemma 40, $\langle t, n, v, a, u, k, s', r \rangle \cong^{gc} \Sigma$ with $\sim$ as the witness store correspondence (where $\epsilon = \langle s, s' \rangle$). Let $\sim$ extend to the term correspondence $\simeq^{gc}$ where $\alpha = \langle \sim, s, s' \rangle$. Then, by the same lemma, $d' \simeq^{gc} s(l)$.

If $s(l)$ is not a pointer, then by definition of *gc-trace*, $s(l) = s'(l) = d'$. Thus, $s'[l \mapsto d'] = s'$ and so $\langle t, n, v, a, u, k, s'[l \mapsto d'], r \rangle \cong^{gc} \Sigma$.

If $s(l) = \langle \text{PTR } a \rangle$ for some a, then by definition of *gc-trace*, $s(l) = s'(l) = \langle \text{PTR } a \rangle$ since *gc-trace* only modifies $s(a-1)$ which is a header (by lemma 38) and hence $a - 1 \neq l$. Thus, we have that $d' \simeq^{gc} s'(l) = \langle \text{PTR } a \rangle$. We show that

$$\langle t, n, v, a, u, k, s'[l \mapsto d'], r \rangle \cong^{gc} \langle t, n, v, a, u, k, s', r \rangle$$

since the result will then follow by transitivity of $\cong^{gc}$ (lemma 37).

Let $\sim$ be the identity function on all defined locations of s' that are not headers of stored objects, and generate term correspondence $\simeq^{gc}$. We prove that $\sim$ is the witness store correspondence to the above state correspondence.

For all terms vd that are components of the state, $vd \simeq^{gc} vd$ holds. This is immediate for non-pointer values, while for pointer values $\langle \text{PTR } b \rangle$ it depends on $s'(b-1)$ and $s'[l \mapsto d'](b-1)$ being the same header value. This holds since we know that $s'(l)$ is not a header and hence $l \neq b - 1$.

Moreover, $s'[l \mapsto d'](l) \simeq^{gc} s'(l)$, because, as we showed above, $d' \simeq^{gc} s'(l) = \langle \text{PTR } a \rangle$.

Finally, we must show that *gc-scan-heap* terminates. We assume that the original store s consists of stored objects, and that s_1 has the property that if $i < \#s$ and $s(i)$ is not a header, then $s_1(i) = s(i)$. Let $\langle d, s' \rangle = \textit{gc-trace}(s_1(m), s_1)$, let $s'' = s'[m \mapsto d]$, and let $\#s \leq m$. We have:

1. s_1, s', and s'' all contain the same total number of headers;
2. Assuming

   $$\forall n \geq m, \forall l' \,.\, s(n) = \langle \text{PTR } l' \rangle \Rightarrow l' < \#s,$$

   then we may infer

   $$\forall n \geq m + 1, \forall l' \,.\, s''(n) = \langle \text{PTR } l' \rangle \Rightarrow l' < \#s,$$

   using the assumptions that s consists of stored objects and agrees with s_1 on non-headers.
3. If $s'' \neq s_1$, then s'' has one fewer header below $\#s$ than did s_1.

∎

The correctness of gc_0 now follows from Lemmas 37, 40, and 41.

Lemma 42 *Let* $\Sigma = \langle t, n, v, a, u, k, s, r \rangle$ *and* $\Sigma' = \mathrm{gc}_0(\Sigma, \#s)$. *Then* $\Sigma' \cong^{gc} \Sigma$.

We now justify the application of *shift*:

Lemma 43 *Let* $\Sigma = \langle t, n, v, a, u, k, s, r\rangle$ *be a* GCSM *state. If, for some* b, $\langle \text{PTR } b\rangle$ *is a component of* $\text{gc}(\Sigma) = \langle t_1, n_1, v_1, a_1, u_1, k_1, s_1, r_1\rangle$ *(i.e., if one of* t_1, v_1, u_1, k_1, $a_1(i)$, *or* $s_1(i)$ *is the value* $\langle \text{PTR } b\rangle$*) then*

1. $b \geq \#s$;
2. $s_1(b-1) = \langle \text{HEADER } h\ p\ m\rangle$ *for some* h, p, m.

Proof: By definition of *gc*, the state components t, v, u, k, $a(i)$, and $s(i)$ are replaced by the values of *gc-trace* applied to the components.

Let $\Sigma' = \langle t', n', v', a', u', k', s', r'\rangle$ be a state such that if s' contains a header of the form $\langle \text{PTR } b\rangle$, then $b > \#s$ and $s'(b-1) = \langle \text{HEADER } h\ p\ m\rangle$ for some h, p, m. If d' is the value of one of the above mentioned state components of Σ', $\textit{gc-trace}(d', s') = (d'', s'')$, and Σ'' is the state obtained by modifying Σ' with the values d" and s", then we prove that Σ'' also satisfies the above property.

First, we prove that if s'' contains a header of the form $\langle \text{PTR } b\rangle$, then $b > \#s$ and $s'(b-1) = \langle \text{HEADER } h\ p\ m\rangle$ for some h, p, m. If s' also contains the same header $\langle \text{PTR } b\rangle$, this follows by induction hypothesis. Otherwise, by definition of *gc-trace*,

$$s' = (s \frown s(l-1) \frown s(l) \frown \ldots \frown s(l+m-1))[l-1 \mapsto \langle \text{PTR } \#s+1\rangle]$$

where $s(l-1) = \langle \text{PTR } \#s+1\rangle$ is the above mentioned header. The result follows since $\#s+1 > \#s$ and $s''(\#s) = s(l-1) = \langle \text{HEADER } h\ p\ m\rangle$ for some h, p, m (by definition of *gc-trace*).

Next we show that if $d'' = \langle \text{PTR } b\rangle$ for some b, then $b > \#s$ and $s'(b-1) = \langle \text{HEADER } h\ p\ m\rangle$ for some h, p, m. If $\langle \text{PTR } b\rangle$ is a header in s', this follows by induction hypothesis. Otherwise, by definition of *gc-trace*, $b = \#s+1$ and $s''(\#s) = \langle \text{HEADER } h\ p\ m\rangle$ for some h, p, m.

Since s'' and d'' are the only values that differ between Σ'' and Σ', it follows that Σ'' satisfies the desired property. By induction, we have that all intermediate states during garbage collection satisfy the property. But it follows from the definition of *gc-trace* that in such states, if *gc-trace* returns a pointer $\langle \text{PTR } b\rangle$, then $b \geq \#s$ and $s_1(b-1) = \langle \text{HEADER } h\ p\ m\rangle$ for some h, p, m. ∎

Finally, combining Theorem 39 with Lemmas 42 and 43, we have:

Theorem 44 (Garbage Collection preserves $\cong^{gc}$, $\cong$) *If* Σ *consists of stored objects, then*

$$\text{gc}(\Sigma) \cong^{gc} \Sigma$$

and

$$\Sigma \cong \text{gc}(\Sigma).$$

7. The Finite Stored Program Machine

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

The finite stored byte code (FSBC) language is defined to be a strict subset of GSBC. The operational semantics of FSBC is given by a state machine similar to the GCSM. Each microcode transition of the FGCSM is defined to be a restriction of the corresponding microcode transition of the GCSM. That is, if a FGCSM transition is defined on a state, then its value at that state is the same as the corresponding GCSM transition.

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

8. The VLISP Scheme Implementation

VLISP is a comprehensive implementation of Scheme, as is illustrated by the fact that a full bootstrap could be carried out. Moreover, the system has quite acceptable performance. Finally, the verification covers the implementation quite comprehensively.

8.1. The VLISP Bootstrap Process

VLISP is certainly the only verified programming language implementation there is that can be used to bootstrap itself. The possibility of a bootstrap underlines the power of the Scheme source language, as well as the comprehensive character of the VLISP implementation.

There are five source files and two binary files in a VLISP distribution. One binary file is an executable version of the VLISP VM and the other is an image which results from compiling the Scheme to Byte Code compiler. The source files are listed in Table 23. One cycle in the bootstrap process involves using these files to produce a new version of the binary files.

The first action of the VM is to read an image which encodes the initial state of the machine. A new version of the Scheme to Byte Code image is produced using the old image to compile its own source. This new image is used to compile the other Scheme programs, `vps.scm` and `pps.scm`. A new VM is produced by first translating the source for the VM into Pure

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

Table 23: VLISP Source Files

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

Table 24: Sun4 Bootstrap Run Times

PreScheme using the image produced from `vps.scm`, and then into assembly language using the image produced from `pps.scm`. An executable is produced by linking the assembly language with Pure PreScheme's I/O primitives from `prims.c`. Since I/O primitives simply call the operating system, and since we are not in a position to prove anything about the effect of a call to the operating system, we did not consider it worth the trouble to code the I/O primitives directly in assembly language. Apart from compiling these C-coded primitives, the only task of the `gcc` C compiler is to combine the result with the assembly language output `vvm.s`, and to call the assembler.

One cycle in the bootstrap processes is diagrammed in Figure 3. The small circles in the diagram indicate that the virtual machine given by its right arrow is executed. The image used to initialize the machine is indicated by the top arrow. The input is taken from the file named on the left arrow, and the output is written to the file named on the bottom arrow.

Results of the Bootstrap

Table 24 presents the time required for each step when run on a Sun4 IPX, a 25 SPECmark SPARC-based computer.

After the first bootstrap cycle, the assembly source for the VM is unchanged by any succeeding bootstrap cycle; because of a peculiarity in the

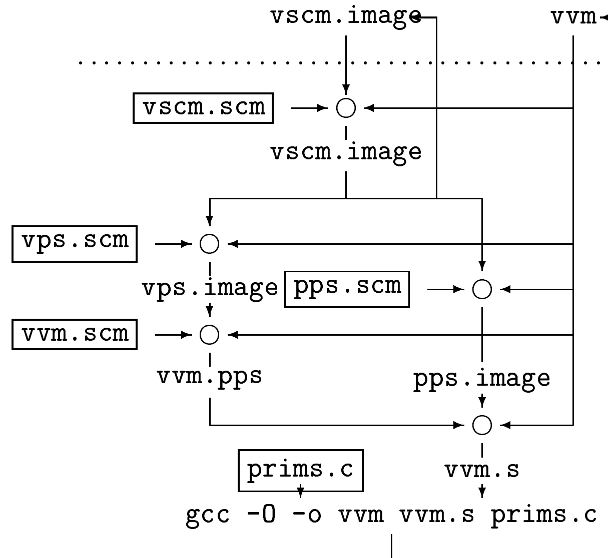

Figure 3: VLISP Bootstrap Process

way symbols are generated, the image of the Scheme to Byte Code compiler is unchanged by a pair of *two* bootstrap cycles.

Two initial versions for the VM were constructed. The Scheme sources for the front end and the Pure PreScheme compiler were compiled using Scheme->C [1]. These programs were used to compile the VLISP VM. A second initial VM was constructed by directly implementing the VM algorithms in C.

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

8.2. VLISP Virtual Machine Performance

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

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

Differences in performance are due to two main sources. First, we structured the virtual machine program to facilitate our verification. In particular, we used abstractions to encapsulate successive refinement layers in the machine. We also have many run-time checks in the code. They ensure that the conditions for application of the formally specified rules are in fact met as the interpreter executes. Second, `gcc` has sophisticated optimizations, for which there is nothing comparable in the Pure PreScheme compiler.

Nevertheless, the ratio of the speed of CVM to the speed of the VLISP virtual machine, when run on a Sun 3, is only 3.9. On the Sun 4, it is 6.3. The ratio is higher on the Sun 4 partly because `gcc` makes very good use of the large Sun 4 register set. Also, `gcc` does instruction scheduling to keep the Sun 4 instruction pipeline full when possible. Finally, some optimizations to primitives in the PreScheme compiler have been implemented in the Sun 3 version, but not yet in the newer Sun 4 version.

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

8.3. Unverified Aspects of VLISP

Although the VLISP verification is quite comprehensive, and covers the great majority of the detailed implementation, we do not call it a complete verification:

- The verification covers algorithms and data structures used, rather than the concrete code. In most cases the relationship is straightforward, and indeed particular coding approaches were often adopted so that the relationship would be transparent. However, sometimes there is an "interesting" gap between the form of the specification and that of the code.

- The VLISP implementation provides all the standard procedures stipulated in the Scheme language definition. However, because no formal specification has been provided for these user-level procedures, we have not had anything to verify these procedures against.

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

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

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

- In some cases, such as the VLISP Pre-Scheme Front End [16, Section 3], and the compiler proof (Section 2.5), only cases that appeared to us to be "representative" or "interesting" were selected for detailed proof.

- Proofs have not been checked using a mechanical theorem prover.

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

8.4. Conclusion.

We believe that the VLISP effort has shown that the rigorous algorithmic verification of substantial programs such as a language implementation is feasible. We have used a relatively small number of techniques which we consider now to be well-defined and broadly reusable.

We consider our decision to carry out the verification at the algorithmic level to have been one key to our success. We also consider the compact and tractable official Scheme semantics to have been another precondition. The third crucial ingredient was using an operational semantics and a state machine refinement approach to the lower levels of the verification. This enabled us to subdivide the complexities of the proof into a succession of intuitively understandable assertions.

These points are developed in more detail in the companion article [11], which also discusses the appropriateness of the different semantic approaches to different parts of the verification.

Acknowledgments. We are deeply indebted to the other participants in the VLISP effort on which this paper is based. They include William Farmer and Leonard Monk of MITRE, as well as Dino Oliva and Mitchell Wand of Northeastern University. Leonard Monk suggested many improvements to this paper.

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.

References

1. J. F. Bartlett. Scheme–>C: A portable Scheme-to-C compiler. WRL 89/1, Digital Equipment Corporation Western Research Laboratory, January 1989.

2. Will Clinger. The Scheme 311 compiler: An exercise in denotational semantics. In *1984 ACM Symposium on Lisp and Functional Programming*, pages 356–364, New York, August 1984. The Association for Computing Machinery, Inc.

3. W. Clingers and J. A. Rees (eds.). Revised^4 report on the algorithmic language Scheme. Technical Report CIS-TR-90-02, University of Oregon, 1990.

4. Paul Curzon. A verified compiler for a structured assembly language. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 253–262. IEEE Computer Society Press, 1991.

5. W. M. Farmer, J. D. Guttman, L. G. Monk, J. D. Ramsdell, and V. Swarup. The faithfulness of the VLISP operational semantics. M 92B093, The MITRE Corporation, September 1992.

6. W. M. Farmer, J. D. Guttman, L. G. Monk, J. D. Ramsdell, and V. Swarup. The VLISP linker. M 92B095, The MITRE Corporation, September 1992.

7. W. M. Farmer, J. D. Guttman, and F. J. Thayer. Little theories. Technical report, The MITRE Corporation, 1992. Submitted to CADE-11.

8. R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

9. J. D. Guttman, L. G. Monk, W. M. Farmer, J. D. Ramsdell, and V. Swarup. The VLISP byte-code compiler. M 92B092, The MITRE Corporation, September 1992.

10. J. D. Guttman, L. G. Monk, W. M. Farmer, J. D. Ramsdell, and V. Swarup. The VLISP flattener. M 92B094, The MITRE Corporation, 1992.

11. J. D. Guttman, J. D. Ramsdell, V. Swarup, D. Oliva, and M. Wand. Results and conclusions of the VLISP verification. Submitted to LASC.

12. IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language.* Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

13. Richard Kelsey and Jonathan Rees. Scheme48 progress report. Submitted to LASC, 1993.

14. R. E. Milne and C. Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall, 1976.

15. J S Moore. Piton: A verified assembly-level language. Technical Report 22, Computational Logic, Inc., Austin, Texas, 1988.

16. D. P Oliva, J. D. Ramsdell, and M. Wand. A verified compiler for VLISP PreScheme. Submitted to LASC.

17. W. Polak. *Compiler Specification and Verification*, volume 124 of *Lecture Notes in Computer Science.* Springer-Verlag, 1981.

18. J. D. Ramsdell, W. M. Farmer, J. D. Guttman, L. G. Monk, and V. Swarup. The VLISP PreScheme front end. M 92B098, The MITRE Corporation, September 1992.

19. Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

20. V. Swarup, W. M. Farmer, J. D. Guttman, L. G. Monk, and J. D. Ramsdell. The VLISP byte-code interpreter. M 92B097, The MITRE Corporation, September 1992.

21. V. Swarup, W. M. Farmer, J. D. Guttman, L. G. Monk, and J. D. Ramsdell. The VLISP image builder. M 92B096, The MITRE Corporation, September 1992.

22. 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.