, , 1–78 ()

The VLISP Verified Scheme System*

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

Abstract. The VLISP project 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 primary 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.

Keywords: Scheme, verified, compiler, interpreter, denotational semantics, operational semantics, refinement, garbage collection

Table of Contents

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

1. Introduction

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

An implementation for a programming language 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 [10], 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 [10]) to execute the resulting byte code programs;
- A second compiler to translate the PreScheme code of the interpreter into assembly language code for a commercial workstation.

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 [12]. An introductory paper [7] discusses the goals and methods of the effort as a whole. The present paper is intended to serve 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, not about concrete program text. We have carefully constructed the actual program text so that it will be a transparently faithful embodiment of the algorithm. Reasoning directly about the concrete program text (under its formal semantics) is considerably more cumbersome. The mass of detail involved in verifying concrete programs frequently obscures the essential reasons why the program is correct.

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. Formal derivations, possibly assisted by mechanical proof tools, might well provide less real insight than mathematical proofs, unless the derivations were accompanied by carefully written, humanly understandable mathematical proofs. We discuss this point further in [7].

1.1. Structure of the VLISP Implementation

The VLISP Scheme implementation derives from Scheme48 [10], 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.

1.1.1. The Compiler Stages

The compiler stages act on source programs in the Scheme language, taking as its input abstract syntax trees in a format only slightly different from the format used in the Scheme Standard [9]. The ultimate output is in a linearized, binary format byte code suitable for execution in the VM. We will refer to the composite of the compiler stages as the "extended compiler."

The extended compiler consists of a sequence of procedures which rewrite the original source program in a succession of intermediate languages. Of these, we will discuss two in this paper:

- The **byte code compiler** [5] rewrites the Scheme source code into a tree-structured byte code based on Clinger's target language [1]. 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 **flattener** [6] uses branch instructions to linearize conditional code and code to be executed after a procedure call returns. We call the output language of the linearizer the **flattened byte code** (FBC).

The other layers omitted from this description do not raise significant new verification issues. Their purpose is to apply other transformations needed to produce a binary image format that we call **Stored Byte Code** or SBC [16]. This is the form that is loaded and executed by the VLISP Virtual Machine.

1.1.2. 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 implementation steps.

These 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.

The 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 are three implementation steps. They involve the following three machines, of which the first provides the operational semantics for the output of the extended compiler, and of which the last is implemented directly using the PreScheme primitives.

- The **stored byte code machine**, or SBCM, which executes the byte code images produced by the compiler stages;

- The **garbage-collected state machine**, or GCSM, which may stop and copy its live data to a fresh heap non-deterministically. The heaps of this state machine (and its predecessor) are of unbounded size, and each location may store an unbounded integer or pointer (among other objects);

- The **finite stored byte code state machine**, or FSBCM, which has heaps of bounded size and has a bound on the width of an integer or pointer.

The first step defines the state transitions of the SBCM in terms of programs that manipulate the atomic components of its state. The second step demonstrates that the GCSM is a faithful refinement of the SBCM. The third step justifies the use of a finite state machine. The FSBCM cannot 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. This is a sort of partial correctness.

Table 1. VLISP Verification Steps

Proof Step	Languages	Machines	Section
Byte Code Compiler	Scheme, BBC	—	2
Faithfulness	BBC	BBCM	3
Flattener	BBC, FBC	BBCM, FBCM	5
Procedural Instantiation	SBC	SBCM	6.2
Garbage Collector	"	SBCM, GCSM	6.3
Finiteness	"	GCSM, FSBCM	6.5

1.2. Structure of the Verification

The full VLISP Scheme verification is a succession of nine subverifications. Five of these justify successive compiler stages. Three justify the implementation steps embedded in the VM. The remaining subverification justifies the switch from the denotational style of semantics to an operational style. It thus provides a link between, on the one hand, the proofs of the first two compiler stages and, on the other hand, the proofs of the remaining compiler stages and all of the VM proofs.

1.2.1. Main Techniques

In this paper we will describe in detail instances of three of the four main techniques sketched in the introductory paper [7]. They include a Wand-Clinger style denotational compiler proof (Section 2); a proof that an operational semantics is faithful to a denotational semantics (Section 3); and two proofs of state machine refinement using storage layout relations (Sections 5 and 6). Of these, the first uses a simple, inductive form of definition to define the storage layout relation. The second, the garbage collection proof, must contend with cyclic structures, and therefore uses a second-order form of definition. The VLISP Scheme implementation does not use transformational compilation, which is discussed in [12].

Not all the steps in the VLISP Scheme verification are discussed in this paper. In particular, the compiler stages include a tabulator, a linker, and an image builder that do not require new techniques.

We have ordered the material to reflect a top-down view of the implementation. Within the VM, the focus will be on the garbage collection proof.

The next five subsections describe the remainder of this paper, with Subsection 1.2.n devoted to Section n of the paper. Table 1 summarizes the main VLISP verification steps discussed in this paper.

1.2.2. The Byte Code Compiler

Section 2 describes the byte code compiler, the first of the compiler stages mentioned in Section 1.1.1. It presents a proof that the byte code compiler preserves the semantics of the Scheme source code. The proof itself is adapted from Clinger's compiler proof [1], which in turn elaborates an approach derived from Wand [17]. In this approach, the source and target languages are both given denotational definitions using the same domains; thus, one can stipulate that the compiler is correct if the denotation of its output code is the same as the denotation of its input.

The denotational semantics for Scheme and for the byte code target language occupy Sections 2.1 and 2.2. We then turn to the main impediment to adapting Clinger's proof. The problem is that the official Scheme semantics makes theoretical provision for procedures that may return multiple values. However, IEEE standard Scheme offers no way of constructing such procedures, and an implementation need not support them, as VLISP does not. This contrast between the official semantics and the expressive resources of the language in its current state required us to develop (in Sections 2.3–2.4) a denotational theory interrelating procedure objects and the single-valued procedures that approximate them.

The section, thus, provides an illustration of "real-life" formal methods, in which the specification to be established may not reach the verifier in a convenient form. We encountered a mismatch between the given specification and the intended implementation and proof technique.

With the alternative semantics in hand, we carry out a Clinger-style proof in 2.5.

1.2.3. Faithfulness of an Operational Semantics

Later portions of the implementation were justified using an operational semantics based on state machines. Section 3 is devoted to justifying the transition from the denotational semantics of Section 2.2 to a corresponding state machine semantics. We show that the state machine semantics is *faithful* to the denotational semantics. By this we mean that when the state machine, starting with a program p, computes a final answer, then that is the answer predicted by the denotational semantics for p. To prove this, we use the denotational semantics to attach an answer to each state, and we prove that the transitions of the state machine leave the answer unchanged. Thus, faithfulness reduces to observing that *initial* states faithfully embed the semantics of the program to be executed, and that *final* states deliver their answers correctly.

The approach is conceptually simple and eminently suited to mechanization. However, it does not establish the converse: namely, when the denotational semantics predicts that p will evaluate to a particular answer, then the state machine actually delivers it. The method of [12, Section 5] can be used to establish this.

1.2.4. State Machine Refinement

The remainder of the paper consists of a sequence of state machine implementation steps. These steps are of two kinds, which we call refinement steps and instantiation steps. Section 4 sets up a general framework for the refinement steps.

A more concrete machine *refines* a more abstract machine when the former can compute the same final answers as the latter, when started in corresponding initial states. This idea is formalized in Definition 9. In order to prove that this relation holds, it is convenient to have a relation between states of the two machines. As the machines compute, the relation should be preserved, and if they reach final states, the relation should entail that the answers achieved are equal. Following [18], we call such a relation a *storage layout relation*. We formalize this notion in Definition 11; we prove in Theorem 6 that a storage layout relation guarantees that refinement holds. The theorem justifies a general method that in [18] is presented only by means of an example.

In some cases we use state machine refinement to justify a translation between two different programming languages. In these cases, we exhibit a storage layout relation $\sim$ between the two state machines that give semantics for the languages. We also show that the translation function F *establishes* the relation $\sim$. Typically, one state component contains the program to be executed, so that establishing the relation $\sim$ means showing that $\Sigma' \sim \Sigma$, where Σ' is the concrete state that results from Σ by replacing its program p by $F(p)$.

An *instantiation step* replaces a declarative specification of a transition relation by a procedural specification of it. Thus, the state space of the machine is unchanged in an instantiation step. In fact, the transition relation may also be left unchanged in an instantiation step, or alternatively the degree of non-determinism in the transition relation may be reduced. What changes is the *form of description* of the transition relation. In place of a declarative description of the transition relation, we substitute a procedural description instead. To justify an instantiation step, we must show that whenever a state Σ is in the domain of the original transition relation *Tr*, then the procedure being introduced, when started in state Σ, terminates in some state Σ', and moreover $Tr(\Sigma, \Sigma')$.

We have not emphasized instantiation as a separate major technique. As we use it, either the justification requires only simple Hoare-style reasoning, as in Section 6.2; or else the justification centers on a relation much like a storage layout relation, as in Section 6.4.

1.2.5. The Flattener

The primary purpose of this section is to illustrate storage layout relations as a way to justify a language translation F. The translation F is the VLISP algorithm for linearizing code.

After some syntactic preliminaries (Section 5.1) we present the operational semantics of the target language in Section 5.2. The flattener algorithm itself is described in Section 5.3. The main content of the assertion that the flattener F establishes $\sim$ is presented in Section 5.4, Theorem 7. In Section 5.5 we then show that $\sim$ really is a storage layout relation.

1.2.6. The Virtual Machine

In Section 6, we describe the implementation steps used within the virtual machine. In this section there are no further changes in the programming language being executed; this is in all cases the binary stored byte code SBC. By contrast, the notion of execution is successively refined.

Section 6.2 shows how to instantiate the SBCM state machine using very simple operations. These operations are divided into *state observers* and *state modifiers*. The state observers are functions that return the value of one ultimate (atomic) constituent of the state; for instance, the value stored at a particular location in the store. State modifiers return an altered state by changing the contents of one atomic ultimate constituent of the state. To implement the declaratively defined "macro-level" instructions of the SBCM, we give a short program that executes observers and modifiers. Very few of these programs require iteration, so the simplest Hoare-style program verification suffices for ensuring them correct.

Section 6.3, on garbage collection, makes up the bulk of the discussion of the VM. There are two main facts to be established. The first, which is the subject of Section 6.3.2, is that garbage collection is a legitimate implementation strategy for the SBCM. There are many state machines for which this would not be the case; for instance, machines that allow pointer arithmetic. To establish the legitimacy of garbage collection for the SBCM, we specify the state machine GCSM, but without defining a specific garbage collection algorithm. We then exhibit a storage layout relation to establish state machine refinement.

The primary interest in this portion lies in how the storage layout relation is defined. The possibility of circular data structures means that the inductive form of definition used in the flattener proof is not available. We use instead a second order existential quantifier to assert the existence of a function correlating store locations within the GCSM and SBCM states.

The second main fact to be established is that a particular copying garbage collection algorithm does indeed respect the storage layout relation, and this instantiation step is the content of Section 6.4.

The brief Section 6.5 justifies the transition to the state machine FSBCM. This state machine has as its states a subset of those of the GCSM, in which state components are of bounded size. Its transition relation is the restriction of the transition relation of GCSM to the smaller set of states. Therefore it is clearly correct as an implementation of the GCSM, at least in the sense that when it computes an answer a, then the GCSM could also compute the same answer.

Table 2. Scheme Abstract Syntax

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

1.3. How to Read This Paper

Many portions of this paper can be read independently. Section 2 can be read on its own; it describes the denotational methods we used in VLISP. Section 3 relies on Sections 2.1–2.2, but not on the remainder of Section 2. Sections 5 and 6 both depend on Section 4, which justifies the storage layout method in general terms. However, their dependence on earlier material—and on each other—is limited to the BNF presentations of syntax.

We have gathered presentations of syntax, declarations of semantic functions and representative semantic clauses, and similar material into tables. For ease of reference, a list of tables appears on page 77 at the end of the paper.

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 [1] and Kelsey and Rees [10]. 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 two-dimensional $\langle over, down\rangle$ lexical addresses for lexical variables, which it distinguishes from global variables.

The compiler algorithm itself is a syntax-directed recursive descent, based on the abstract syntax shown in Table 2. 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:

$\langle$`dotted_lambda` $\langle$`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.

In this and subsequent BNF presentations, we regard the label on the left hand side, such as e and c, both as a typical variable ranging over the syntax class being defined, and also as symbol naming the class, in this case the Scheme expressions and constants, respectively. Thus, we will write the type of a function f taking arguments from e and yielding values in a class C in the form $f : e \rightarrow C$.

The compiler's target language is the Basic Byte Code language (BBC). Expressions of BBC are nested sequences constructed ultimately 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 *templates*, representing procedures, which consist of the keyword `lap` followed by a constant, which annotates the name of the procedure, and the code of the procedure;
- b for *closed instruction lists* or blocks, in which execution eventually encounters a *call* or *return* instruction;
- y for *open instruction lists*, in which execution eventually runs off the end;
- z for *neutral instructions* that may occur in either closed or open instruction lists; and
- w for (general) *instruction lists*, the union of the open and closed lists.

This syntax as defined in Table 3 is 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, and this pattern is crucial to the correctness of the flattener (Section 5). When a conditional expression is the last Scheme expression in the block being compiled, the compiler can emit a closed instruction list in which the code of each branch will contain a `call` or `return` instruction. 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 emits an open instruction list in which neither branch is terminated by a `call` or `return`. Thus, execution "drops into" the following code when the selected branch is completed. The flattener algorithm uses this distinction between open and closed instruction lists systematically. Its semantic significance is summarized in Lemmas 15 and 16.

After taking cases based on the abstract syntax class of the expression being compiled, the compiler algorithm combines the code produced by its recursive calls with some additional instructions. The main dispatch and one typical case, which

Table 3. Grammar for the Basic Byte Code

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

compiles an application, are shown in slightly simplified form in Figure 1. The remaining cases are not presented here. 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," a BBC instruction list. It is the result of a recursive call to the main dispatch to compile the portion of the procedure after the expression currently being compiled. It will be attached at the end of the code generated from the current expression.

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 on its expression argument, so that it is guaranteed to terminate.

Syntactic correctness. If e is a Scheme expression, then $C(e, \rho_C, w)$ 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 ρ_C and w, which are the empty compile-time environment and code consisting of a bare `return` respectively.

THEOREM 1 **(Compiler syntactic correctness)**
For any Scheme expression e, compile-time environemnt ρ_C, BBC closed instruction sequence b_0, and BBC open instruction sequence y_0,

```
(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.

1. *$C(e)$ is a* BBC *template;*

2. *$C(e, \rho_C, b_0)$ is a* BBC *closed instruction sequence;*

3. *$C(e, \rho_C, y_0)$ is a* BBC *open instruction sequence;*

4. *$C(e, \rho_C, \langle\rangle)$ is a* BBC *open instruction sequence.*

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 for semantic correctness. In order to carry out this idea, we will need to describe the formal semantics of the Scheme and of the byte code.

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. See [15] for a description of the underlying approach, and [5] for additional details; we have summarized some notation in Table 4. Apart from a few cosmetic changes of notation, our version differs from the standard in three ways.

- The domains have been made somewhat more concrete. In particular, the domian of locations `L` has been identified with rthe natural numbers `N`, and consequently `S`, the domain of stores, has been identified with `E*`. This formalizes our decision to regard memory as unbounded, at this level of the verification.

- We have removed tests from the semantics to check whether a new storage location can be allocated in `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 have been added. It was needed to prove that an operational semantics for the BBC is faithful to its denotational semantics.

Perhaps the main cosmetic change is that we regard sequences as partial functions from natural numbers to values. To project the ith element from a sequence s, we write the function application $s\,i$. We treat pairs similarly.

Table 4. Some Notation

$\langle \ldots \rangle$	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 $s(\#s - 1) = x$
$a :: s$	prepend (cons) a before s
$p\,0, \quad p\,1$	left and right components (resp) of a pair p of type $\mathtt{D}_0 \times \mathtt{D}_1$
$rev\ s$	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
$p \rightarrow a, b$	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 $\mathtt{D}$
$x \,\vert\, \mathtt{D}$	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 5. Domains for the Semantics of Scheme

$\nu \in \mathtt{N}$			natural numbers
$\alpha \in \mathtt{L}$	$=$	$\mathtt{N}$	locations
$\rho \in \mathtt{U}$	$=$	$i \rightarrow \mathtt{L}$	environments
$\mathtt{T}$	$=$	$\{false,\ true\}$	booleans
$\mathtt{T}_{\mathrm{L}}$	$=$	$\{mutable, immutable\}$	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
$\mathtt{M}$	$=$	$\mathtt{T} + \mathtt{T}_{\mathrm{L}} + \{null,\ empty,\ unspecified\}$	miscellaneous
$\phi \in \mathtt{F}$	$=$	$\mathtt{L} \times (\mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C})$	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
$\theta \in \mathtt{C}$	$=$	$\mathtt{S} \rightarrow \mathtt{A}$	command continuations
$\kappa \in \mathtt{K}$	$=$	$\mathtt{E}^* \rightarrow \mathtt{C}$	expression continuations
$\mathtt{A}$			answers

Table 6. Scheme Semantic Functions

$$\begin{aligned}\mathcal{K} &: c \to \mathtt{E}\\ \mathcal{E} &: e \to \mathtt{U} \to \mathtt{K} \to \mathtt{C}\\ \mathcal{E}* &: e* \to \mathtt{U} \to \mathtt{K} \to \mathtt{C}\\ \mathcal{C} &: \Gamma* \to \mathtt{U} \to \mathtt{C} \to \mathtt{C}\end{aligned}$$

The domains used in the denotational semantics of Scheme are presented in Table 5. Exactly one "domain equation" is actually not an equation. It implicitly introduces an isomorphism $\psi_{\mathtt{E}}$ between $\mathtt{E}$ and a disjoint sum domain, $\mathtt{E}_\Sigma = \mathtt{Q}+\mathtt{H}+\mathtt{R}+\ldots$ 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, but not involving $\mathtt{E}$.

The semantic domains are given in Table 6. They are not disjoint from the syntactic domains of the abstract syntax of Scheme as given in Table 2, because a Scheme constant c may be a number, boolean, ..., or else a list, dotted list, or vector of other constants. These are members of the domains $\mathtt{R}$, $\mathtt{T}$, ..., or else $\mathtt{E}_\mathrm{p}$ or $\mathtt{E}_\mathrm{v}$. In this second group of cases, the Scheme definition stipulates that the pair or vector be immutable, in contrast to the objects produced by calling `cons` or `make-vector`, which may be side-effected.

Semantics of Constants. The semantics of constants, given by a function $\mathcal{K}$, is "intentionally undefined" in [9]. Although we will not define a particular choice of $\mathcal{K}$, we will need to give some constraints that it must satisfy. In essence, we consider its actual value to be a parameter to the semantics of Scheme. We do, however, explicitly define its behavior on primitive constants, i.e. constants not requiring storage at run-time.

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

$\mathcal{K}[\![\textsc{nil}]\!] = \mathit{null}$ in $\mathtt{E}$; $\mathcal{K}[\![\#\textsc{f}]\!] = \mathit{false}$ in $\mathtt{E}$; and $\mathcal{K}[\![\#\textsc{t}]\!] = \mathit{true}$ in $\mathtt{E}$

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

1. 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 $(\epsilon \mid \mathtt{E}_\mathrm{p})\ 2 = \mathit{immutable}$.

   From this it follows that types are correct: if c is a string, vector, or pair, then $\mathcal{K}[\![c]\!] : \mathtt{D}$ where $\mathtt{D}$ is $\mathtt{E}_\mathrm{s}$, $\mathtt{E}_\mathrm{v}$, or $\mathtt{E}_\mathrm{p}$, respectively.

2. Lengths are correct:

   (A) Suppose c is a vector of length n, and $\mathcal{K}[\![c]\!]$ is a non-bottom value ϵ; then the length of its left component is n, i.e. $\#((\epsilon \mid \mathtt{E}_\mathrm{v})\ 0) = n$.

Table 7. Scheme Semantics: Some Semantic Clauses

$$
\begin{array}{l}
\mathcal{E}[\![e_0 :: e^*]\!] = \\
\quad \lambda\rho\kappa\,.\,\mathcal{E}^*(\mathit{permute}(\langle e_0\rangle \frown e^*)) \\
\qquad\qquad\quad \rho \\
\qquad\qquad\quad (\lambda\epsilon^*\,.\,((\lambda\epsilon^*\,.\,\mathit{applicate}\,(\epsilon^*\,0)\,(\epsilon^* \dagger 1)\,\kappa) \\
\qquad\qquad\qquad\qquad (\mathit{unpermute}\,\epsilon^*))) \\
\\
\mathcal{E}^*[\![\langle\rangle]\!] = \lambda\rho\kappa\,.\,\kappa\langle\,\rangle \\
\\
\mathcal{E}^*[\![e_0 :: e^*]\!] = \\
\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![e_0]\!]\,\rho\,(\mathit{single}\,(\lambda\epsilon_0\,.\,\mathcal{E}^*[\![e^*]\!]\,\rho\,(\lambda\epsilon^*\,.\,\kappa\,(\langle\epsilon_0\rangle \frown \epsilon^*)))) \\
\\
\mathit{single} : (\mathtt{E} \to \mathtt{C}) \to \mathtt{K} \\
\mathit{single} = \\
\quad \lambda\psi\epsilon^*\,.\,\#\epsilon^* = 1 \to \psi(\epsilon^*\,0), \\
\qquad\qquad\quad \mathit{wrong}\ \text{“wrong number of return values”} \\
\\
\mathit{applicate} : \mathtt{E} \to \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \\
\mathit{applicate} = \\
\quad \lambda\epsilon\epsilon^*\kappa\,.\,\epsilon : \mathtt{F} \to ((\epsilon \mid \mathtt{F})\,1)\epsilon^*\kappa, \mathit{wrong}\ \text{“bad procedure”}
\end{array}
$$

(B) Suppose c is a string of length n, and $\mathcal{K}[\![c]\!]$ is a non-bottom value ϵ; then $\#((\epsilon \mid \mathtt{E}_\mathrm{s})\,0) = n$.

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.

The syntax of constants, together with the type correctness condition 1, ensures 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 pairs and vectors.

Semantics of Expressions. In Table 7, we give the clauses—almost unchanged from [9]—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 [5].

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

Table 8. Byte Code Semantics: Additional Domains

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

Table 9. Byte Code Semantic Functions

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

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

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 the contents of 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 8. Semantic functions appear in Table 9, and representative semantic clauses stated in terms of these semantic functions are presented in Table 10. Together with similar clauses, they specify a semantics for the BBC.

The $\mathcal{B}[\![\langle\rangle]\!]$ clause for a null code sequence defines the "answer function," that is, the way that a computational answer in $\mathtt{A}$ 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 11) follow the Wand-Clinger style of byte code specification. In each case, the auxiliary function takes as arguments:

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

Table 10. Some Byte Code Semantic Clauses

$$\mathcal{T}[\![\langle \texttt{lap}\ c\ b\rangle]\!] = \mathcal{B}[\![b]\!]$$

$$\mathcal{B}[\![\langle\rangle]\!] = \lambda\rho\epsilon\epsilon^*\rho_R\psi\sigma\ .\ \epsilon : \texttt{R} \rightarrow \epsilon \mid \texttt{R}\ \text{in}\ \texttt{A}, \bot$$
$$\mathcal{B}[\![z :: b]\!] = \lambda\rho\ .\ \mathcal{Z}[\![z]\!]\rho\,(\mathcal{B}[\![b]\!]\rho)$$
$$\mathcal{B}[\![\langle \texttt{make-cont}\ b_1\ n\rangle :: b_2]\!] = \lambda\rho\ .\ \mathit{make\_cont}\,(\mathcal{B}[\![b_1]\!]\rho)n(\mathcal{B}[\![b_2]\!]\rho)$$
$$\mathcal{B}[\![\langle\langle \texttt{return}\rangle\rangle]\!] = \lambda\rho\ .\ \mathit{return}$$
$$\mathcal{B}[\![\langle\langle \texttt{call}\ n\rangle\rangle]\!] = \lambda\rho\ .\ \mathit{call}\,n$$

$$\mathcal{Y}[\![\langle \texttt{make-cont}\ y'\ n\rangle :: b]\!] = \lambda\rho\pi\ .\ \mathit{make\_cont}\,(\mathcal{Y}[\![y']\!]\rho\pi)\ n\ (\mathcal{B}[\![b]\!]\rho)$$
$$\mathcal{Y}[\![z :: y']\!] = \lambda\rho\pi\ .\ \mathcal{Z}[\![z]\!]\rho\,(\mathcal{Y}[\![y']\!]\rho\pi)$$
$$\mathcal{Y}[\![\langle\rangle]\!] = \lambda\rho\pi\ .\ \pi$$

$$\mathcal{Z}[\![\langle \text{push}\rangle]\!] = \lambda\rho\ .\ \mathit{push}$$

Table 11. Some Byte Code Auxiliary Functions

$$\mathit{make\_cont} : \texttt{P} \rightarrow \texttt{N} \rightarrow \texttt{P} \rightarrow \texttt{P}$$
$$\mathit{make\_cont} =$$
$$\lambda\pi'\nu\pi\ .\ \lambda\epsilon\epsilon^*\rho_R\psi\ .\ \#\epsilon^* = \nu \rightarrow \pi\epsilon\langle\rangle\rho_R(\lambda\epsilon\ .\ \pi'\epsilon\epsilon^*\rho_R\psi),\ \mathit{wrong}\ \text{“bad stack”}$$

$$\mathit{call} : \texttt{N} \rightarrow \texttt{P}$$
$$\mathit{call} = \lambda\nu\ .\ \lambda\epsilon\epsilon^*\rho_R\psi\ .\ \#\epsilon^* = \nu \rightarrow \mathit{applicate}\,\epsilon\epsilon^*(\mathit{single}\,\psi),\ \mathit{wrong}\ \text{“bad stack”}$$

$$\mathit{return} : \texttt{P}$$
$$\mathit{return} = \lambda\epsilon\epsilon^*\rho_R\psi\ .\ \psi\epsilon$$

$$\mathit{push} : \texttt{P} \rightarrow \texttt{P}$$
$$\mathit{push} = \lambda\pi\ .\ \lambda\epsilon\epsilon^*\rho_R\psi\ .\ \pi\epsilon(\epsilon^* {}^\frown \langle\epsilon\rangle)\rho_R\psi$$

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 $\psi\epsilon$ will β-reduce 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-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 [12, Theorem 9], 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** [14] `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 `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, 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 following association of values, the semantics predicts that the correct computational answer from the program is 3:

Variable	Value
ρ	$\mathtt{F} \mapsto \ell_1$
σ	$\ell_1 \mapsto \phi_1;\ \ell_2 \mapsto$ *unspecified*
κ_0	$\lambda\epsilon^*\sigma\,.\,\#\epsilon^*$
ϕ_1	$\langle \ell_2, \lambda\epsilon^*\kappa_0\sigma\,.\,\kappa\langle 1,2,3\rangle\rangle$

When the program `(F)` is executed, the procedure value stored in location ℓ_1 is retrieved, and its second component, ϕ_1 is invoked with no arguments. It in turn applies the continuation κ_0 to the sequence $\langle 1,2,3\rangle$, which should then return the computational answer 3. However, if an implementation makes no provision for multiple-value returners, and if κ_0 is implemented at all, then it always results in the wrong answer 1.

Table 12. Pure Procedure Objects

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

In the next section, we will formalize the reasoning that justifies an implementation in assuming it need implement only "single-valued" objects, and need make no provision for multiple value returners.

In essence our approach is to introduce a new semantics for Scheme. In this semantics, it is clear that there is no mechanism for multiple return values. Since this alternate semantics uses "single valued approximations" to the values used in the standard semantics, we will introduce an operator called **sva** that will transform the standard semantics $\mathcal{E}$ into an alternate semantic function $(\mathbf{sva}\,\mathcal{E})$. With the alternate semantics 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:

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

   at least in the intended case in which the initial values κ and σ—the halt continuation and the initial store respectively—are unproblematic, single-valued objects in a sense to be defined (Theorem 2, clause 2).

2. The compiled byte code is faithful to the alternate semantics, in the sense that

$$(\mathbf{sva}\,\mathcal{E})[\![e]\!]\rho \quad \text{and} \quad (\mathbf{sva}\,\mathcal{B})[\![C(e)]\!]\rho$$

   yield the same answer when applied to suitable initial values (Theorem 4).

We will introduce for convenience a new explicitly defined domain of *pure procedure objects.* Unlike the procedure objects in $\mathtt{F}$ (which equals $\mathtt{L} \times (\mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C})$), those in $\mathtt{F}_p$ contain only the function-like part (namely $\mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C}$), without the location that serves as a tag (see Table 12). 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* : $\mathtt{E}^* \rightarrow \mathtt{E}$ *is the strict function returning* $\perp_{\mathtt{E}}$ *if its argument* ϵ^* *is either* $\perp_{\mathtt{E}^*}$ *or* $\langle\rangle$, *and which returns* $(\epsilon^*\,0)$ *otherwise.*

2. *trunc* : $\mathtt{E}^* \rightarrow \mathtt{E}^*$ *is the non-strict function which takes its argument* ϵ^* *to* $\langle \mathit{zeroth}\,\epsilon^* \rangle$. *Hence*

$$\mathit{trunc}\,\epsilon^* = \begin{cases} \langle \perp_{\mathtt{E}^*} \rangle & \mathit{if}\ \epsilon^* = \perp_{\mathtt{E}^*}\ \mathit{or}\ \epsilon^* = \langle\rangle \\ \langle \epsilon^*\,0 \rangle & \mathit{otherwise.} \end{cases}$$

   and $\#(\mathit{trunc}\,\epsilon^*) = 1$, *even for* $\epsilon^* = \perp_{\mathtt{E}^*}$.

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

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

The auxiliaries *one_arg* and *multiple* coerce from $\mathtt{K}$ to $\mathtt{K}_1$ and back. Expanding the definitions, we have:

LEMMA 1 *1.* $\mathit{one\_arg}\,(\mathit{multiple}\,\psi) = \psi$*;*

2. $\mathit{multiple}\,(\mathit{one\_arg}\,\kappa) = \lambda\epsilon^* \,.\, \kappa(\mathit{trunc}\,\epsilon^*)$*.*

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 $\mathtt{E}_\Sigma$.

DEFINITION 2 **(D, Ω, E-free)**
D *is the smallest set of domains containing:*

1. *The primitive Scheme semantic domains* $\mathtt{N}$, $\mathtt{T}$, $\mathtt{T}_{\mathrm{L}}$, $\mathtt{Q}$, $\mathtt{H}$, $\mathtt{R}$, $\mathtt{M}$, $\mathtt{E}$, *and* $\mathtt{A}$*;*

2. *The Scheme syntactic classes* e, c_{pr}, i, *and* Γ, *regarded as flat domains to which a* $\perp$ *has been added;*

3. *The byte code syntactic classes* t, b, y, z, *and* w, *also regarded as flat domains to which a* $\perp$ *has been added;*

and closed under the operations:

1. $\rightarrow$, *producing function domains (containing continuous functions);*

2. $+$, *producing disjoint union domains;*

3. *, *producing sequence domains; and*

4. $\times$, *producing product domains.*

We 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; in this case we will refer to the operator *of the domain and the* arguments *of the domain. A domain not produced by one of these operators will be called an* indecomposable *domain.*

Ω *is the disjoint union of* **D** *(in some order).*

If $D \in \mathbf{D}$, *we call* D E-free *if* D *is not* $\mathtt{E}$, *and every argument of* D *is* E-free*.*

We assume $\mathtt{E}$ to be indecomposable, as opposed to $\mathtt{E}_\Sigma$, which has $+$ as its operator.

We next define 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. This function ϕ'_p calls ϕ_p with a single-valued approximation to the same expressed value arguments and with the same store. The continuation argument is truncated to ensure that the modified continuation can access only one expressed value, and only a single-valued approximation to that. Crudely put, ϕ'_p cannot pass anything infected with multiple values 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 operator of the domain. Thus, the only interesting case is the one for procedure objects ϕ_p.

Definition 3 **(Respecting types, f^D, Single-valued approximation)**
Given $f : \Omega \rightarrow \Omega$, f respects types *if f is strict, and for every D in $\mathbf{D}$ and every x in Ω such that $x : D$, we have $fx : D$.*
For each D in $\mathbf{D}$, let $f^D \in D \rightarrow D$ be defined, for $d \in D$, to be

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

sva *$: \Omega \rightarrow \Omega$ is the least fixed point of the operator $\alpha \in (\Omega \rightarrow \Omega) \rightarrow (\Omega \rightarrow \Omega)$, which is defined as follows. Let $f \in \Omega \rightarrow \Omega$ and $\omega \in \Omega$ be arbitrary. Then:*

1. *if $\omega = \perp_\Omega$, then $\alpha f \omega = \omega$;*
2. *if $\omega : D$ for an indecomposable D other than $\mathtt{E}$, then $\alpha f \omega = \omega$;*
3. *if $\omega : \mathtt{E}$, then*

$$\alpha f \omega = (\psi_{\mathtt{E}}^{-1}(f(\psi_{\mathtt{E}}(\omega \mid \mathtt{E}) \text{in } \Omega) \mid \mathtt{E}_\Sigma)) \text{in } \Omega;$$

4. *if $\omega : \mathtt{F}_p$, then*

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

where $(\mathit{map}\, f^{\mathtt{E}}\, \epsilon^*)$ is taken to be $\perp_{\mathtt{E}^*}$ for $\epsilon^* = \perp_{\mathtt{E}^*}$;

5. *if $\omega : D$ for a decomposable D other than $\mathtt{F}_p$, then associate with each argument D_i of D the functional $f^{D_i} : D_i \rightarrow D_i$, and lift these to $g : D \rightarrow D$; $\alpha f \omega = g(\omega \mid D) \text{in } \Omega$. More explicitly:*
   - *(A) If $\omega : D_0 \rightarrow D_1$, then $\alpha f \omega = (\lambda y \,.\, (f^{D_1}\, ((\omega \mid (D_0 \rightarrow D_1))\, (f^{D_0}\, y)))) \text{in } \Omega$;*
   - *(B) If $\omega : (D_0 + \ldots + D_n)$, and comes from the ith argument D_i in the sense that $(\omega \mid (D_0 + \ldots + D_n)) : D_i$, then $\alpha f \omega = (f^{D_i}((\omega \mid (D_0 + \ldots + D_n)) \mid D_i)) \text{in } \Omega$;*
   - *(C) If $\omega : D^*$ and $\omega \mid D^* = \langle x, \ldots, y \rangle$, then $\alpha f \omega = \langle f^D\, x, \ldots, f^D\, y \rangle \text{in } \Omega$;*
   - *(D) If $\omega : D_0 \times D_1$ and $\omega \mid D_0 \times D_1 = \langle x, y \rangle$, then $\alpha f \omega = \langle f^{D_0}\, x, f^{D_1}\, y \rangle \text{in } \Omega$.*

All of the real work of α is done by the truncation in the case 4.

If $\omega : D$, then $\alpha f \omega : D$; because in addition αf is strict, for all $f : \Omega \to \Omega$, αf respects types.

As a consequence of the definition, we may infer computational rules for manipulating expressions of the form $\mathbf{sva}\, t$, depending on the type of t.

LEMMA 2 **(Rules for sva)**

1. *If $D \in \mathbf{D}$ is indecomposable and different from* $\mathtt{E}$, *then $\mathbf{sva}^D$ is the identity on D.*
2. *If $D \in \mathbf{D}$ is of the form $D_0 \to D_1$ and different from $\mathtt{F}_p$, and $f \in D$, then*
$$(\mathbf{sva}\, f) = \lambda y\, .\, \mathbf{sva}(f(\mathbf{sva}(y))).$$
3. *If $D \in \mathbf{D}$ is of the form D_0^*, and $x = \langle x_0, \ldots, x_n \rangle \in D$, then*
$$\mathbf{sva}\, x = \langle \mathbf{sva}\, x_0, \ldots, \mathbf{sva}\, x_n \rangle,$$
and similarly for product types.
4. *If $D \in \mathbf{D}$ is of the form $D_0 + \ldots + D_n$, and $x : D_i$, then*
$$\mathbf{sva}\, x = (\mathbf{sva}(x \mid D_i))\text{in } D.$$

Hence, if D is $\mathtt{E}$*-free, then $\mathbf{sva}^D$ is the identity on D.*

For $D \in \mathbf{D}$ and $d \in D$, we will abuse notation by writing $\mathbf{sva}\, d$ for $\mathbf{sva}^D\, d$, i.e., for $(\mathbf{sva}(d\text{in } \Omega))|D$. The following lemma provides the crucial justification for regarding **sva** as 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 the 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 clause 4. Let $\omega : \mathtt{F}_p$; we must show that $(\alpha\, f)(\alpha\, f\, \omega) = (\alpha\, f\, \omega)$. By clause 4, the former (projected into $\mathtt{F}_p$ for convenience) equals:

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

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

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

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

By this lemma, x is of the form **sva** y if and only if $x =$ **sva** x. We will call an object *single-valued* if it is fixed under **sva**.

Since **D** contains the syntactic classes as well as the semantic domains, it follows that **sva** may be applied to the semantic functions $\mathcal{K}$, $\mathcal{E}$, $\mathcal{E}^*$, and $\mathcal{C}$.

In particular, **sva** $c = c$; since, moreover, Scheme has no constants denoting procedure objects, the type of $\mathcal{K}[\![c]\!]$ is always $\mathtt{E}$-free. Thus,

$$(\mathbf{sva}\, \mathcal{K})[\![c]\!] = \mathbf{sva}\, (\mathcal{K}[\![\mathbf{sva}\, c]\!]) = \mathcal{K}[\![c]\!],$$

so $(\mathbf{sva}\, \mathcal{K}) = \mathcal{K}$. However, the remaining semantic functions are not unchanged under **sva**, 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 byte code is justified against the denotational semantics using the initial continuation:

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

which is single-valued.

Theorem 2 **(Faithfulness of Alternative Semantics)**

1. *For all Scheme expressions e, environments ρ, expression continuations κ, and stores σ,*
$$(\mathbf{sva}\,\mathcal{E})[\![e]\!]\rho\,\kappa\,\sigma = \mathcal{E}[\![e]\!]\rho\,(\mathbf{sva}\,\kappa)\,(\mathbf{sva}\,\sigma)$$

2. *Suppose $\kappa = \mathbf{sva}\,\kappa$ and $\sigma = \mathbf{sva}\,\sigma$. Then $\mathcal{E}[\![e]\!]\rho\,\kappa\,\sigma = (\mathbf{sva}\,\mathcal{E})[\![e]\!]\rho\,\kappa\,\sigma$.*

Proof: The second assertion follows immediately from the first.

1. We first use Lemma 2, clause 2 repeatedly:

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

However, since the domain of Scheme expressions and the domain of environments are `E`-free, $\mathbf{sva}\,e = e$ and $\mathbf{sva}\,\rho = \rho$. Finally, since the domain of answers is `E`-free,

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

∎

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 3 **(Truncate Continuations, Eliminate "Single")**

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

2. $(\mathbf{sva}\,\mathcal{E})[\![e]\!]\rho\,(\mathit{single}\;\psi) = (\mathbf{sva}\,\mathcal{E})[\![e]\!]\rho\,(\lambda\epsilon^*\,.\,\psi\,(\epsilon^*\,0))$.

Proof: The (very similar) proofs are by induction on e; we will show the details for clause 1.

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

$$(\mathbf{sva}\,\mathcal{E}^*)(\mathit{permute}(\langle e_0\rangle \frown e^*))\rho\,\kappa_1 = (\mathbf{sva}\,\mathcal{E}^*)(\mathit{permute}(\langle e_0\rangle \frown e^*))\rho\,\kappa_2$$

where

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

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

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

Pushing **sva**s through κ_2 yields:

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

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:

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

■

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 that $(\mathbf{sva}\,\mathcal{E})[\![e]\!]$ and $(\mathbf{sva}\,\mathcal{B})[\![b]\!]$ deliver the same values when applied to suitable initial values. Suitable initial values for e include any environment ρ and expression continuation κ. Since the type of $(\mathbf{sva}\,\mathcal{B})[\![b]\!]$ is $\mathtt{U} \to \mathtt{E} \to \mathtt{E}^* \to \mathtt{U}_R \to \mathtt{K}_1 \to \mathtt{C}$, we use the same environment ρ as the first argument and $\mathit{one\_arg}\,\kappa$ for the last. As for the other arguments, we choose:

1. *unspecified* as the initial value in the "value register;"

2. $\langle\rangle$ as the initial "argument stack;" and

3. $\lambda \nu_1 \nu_2 \,.\, \bot$ as the initial "run-time environment."

DEFINITION 4 **(Environment Composition, MT_C, MT_R)** *Consider $\rho : Ide \rightarrow \mathtt{L}$, $\rho_C : Ide \rightarrow (\mathtt{N} \times \mathtt{N} \cup \{not\_lexical\})$, and $\rho_R : \mathtt{N} \rightarrow \mathtt{N} \rightarrow \mathtt{L}$. Their* environment composition $\rho \triangleleft_{\rho_C}^{\rho_R}$ *is the environment:*

$$\lambda i \,.\, \rho_C\ i = not\_lexical \rightarrow \rho\ i,\ (\lambda p \,.\, \rho_R\ (p\ 0)(p\ 1))(\rho_C\ i)$$

We define MT_C *to be $\lambda i \,.\, not\_lexical$, and* MT_R *to be $\lambda i \,.\, \bot$.*

We will refer to objects ρ_C and ρ_R as "compile-time environments" and "run-time environment" respectively. Expanding definitions and applying extensionality, we have:

LEMMA 4 $\rho \triangleleft_{\mathrm{MT}_C}^{\rho_R} = \rho$.

We return now to proving the correctness of our compiler function, namely $C(e, \rho_C, w)$, which, for the top-level call $C(e, \mathrm{MT}_C, \langle\langle \mathtt{return} \rangle\rangle)$, we abbreviate $C(e)$. We prove first that code generated by compiling a full Scheme expression (as opposed to code fragments) does not depend on the argument given for the value register. In effect, it ignores the original value in the register, and sets it before any reference to it. We use C_{args} to refer to the function implemented by `comp-args` in Table 1 on page 12.

LEMMA 5 **(Compiled expressions ignore the value register)**

1. *Let $\pi = (\mathbf{sva}\ \mathcal{B})[\![C(e, \rho_C, b)]\!]\rho$. For all ϵ, ϵ', $\pi\epsilon = \pi\epsilon'$.*
2. *Let $\pi = (\mathbf{sva}\ \mathcal{Y})[\![C(e, \rho_C, y)]\!]\rho\pi_0$. For all ϵ, ϵ', $\pi\epsilon = \pi\epsilon'$.*
3. *Let $\pi = (\mathbf{sva}\ \mathcal{B})[\![C_{args}(e*, \rho_C, b)]\!]\rho$, 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 $C_{args}(e*, \rho_C, b) = b$. Otherwise, it is of the form:

$$C(e, \rho_C, \langle \mathtt{push} \rangle :: C_{args}(e_1*, \rho_C, b)),$$

so that the result follows by Clause 1. ■

We give next a portion of the proof of the main lemma, which is due to Will Clinger [1], for the case of the VLISP compiler.

LEMMA 6 **(Clinger's Lemma)** *Consider any Scheme expression e,* BBC *closed instruction list b,* BBC *open instruction list y, segment π, environment ρ, compile-time environment ρ_C, value sequence $\epsilon*$, run-time environment ρ_R, initial values ϵ, ϵ_1, and one-argument expression continuation ψ.*

1. *Let* $\pi_1 = (\mathbf{sva}\,\mathcal{B})[\![b]\!]\rho$ *and* $\pi_2 = (\mathbf{sva}\,\mathcal{B})[\![C(e, \rho_C, b)]\!]\rho$. *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. *Let* $\pi_1 = (\mathbf{sva}\,\mathcal{Y})[\![y]\!]\rho\pi$ *and* $\pi_2 = (\mathbf{sva}\,\mathcal{Y})[\![C(e, \rho_C, y)]\!]\rho\pi$. *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. *Let* $\pi_1 = (\mathbf{sva}\mathcal{B})[\![b]\!]\rho$ *and* $\pi_2 = (\mathbf{sva}\mathcal{B})[\![C_{args}(e^*, \rho_C, b)]\!]\rho$. *Suppose that the after code* π_1 *ignores the value register, 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, 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. In proving 2, we make the same assumption and also assume that 1 holds for e itself. 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 [1].

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 $\mathcal{E}^*$,

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

Since π_1 is single-valued, the latter equals $\pi_1\epsilon_1\epsilon^*\rho_R\psi$. Examining the compiler's code, $C_{args}(\langle\rangle, \rho_C, b) = b$, 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 $e :: e^*$. Abbreviating $\lambda\epsilon_1{}^*\,.\,\pi_1\epsilon_1(\epsilon^{*\frown}\epsilon_1{}^*)\rho_R\psi$ by κ, we may apply the semantic clause for $\mathcal{E}^*$ to put the left hand side of our goal in 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,

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

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

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

we may infer:

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

Plugging this in, followed by Lemma 5, then the definition of *push*, and finally the semantics of `push`, we have:

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

We will apply Clause 1 with

$$b = \langle\mathtt{push}\rangle :: C_{args}(e^*,\rho_C,b),$$

and thus with

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

Hence,

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

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

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

as the latter is the code:

$$C(e,\rho_C,\langle\mathtt{push}\rangle :: C_{args}(e^*,\rho_C,b)).$$

■

Finally, we define the appropriate notion of computational equivalence, and prove the main theorem using Clinger's lemma.

Definition 5 **(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,

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

"Equivalence" is a slight abuse of language, as this is not formally an equivalence relation (its domain and codomain are disjoint). Using the rules for **sva** in Lemma 2 and also Theorem 2, we infer that the definition of $\equiv$ could equally have been stated only for *single-valued* κ:

Lemma 7 *1.* $\mathbf{sva}\,(\mathit{one\_arg}\,\kappa) = \mathit{one\_arg}\,(\mathbf{sva}\,\kappa)$*;*

2. $e \equiv b$ *iff for every* ρ *and* κ *such that* $\kappa = (\mathbf{sva}\,\kappa)$,

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

Theorem 4 **(Compiler semantic correctness)**
For any Scheme expression e, $e \equiv C(e)$.

Proof: The theorem is a direct consequence of Clinger's lemma, Clause 1. Let ρ be arbitrary, and let κ be single-valued. Since

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

we instantiate Clause 1 by putting $(\mathbf{sva}\mathcal{B})[\![C(e)]\!]\rho = \pi_2$, and $(\mathbf{sva}\mathcal{B})[\![\langle\langle\mathtt{return}\rangle\rangle]\!]\rho = \mathbf{sva}\;\mathit{return} = \pi_1$; we also let $\psi = \mathit{one\_arg}\,\kappa$ and $\rho_R = \mathrm{MT}_R$. Substituting the other initial values in the right hand side, we infer:

$$\begin{aligned}&(\mathbf{sva}\,\mathcal{E})[\![e]\!]\,(\rho \triangleleft^{\mathrm{MT}_R}_{\mathrm{MT}_C})\,(\mathit{multiple}\,\lambda\epsilon\,.\,(\mathbf{sva}\;\mathit{return})\,\epsilon\langle\rangle\mathrm{MT}_R(\mathit{one\_arg}\,\kappa)) \\ &\quad= \pi_2\;\mathit{unspecified}\,\langle\rangle\;\mathrm{MT}_R\,(\mathit{one\_arg}\,\kappa).\end{aligned} \tag{1}$$

The right hand side is already in the form we want; we need to reduce the left hand side to $(\mathbf{sva}\,\mathcal{E})[\![e]\!]\,\rho\,\kappa$. Since $\mathit{return} = \lambda\epsilon\epsilon^*\rho_R\psi\,.\,\psi\epsilon$, we may use Lemma 2, followed by Lemma 7 and the assumption that $\kappa = (\mathbf{sva}\,\kappa)$, to compute:

$$\begin{aligned}&\lambda\epsilon\,.\,(\mathbf{sva}\;\mathit{return})\,\epsilon\langle\rangle\mathrm{MT}_R(\mathit{one\_arg}\,\kappa) \\ &\quad= \lambda\epsilon\,.\,(\mathbf{sva}\,(\mathit{one\_arg}\,\kappa))(\mathbf{sva}\,\epsilon) = \lambda\epsilon\,.\,\mathbf{sva}\,((\mathit{one\_arg}\,\kappa)(\mathbf{sva}\,(\mathbf{sva}\,\epsilon))) \\ &\quad= \lambda\epsilon\,.\,\mathbf{sva}\,((\mathit{one\_arg}\,\kappa)(\mathbf{sva}\,\epsilon)) = \lambda\epsilon\,.\,(\mathbf{sva}\,(\mathit{one\_arg}\,\kappa))\epsilon \\ &\quad= \lambda\epsilon\,.\,(\mathit{one\_arg}\,(\mathbf{sva}\,\kappa))\epsilon = (\mathit{one\_arg}\,\kappa).\end{aligned}$$

Hence we may rewrite Equation 1 as:

$$\begin{aligned}&(\mathbf{sva}\,\mathcal{E})[\![e]\!]\,(\rho\triangleleft^{\mathrm{MT}_R}_{\mathrm{MT}_C})\,(\mathit{multiple}\,(\mathit{one\_arg}\,\kappa))\\ &\quad=\ \pi_2\ \mathit{unspecified}\,\langle\rangle\ \mathrm{MT}_R\,(\mathit{one\_arg}\,\kappa).\end{aligned}$$

By Lemmas 1 and 4, the left hand side equals:

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

But by Theorem 3, the latter equals $(\mathbf{sva}\,\mathcal{E})[\![e]\!]\,\rho\,\kappa$. ■

3. Faithfulness of an Operational Semantics

With this compiler correctness result, we have completed the denotational portion of the paper. We are ready to introduce a state machine semantics for the Basic Byte Code, and to prove that the state machine semantics is faithful to the denotational one that we have been using up to now.

3.1. Operational Semantics of the Basic Byte Code

The operational semantics of the BBC is determined by a state machine. That state machine uses BBC code as well as other objects as the components of its states. We introduce an expanded syntax called the *augmented byte code* or ABC; it is a language which includes the BBC and also expressions belonging to other grammar classes. The objects from these other classes are used as state components together with code to build up the states of the machine.

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 locations) and the constructors appearing in small capitals in Table 13. The new ABC syntactic categories, which will be defined by BNF, are

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

Syntactic categories from the BBC such as templates t and closed instruction lists b retain the same meaning. The additional productions do not feed back into the recursive construction of the old categories, so the productions for the old categories generate the same set of tree-like objects as they did in the BBC. The new productions are given in Table 13; the old productions, which were given in Table 3, will not be repeated here. The components of a state Σ are called, in order, its *template*, *code*, *value*, *argument stack*, *environment*, *continuation*, and

Table 13. ABC Syntactic Definition

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

store, and we may informally speak of them as being held in registers. Strictly speaking, the template is hardly needed. It contains the name of the procedure, which is useful for debugging; moreover, its presence makes it easier to compare ABC states with the states in lower level machines, in which the template does contain crucial information. The halt states in this machine are the states such the code is $\langle\rangle$. The answer function *ans* is defined for halt states Σ by

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

Form of the Rules. We present the machine transition relation, which we will call *acts*, as the union of subfunctions called (*action*) *rules*. The action rules are functions from disjoint subsets of *states* into *states*. Because their domains are disjoint, the union is in fact a function, and the resulting machine is deterministic.

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.

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 (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

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

Table 14. Some Operational Rules for the Basic Byte Code

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

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 Basic Byte Code Machine. In Table 14, we present the rules connected with `make-cont`, `call`, and `return`. This form of presentation is more compact and convenient than the explicit form, which would write (for instance) the rule for `return` in the form:

$$\begin{aligned}&\langle t,\ \langle\langle \texttt{return} \rangle\rangle,\ v,\ a,\ u,\ \langle \text{CONT}\ t_1\ b_1\ a_1\ u_1\ k_1 \rangle,\ s \rangle \\ &\quad\Longrightarrow \langle t_1,\ b_1,\ v,\ a,\ u_1,\ k_1,\ s \rangle\end{aligned}$$

We will refer to all the rules R given in [6] 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 function *acts*.

The full set of rules is not exhaustive, in the sense of there being a next state whenever a state s is not a halt state. On the contrary, when a state represents a program that has encountered a run-time error, the operational semantics simply provides no next state.

Table 15. Semantic Functions for the Faithfulness Proof

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

3.2. Faithfulness: Form of the Proof

The main idea of the faithfulness proof, which is presented in full in [3], is to associate a denotation in the domain A with each state. If Σ is an initial state with template $t = \langle \mathtt{lap}\ c\ b \rangle$, then the denotation of Σ agrees with the denotational value of

$$\mathcal{T}[\![t]\!]\rho\ \mathit{unspecified}\ \langle\rangle\ (\lambda \nu_1 \nu_2\ .\ \bot)\ (\mathit{one\_arg}\ \kappa_0)$$

where $\kappa_0 = \lambda \epsilon^*\ .\ (\epsilon^*\ 0) : \mathtt{R} \rightarrow (\epsilon^*\ 0) \mid \mathtt{R}\ \text{in}\ \mathtt{A}, \bot$. This ensures that the denotation of an initial state containing t matches its value as used in the compiler proof. For a halt state Σ, the denotation is equal to its computational answer $ans(\Sigma)$, which has been defined to yield the same value as 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. The definitions of the functions take an additional primitive, namely a function ρ_G : Ide $\rightarrow$ L. It associates the names of global variables with the locations in which their values are stored. It is not defined, but only constrained below in the definition of *normal states*. The semantic functions are defined by the equations shown in Table 16.

DEFINITION 6 **(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*

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

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* $\mathcal{O}[\![\Sigma]\!] = ans(\Sigma)\quad [= \mathcal{D}_v[\![v]\!] | \mathtt{R}\ in\ \mathtt{A}]$;

Table 16. Semantic Clauses for the Faithfulness Proof

$\mathcal{D}_v[\![c]\!] = \mathcal{K}[\![c]\!]$.
$\mathcal{D}_v[\![\langle \text{CLOSURE } t \ u \ l\rangle]\!] = \mathit{fix}(\lambda\epsilon.\langle l, (\lambda\epsilon^*\kappa.\mathcal{T}[\![t]\!]\rho_G\epsilon\epsilon^*\mathcal{D}_u[\![u]\!](\lambda\epsilon.\kappa\langle\epsilon\rangle))\rangle)$ 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 $\mathtt{E}$.
$\mathcal{D}_v[\![\text{UNSPECIFIED}]\!] = \mathit{unspecified}$ in $\mathtt{E}$.
$\mathcal{D}_v[\![\text{UNDEFINED}]\!] = \mathit{empty}$ in $\mathtt{E}$.

$\mathcal{D}_a = \mathit{rev} \circ (\mathit{map}\ \mathcal{D}_v)$.

$\mathcal{D}_u[\![\text{EMPTY-ENV}]\!] = (\lambda\nu_1\nu_2.\bot)$.
$\mathcal{D}_u[\![\langle \text{ENV } u \ l^*\rangle]\!] = \mathit{extend}_R\,\mathcal{D}_u[\![u]\!](\mathit{rev}\,l^*)$.

$\mathcal{D}_k[\![\text{HALT}]\!] = (\lambda\epsilon\sigma.\epsilon : \mathtt{R} \rightarrow \epsilon|\mathtt{R} \text{ in } \mathtt{A}, \bot)$.
$\mathcal{D}_k[\![\langle \text{CONT } \langle \mathtt{lap}\ c\ b_0\rangle \ b \ a \ u \ k\rangle]\!] = \lambda\epsilon.\mathcal{B}[\![b]\!]\rho_G\epsilon\mathcal{D}_a[\![a]\!]\mathcal{D}_u[\![u]\!]\mathcal{D}_k[\![k]\!]$.

$\mathcal{D}_s = (\mathit{map}\ \mathcal{D}_v)$.

2. *Otherwise,* $\mathcal{O}[\![\Sigma]\!]$ *is* $\mathcal{O}[\![\mathit{acts}(\Sigma)]\!]$.

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 $\Sigma = \langle\langle \mathtt{lap}\ c\ b_0\rangle, b, v, a, u, k, s\rangle$ be an ABC state. The functions L_{glo}, L_{env}, L_{mp}, and L_{ip} map ABC states to finite sets of locations. The formal definitions ([3]) will not be repeated here; the functions represent respectively the sets of locations in use in Σ for global variables, local variables, mutable pairs, and immutable pairs. For instance, L_{glo} is the range of the function ρ_G.

Definition 7 **(Normal States, Initial States)**
Σ is normal if the following conditions hold:

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

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

(3) For each immutable pair $p = \texttt{'}(c_1\ \texttt{.}\ c_2)$ *occurring in* Σ*, there are* $l_1, l_2 \in \mathtt{L}$ *such that* $s(l_1) = c_1$ *and* $s(l_2) = c_2$*; and similarly for immutable vectors and strings.*

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 BBC *template* $t = \langle \mathtt{lap}\ c\ b\rangle$ *is any normal* ABC *state of the form*

$$\langle t, b, \text{UNSPECIFIED}, \langle\rangle, \text{EMPTY-ENV}, \text{HALT}, s\rangle$$

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. The following lemmas are needed in the proof of faithfulness.

LEMMA 8 *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 9 *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 10 *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 10, 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 $\Sigma = \langle\langle\mathtt{lap}\ c\ b\rangle, \langle\langle\mathtt{return}\rangle\rangle, v, a, u, k, s\rangle$, where $k = \langle\text{CONT}\ t_1\ b_1\ a_1\ u_1\ k_1\rangle$ and $t_1 = \langle\mathtt{lap}\ c'\ b'\rangle$.

Then $R(\Sigma) = \Sigma' = \langle t_1, b_1, v, a_1, u_1, k_1, s\rangle$.

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

■

Although not all of the cases are as straightforward as this (see [3] 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.

THEOREM 5 **(Faithfulness)** *The operational semantics for* BBC *is faithful: for all* BBC *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 $\mathcal{O}[\![\Sigma]\!]$, using Lemmas 8–10. ■

In fact the actual VLISP faithfulness proof as described in [3] differs from this description in several ways. First, the proof is not carried out at the level of the BBC, but rather at the level of the Tabular Byte Code introduced in [5]. This version of the byte code uses a template table to store embedded templates (representing locally introduced procedures), constants, and global variables. These are referenced by their numeric index within the table. Conceptually this makes no difference to any of the issues described here, so we have opted to avoid introducing another grammar.

Moreover, the actual proof is somewhat more complex. The operational semantics and the denotational semantics differ in the order in which they copy values from 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 [3] splits the faithfulness proof into two parts. The first part has exactly the form described in this section, but uses a state machine with an appropriate version of the rule to make an environment rib. The second part shows that the machine with the actual, implemented rule *refines* the one used in the first part, in the sense to be developed in the next section.

Theorem 5 is a conditional: it asserts only that *if* the machine computes an answer, then that is the correct answer. This property also holds of less useful machines than the BBCM, for instance a machine whose transition relation is empty. However, we conjecture that the technique of [12, Theorem 9] would allow us to prove the converse, which would establish the adequacy of the operational semantics. It would assert that the machine succeeds in computing an answer when the denotational semantics predicts a non-$\perp$, non-erroneous answer. The work described in this section was carried out before we became aware of that technique.

4. State Machine Refinement

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) that we already know would be acceptable.

We consider non-deterministic machines but not I/O here. Non-deterministic machines arose naturally when we introduced garbage collection. By contrast, we avoided treating I/O. On the one hand, the official Scheme semantics does not treat I/O. On the other hand, I/O can be added, either within the denotational or the operational style, without any very serious theoretical problem, but with more notational burden.

In this section we will define a notion of state machine refinement, and we will justify a particular technique for showing that one state machine refines another. We call this technique "storage layout relations," following Wand and Oliva [18], although the underlying idea appears to be much older [8]. Wand and Oliva introduce the method by means of an example. Because VLISP used the method repeatedly, we will use this section to provide a formal definition, and to justify the method relative to a precise notion of refinement. We begin with some notation for state machines.

4.1. State Machines

There are many ways of pinning down details for state machines. Since we consider non-deterministic machines but not I/O, the following is convenient.

DEFINITION 8 **(State Machines)**
State transition machines *are 5-tuples*

$$\langle \mathit{states},\ \mathit{inits},\ \mathit{halts},\ \mathit{acts},\ \mathit{ans} \rangle$$

such that

(a) $\mathit{inits} \subseteq \mathit{states}$*, and* $\mathit{halts} \subseteq \mathit{states}$*;*

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

(c) $\Sigma \in \mathit{halts}$ *implies that* Σ *is not in the domain of acts; and*

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

If $\langle \Sigma, \Sigma' \rangle \in \mathit{acts}$, we will say that Σ *can proceed to* Σ'. A state Σ is called a *halt state* if $\Sigma \in \mathit{halts}$, which is equivalent to $\mathit{ans}(\Sigma)$ being well defined. If *acts* is actually a function, then the machine is *deterministic*.

If T is a finite or infinite sequence of states, then it is a *computation* (or a *trace*) if and only if $T(0) \in \mathit{inits}$ and, for every i such that $T(i+1)$ is defined, $\langle T(i), T(i+1) \rangle \in \mathit{acts}$.

A computation is *maximal* if it is infinite or its last state is not in the domain of *acts*. A finite, maximal computation is *successful* if its last state is a halt state; any other finite, maximal computation is *erroneous*.

A state is *accessible* if it is $T(i)$ for some computation T and integer i.

When T is a successful computation, that is, a finite computation terminating with a halt state Σ, we will write $\mathit{ans}(T)$ to mean $\mathit{ans}(\Sigma)$. Otherwise, $\mathit{ans}(T)$ is undefined. We will say that M *can compute* a *from* Σ if there exists a T such that $T(0) = \Sigma$ and $\mathit{ans}(T) = a$.

4.2. Refinement and Storage Layout Relations

In this section we will define a notion of refinement suitable for state machines without I/O. We will then introduce the notion of a *storage layout relation*, which we will show guarantees refinement.

In this section we will let M^A and M^C be state machines, which we regard as abstract and concrete respectively, and we will let decorated symbols refer to objects in these state machines. So for instance $inits^A$ is the set of initial states of M^A; the variable T^C ranges over the traces of M^C; and the variable Σ^C ranges over the states of M^C.

The reader may want to pronounce the relations $\sim_0$, $\sim$, and $\approx$ in this section using terms such as "refines" or "implements." This motivates the order of the arguments, with the more concrete argument always appearing on the left.

DEFINITION 9 **(Refinement)** *Consider a relation* $\sim_0 \;\subseteq\; inits^C \times inits^A$.

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

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

1. *Every abstract initial state corresponds to some concrete initial state:*

$$\forall \Sigma^A : inits^A \;.\; \exists \Sigma^C : inits^C \;.\; \Sigma^C \sim_0 \Sigma^A$$

2. *Whenever* $\Sigma^C \sim_0 \Sigma^A$ *and* M^C *can compute* a *from* Σ^C*, then* M^A *can compute* a *from* Σ^A*;*

3. *Whenever* $\Sigma^C \sim_0 \Sigma^A$ *and* M^A *can compute* a *from* Σ^A*, then* M^C *can compute* a *from* Σ^C.

Strong refinement in the sense given here is a very natural notion when state machines return information only through eventually computing an answer. As for weak refinement, it is an informative relation between state machines—it explains what behavior to expect from the concrete machine M^C—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, the computations that can be simulated are those in which all the values that arise can be stored in a machine word of fixed width, such as 32 bits.

Our standard approach [18] for proving that one state machine refines another is to prove that there is a *storage layout relation* between them. The definition makes use of a similarity relation on traces, induced from a similarity relation on states.

DEFINITION 10 **(Sequential extension $\approx$)** *Consider a relation*

$$\sim \;\subseteq\; states^C \times states^A.$$

Then $\approx$, the sequential extension *of $\sim$, is the relation between traces of M^C and traces of M^A such that $T^C \approx T^A$ iff there exists a monotonic function f from the domain of T^C onto the domain of T^A, such that for all i in the domain of T^C,*

$$T^C(i) \sim T^A(f(i)).$$

Thus, $f(i)$ tells us "where we are" in the abstract trace, when we have reached position i in the concrete trace. In some cases, however, more than one function f will meet the condition.

The following lemma is intended to bring out the operational significance of $\approx$. It holds between two traces when their successive states are related by $\sim$. Either the concrete machine can proceed by a "stuttering" transition, which corresponds to no transition in the abstract machine, or else the two machines may each perform a transition. In either case, the resulting states must still be related by $\sim$. The stuttering transition is expressed below in Clause 3, while the lockstep transition is expressed in Clause 4.

Lemma 11 *Suppose that $\sim \subseteq states^C \times states^A$. Let $\simeq$ be the least relation between traces of M^C and traces of M^A such that:*

1. $\langle\rangle \simeq \langle\rangle$;
2. *If Σ^C and Σ^A are initial states and $\Sigma^C \sim \Sigma^A$, then $\langle\Sigma^C\rangle \simeq \langle\Sigma^A\rangle$;*
3. *If Σ_0^C can proceed to Σ_1^C, and $\Sigma_1^C \sim \Sigma^A$, then*
$$T^C \frown \langle\Sigma_0^C\rangle \simeq T^A \frown \langle\Sigma^A\rangle \quad \text{implies} \quad T^C \frown \langle\Sigma_0^C, \Sigma_1^C\rangle \simeq T^A \frown \langle\Sigma^A\rangle;$$
4. *If Σ_0^C can proceed to Σ_1^C, Σ_0^A can proceed to Σ_1^A, and $\Sigma_1^C \sim \Sigma_1^A$, then*
$$T^C \frown \langle\Sigma_0^C\rangle \simeq T^A \frown \langle\Sigma_0^A\rangle \quad \text{implies} \quad T^C \frown \langle\Sigma_0^C, \Sigma_1^C\rangle \simeq T^A \frown \langle\Sigma_0^A, \Sigma_1^A\rangle.$$

Then if T^C and T^A are finite, $T^C \simeq T^A$ iff $T^C \approx T^A$.

Proof: We first make a general observation. Suppose that i is in the domain of T^C, and $x = \{j \mid 0 \leq j \leq i\}$. Then, since f is monotonic and onto the domain of T^A, the image of x under f is

$$\{j \mid 0 \leq j \leq f(i)\}.$$

Letting $i = 0$, we may infer that $f(0) = 0$ unless the trace T^C is $\langle\rangle$. Similarly, we must have either $f(i+1) = f(i)$ or $f(i+1) = f(i) + 1$, whenever $i+1$ is in the domain of T^C.

$\Rightarrow$: If $T^C \simeq T^A$ by Clause 1 or 2, then the function f is the empty function or the function mapping 0 to 0 respectively.

Suppose that $T^C \frown \langle\Sigma_0^C, \Sigma_1^C\rangle \simeq T^A \frown \langle\Sigma^A\rangle$ holds by Clause 3, that the index of the last occurrence of Σ_1^C is $i+1$, and the index of the last occurrence of Σ^A

is j. Then inductively there is an acceptable f_0 mapping $\{k \mid 0 \leq k \leq i\}$ onto $\{k \mid 0 \leq k \leq j\}$. Then extending f_0 to f by stipulating $i+1 \mapsto j$, f is monotone and onto $\{k \mid 0 \leq k \leq j\}$. Moreover, by the hypothesis to Clause 3, $T^C(i+1) \sim T^A(j)$.

Suppose that $T^C \frown \langle \Sigma_0^C, \Sigma_1^C \rangle \simeq T^A \frown \langle \Sigma_0^A, \Sigma_1^A \rangle$ holds by Clause 4, where the last occurrences of Σ_1^C and Σ_1^A have indices $i+1$ and $j+1$ respectively. Then inductively there is an acceptable f_0 mapping $\{k \mid 0 \leq k \leq i\}$ onto $\{k \mid 0 \leq k \leq j\}$, which we may extend to the needed f by stipulating $i+1 \mapsto j+1$.

$\Leftarrow$: By induction on T^C. Let f be an acceptable function from the domain of T^C to the domain of T^A. The cases for $\langle\rangle$ and $\langle \Sigma^C \rangle$ are immediate from Clauses 1 and 2. Otherwise, T^C is of the form $T_0^C \frown \langle \Sigma_0^C, \Sigma_1^C \rangle$, where the last occurrence of Σ_1^C has index $i+1$. If $f(i+1) = f(i)$, then apply Clause 3; if $f(i+1) = f(i)+1$, apply Clause 4. ∎

In the following definition, we use the notion of quasi-equality [2, 4], written $s == t$, meaning that s and t are equal if either of them is well defined. If $\downarrow$ means "is defined," then $s == t$ is in effect an abbreviation of $(s \downarrow \vee\, t \downarrow) \Rightarrow s = t$. In particular, $ans^C(\Sigma^C) == ans^A(\Sigma^A)$ implies that either both Σ^C and Σ^A are halt states, or else neither of them is.

Definition 11 **(Storage layout relation)** *A relation $\sim \subseteq states^C \times states^A$ is a* storage layout relation *for M^C and M^A iff:*

1. *Every abstract initial state corresponds to some concrete initial state:*

$$\forall \Sigma^A : inits^A \,.\, \exists \Sigma^C : inits^C \,.\, \Sigma^C \sim \Sigma^A;$$

2. *For every concrete trace, and abstract initial state Σ^A corresponding to its first member, some corresponding abstract trace starts from Σ^A:*

$$\forall T^C, \Sigma^A : inits^A \,.\, T^C(0) \sim \Sigma^A \Rightarrow \exists T^A \,.\, T^A(0) = \Sigma^A \wedge T^C \approx T^A;$$

3. *For every abstract trace, and concrete initial state Σ^C corresponding to its first member, some corresponding concrete trace starts from Σ^C:*

$$\forall T^A, \Sigma^C : inits^C \,.\, \Sigma^C \sim T^A(0) \Rightarrow \exists T^C \,.\, T^C(0) = \Sigma^C \wedge T^C \approx T^A;$$

4. *Of two corresponding states, if both are halt states, then they deliver the same answer, and otherwise neither is a halt state:*

$$\forall \Sigma^C, \Sigma^A \,.\, \Sigma^C \sim \Sigma^A \Rightarrow ans^C(\Sigma^C) == ans^A(\Sigma^A).$$

A relation $\sim \subseteq states^C \times states^A$ is a weak storage layout relation *for M^C and M^A iff conditions 2 and 4 above hold.*

THEOREM 6 **(Storage layout relations guarantee refinement)** *If $\sim$ is a weak storage layout relation for M^C and M^A, then M^C weakly refines M^A via the relation $\sim_0$ defined to be $\sim \cap\, inits^C \times inits^A$.*

If $\sim$ is a storage layout relation $\sim$ for M^C and M^A, then M^C strongly refines M^A via $\sim_0 = \sim \cap\, inits^C \times inits^A$.

Proof: Clause 1 of Definition 9 is immediate from Clause 1 of Definition 11.

We next argue that Clause 4 of Definition 11 implies that, when $T^C \approx T^A$,

$$ans^C(T^C) == ans^A(T^A). \qquad (\dagger)$$

On the one hand, if $ans^C(T^C)$ is well-defined, then T^C terminates in a halt state at its last index i. Moreover, $T^C(i) \sim T^A(f(i))$, so by Clause 4, $answer^C(T^C(i)) = answer^A(T^A(f(i)))$. Finally, since T^A is a trace, $f(i)$ must be its last index, and $ans^A(T^A) = answer^A(T^A(f(i)))$. On the other hand, if $ans^A(T^A)$ is well-defined, similar reasoning applies, using the fact that f is onto.

To prove Clause 2 of Definition 9, suppose that T^C computes the answer a from Σ^C. By Clause 2 of Definition 11, there is a T^A with $T^A(0) = \Sigma^A$ and $T^C \approx T^A$. By $\dagger$, $ans^A(T^A) = a$.

For Clause 3, we use Clause 3 of Definition 11 similarly. ∎

Clause 2 in Definition 11 is, in essence, a safety condition on the concrete machine, while Clause 3 is a liveness condition on it, as the following lemma suggests. In this lemma, we will use γ as a variable over sequences of states of M^C.

LEMMA 12 **(Safety and liveness conditions)** *Consider a relation $\sim \subseteq states^C \times states^A$.*

1. *(Safety) Suppose that whenever $\Sigma_0^C \sim \Sigma_0^A$ and Σ_0^C can proceed to Σ_1^C, then*
   - *(A) either $\Sigma_1^C \sim \Sigma_0^A$ (a stuttering step);*
   - *(B) or else there exists a Σ_1^A such that Σ_0^A can proceed to Σ_1^A and $\Sigma_1^C \sim \Sigma_1^A$.*

   Then Clause 2 of Definition 11 holds.
2. *(Liveness) Suppose that whenever $\Sigma_0^C \sim \Sigma_0^A$ and Σ_0^A can proceed to Σ_1^A, then there exists a γ such that:*
   - *(A) $\forall i \,.\, i < \#\gamma - 1 \Rightarrow \gamma(i) \sim \Sigma_0^A$;*
   - *(B) $\gamma(\#\gamma - 1) \sim \Sigma_1^A$; and*
   - *(C) γ is* tracelike, *in the sense that*

     $$\forall i < \#\gamma - 1 \,.\, \langle \gamma(i), \gamma(i+1) \rangle \in acts^C.$$

   With this definition, γ is a trace T^C if it is tracelike and $\gamma(0)$ is an initial state.

Then Clause 3 of Definition 11 holds.

Proof: 1. By induction on T^C. 2. By induction on T^A. ■

This lemma shows that the notion of storage layout relation is akin to the notion of a *weak bisimulation* (see [11, Chapter 5, Definition 5], for example). However, in our formulation, the relation is one-sided in that all the silent τ-transitions are taken by the concrete machine.

In many cases, such as the linearizer algorithm presented in Section 5, we use a storage layout relation to prove not just that one state machine refines another, but that a particular language translation F is correct. To see why that makes sense, let M^A be the state machine that gives the operational semantics for the source language, and let M^C give the operational semantics for the target language. Also, suppose that the states of M^A have the form of pairs $\langle p, \Sigma_0 \rangle$, where p is a program in the *source* language of F and Σ_0 contains the remaining state components, while the states of M^C have the form $\langle p', \Sigma_0' \rangle$, where p' is a program in the *target* language of F. Suppose further that at least for *initial* states, the non-code portion of a state may be shared between an abstract state and a concrete state. That is,

$$\langle p, \Sigma_0 \rangle \in \mathit{inits}^A \Rightarrow \langle F(p), \Sigma_0 \rangle \in \mathit{inits}^C.$$

This was the point of the restriction to codeless stores in Definition 7.

Then we may say that F *exhibits M^C as a refinement of M^A* if M^C refines M^A via a relation built from projection, F, and pairing. In particular, the relation is to hold between a concrete initial state $\langle p', \Sigma_0' \rangle$ and an abstract initial state $\langle p, \Sigma_0 \rangle$ if $p' = F(p)$ and $\Sigma_0' = \Sigma_0$. If this relation is a refinement, then M^C computes the same answers when started in state $\langle F(\Sigma^A(0)), \Sigma^A(1) \rangle$ that M^A computes when started in state Σ^A.

How can we use the storage layout relation method to prove that F refines M^C to M^A? We must exhibit a storage layout relation $\sim$ such that F establishes $\sim$, in the sense that $\langle F(p), \Sigma_0 \rangle \sim \langle p, \Sigma_0 \rangle$, whenever the latter is an initial state of M^A.

We illustrate this method in the next section, by proving that the VLISP flattener (or code linearizer) is correct. Section 5.4 proves that the flattener establishes a relation $\sim$, which is the real content of Theorem 7. Then Theorem 8 in Section 5.5 shows that $\sim$ is in fact a storage layout relation.

5. The Flattener

The **flattener** [6] transforms byte coded procedures in the BBC into a linear rather than a tree-structured form. Conditionals are represented in a familiar way using `branch` and `branch-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).

However, before we introduce the new language, we will provide a bit more detail about the BBC and its operational semantics (Section 5.1). We will then present the FBC and give it an operational semantics (Section 5.2) in a style very similar to the presentation of the BBC semantics in Section 3.1. We will briefly describe the flattener algorithm(Section 5.3). 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 (Section 5.4). Finally, we extend the code correspondence relation to a storage layout relation between states of the newly introduced flattened byte code state machine (FBCM) and the BBCM (Section 5.5). This establishes that the FBCM is a refinement of the BBCM.

5.1. The Basic Byte Code in More Detail

In order to describe the operational semantics of conditionals in the BBCM, 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 12 **(Open-adjoin)**

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

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

$(\langle \texttt{make-cont}\ y\ n \rangle :: b)^\smile w = \langle \texttt{make-cont}\ y^\smile w\ n \rangle :: b;$ *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 3, a closed BBC 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 13 **(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 $^\smile$, as shown in Table 17.

Table 17. BBCM Open Branch Rules

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:
$b' = y_1 {}^\smile b_1$	$b' = y_2 {}^\smile b_1$

Table 18. Flattened Byte Code: Syntax

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

5.2. Flattened Byte Code Syntax and Operational Semantics

The Flattened Byte Code (FBC) is a modification of the Basic Byte Code that uses unnested linear sequences of tokens for the code part of its templates. Instead of the BBC conditional instruction `unless-false`, FBC has `branchf` and `branch`, 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 appear as Table 18. In this version of the syntax, the code sequences are not exclusively unnested, since the instructions $\langle \texttt{closure}\ t \rangle$ and $\langle \texttt{constant}\ c \rangle$ may have nested subexpressions. However, in the full VLISP development, this is not the case; a "tabularizer" stage had previously moved the nested values into a template table, in which they were referenced by a natural number index (cf. page 37).

FBC states—characterized in Table 19—differ from BBC states in that their templates and code belong to the FBC language rather than the BBC language. In addition, instead of having a "code register" they have a numerical "program counter," to which we will refer conventionally as n. It is interpreted as an offset into the code sequence of the template. We will use the variable w to refer to the code sequence (or just the *code*) of a state, i.e., $w = t(2)$. With our conventions, $w(n)$ is

Table 19. Flattened Byte Code State Machine: State Component Syntax

$$
\begin{array}{rcl}
v & ::= & c \mid \langle \textsc{closure}\ t\ u\ l \rangle \mid \langle \textsc{escape}\ k\ l \rangle \mid \langle \textsc{mutable-pair}\ l_1\ l_2 \rangle \\
 & & \mid \langle \textsc{string}\ l^* \rangle \mid \langle \textsc{vector}\ l^* \rangle \mid \textsc{unspecified} \mid \textsc{undefined} \\
a & ::= & v^* \\
u & ::= & \textsc{empty-env} \mid \langle \textsc{env}\ u\ l^* \rangle \\
k & ::= & \textsc{halt} \mid \langle \textsc{cont}\ t\ b\ a\ u\ k \rangle \\
s & ::= & v^* \\
\Sigma & ::= & \langle t,\ n,\ v,\ a,\ u,\ k,\ s \rangle
\end{array}
$$

the first element of $w \dagger n$, if $0 \leq n < \#w$. The states in *halts* are those 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 rules for the FBC (Table 20) will use the same conventions as were used for the BBC, except that here n stands for the offset of the ingoing state, n' for the offset of the state produced, and w for $t(2)$, 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 by the condition $n_0 \oplus n_1 = (256 * n_0) + n_1$. In the FSBCM implementation, the values of n_0 and n_1 are always stored in a single byte. The operator $\oplus$ is needed so that we can represent offsets that are larger than 255. Although large numbers may be represented in more than one way in the form $n_0 \oplus n_1$ (if n_1 may be large), this does not affect our proof.

5.3. The Flattener

The flattener is implemented by a purely applicative Scheme program, part of which is shown in Figure 2; 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* BBC instruction list b or an *open* BBC instruction list y.

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

In the next two subsections, we will exhibit a relation between BBCM states and FBCM. The task of the first section is to show that the flattener establishes this relation; the task of the second section is to show that the relation is in fact a storage

Table 20. Typical FBCM Rules

Rule 1: **Call**
Domain conditions: $w \dagger n = \langle \texttt{call}\ \#a \rangle ^\frown w_1;\quad v = \langle \textsc{closure}\ t_1\ u_1\ l_1 \rangle$
Changes: $t' = t_1;\quad n' = 0;\quad u' = u_1$
Rule 2: **Make Continuation**
Domain conditions: $w \dagger n = \langle \texttt{make-cont}\ n_1\ n_2\ \#a \rangle ^\frown w_1$
Changes: $n' = n + 4;\quad a' = \langle\rangle;\quad k' = \langle \textsc{cont}\ t\ (n + 4 + (n_1 \oplus n_2))\ a\ u\ k \rangle$
Rule 3: **Branchf/False**
Domain conditions: $w \dagger n = \langle \texttt{branchf}\ n_1\ n_2 \rangle ^\frown w_1;\quad v = \mathit{false}$
Changes: $n' = n + 3 + (n_1 \oplus n_2)$
Rule 4: **Branchf/True**
Domain conditions: $w \dagger n = \langle \texttt{branchf}\ n_1\ n_2 \rangle ^\frown w_1;\quad v \neq \mathit{false}$
Changes: $n' = n + 3$

```
(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)))
```

Figure 2. Flattener Algorithm

Table 21. Recursive Conditions for $\simeq$.

1. For $m = \langle \texttt{return} \rangle$ or $\langle \texttt{call}\, n \rangle$, $m^\frown w_F \simeq \langle m \rangle$.
2. $\langle \texttt{closure}\, t_F \rangle^\frown w_F \simeq \langle \texttt{closure}\, t \rangle :: m^*$ if, first, either $m^* = w_F = \langle\rangle$ or $w_F \simeq m^*$, and, second, the code correspondence holds between the code embedded within t_F and t respectively:
$$t_F(2) \simeq t(2).$$
3. For atomic z other than $\texttt{closure}$, $z^\frown w_F \simeq z :: m^*$ if either $m^* = w_F = \langle\rangle$ or $w_F \simeq m^*$.
4. $\langle \texttt{make-cont}\, n_0\; n_1\; n \rangle^\frown w_F \simeq \langle \texttt{make-cont}\, w\; n \rangle :: b$ if
$$w_F \simeq b \quad \text{and} \quad w_F \dagger (n_0 \oplus n_1) \simeq w.$$
4′. $\langle \texttt{make-cont}\, n_0\; n_1\; n \rangle^\frown w_F \simeq \langle \texttt{make-cont}\, \langle\rangle\; n \rangle :: b$ if
$$w_F \simeq b \quad \text{and} \quad w_F \dagger (n_0 \oplus n_1) = \langle\rangle.$$
5. $\langle \texttt{branchf}\, n_0\; n_1 \rangle^\frown w_F \simeq \langle\langle \texttt{unless-false}\, b_1\; b_2 \rangle\rangle$ if
$$w_F \simeq b_1 \quad \text{and} \quad w_F \dagger (n_0 \oplus n_1) \simeq b_2.$$
6. $\langle \texttt{branchf}\, n_0\; n_1 \rangle^\frown w_F \simeq \langle \texttt{unless-false}\, y_1\; y_2 \rangle :: w$ if
$$w_F \simeq y_1{}^\smile w \quad \text{and} \quad w_F \dagger (n_0 \oplus n_1) \simeq y_2{}^\smile w.$$
7. $\langle \texttt{branch}\, n_0\; n_1 \rangle^\frown w_F \simeq w$ if $w_F \dagger (n_0 \oplus n_1) \simeq w$.

layout relation. Thus, the two subsections together establish that flattening code leaves its computational results unchanged.

5.4. Code Correspondence

We will define the storage layout relation between states justifying this refinement in terms of a preliminary "code correspondence" relation $\simeq$ holding between flattened and unflattened code. We will write $w_F \simeq w$; the main theorem of this section asserts that the flattener establishes this relation: $F(w) \simeq w$. The correspondence relation is defined as the least relation satisfying the conditions given in Table 21. 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 14 **(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 15 **($\simeq$/adjoin)** *If* $w_0 \simeq b_0$*, then for any* w_1*,* $w_0{}^\frown w_1 \simeq b_0$*.*

Proof: We will assume that w_0 is not of the form $\langle \mathtt{branch}\; n_0\; n_1\rangle^\frown w_2$, so that $w_0{}^\frown w_1 \simeq b_0$ is not true in virtue of clause 6. For otherwise, $w_2 \dagger (n_0 \oplus n_1) \simeq b_0$, and we may apply the lemma to infer that

$$w_2 \dagger (n_0 \oplus n_1)^\frown w_1 \simeq b_0.$$

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

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:

$$\langle \mathtt{branch}\; n_0\; n_1\rangle^\frown (w_2{}^\frown w_1) \simeq b_0.$$

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

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

1. Suppose $b_0 = \langle m\rangle = \langle\langle \mathtt{return}\rangle\rangle$ or $\langle\langle \mathtt{call}\; n\rangle\rangle$, 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 $b_0 = \langle \mathtt{closure}\; t\rangle :: b$ and $w_0 = \langle \mathtt{closure}\; t_1\rangle^\frown w$. Since $w_0 \simeq b_0$, we may infer from the definition of $\simeq$ that $w \simeq b$ and $t_1(2) \simeq t(2)$. Hence we may apply the induction hypothesis to infer that $w^\frown w_1 \simeq b$, and we may then apply Clause 2 to infer that $w_0{}^\frown w_1 \simeq b_0$.

3. Suppose that $b_0 = z :: b$, $w_0 = z^\frown w$, where z is not a `closure` instruction. (The syntax ensures that m^* really is of the form b.) Since, inductively, $w^\frown w_1 \simeq b$, we simply apply clause 3 to $w^\frown w_1$.

4. If $b_0 = \langle \mathtt{make\text{-}cont}\; b_1\; n\rangle :: b$, and $w_0 = \langle \mathtt{make\text{-}cont}\; n_0\; n_1\; n_2\rangle^\frown w$: Assume inductively:

$$w^\frown w_1 \simeq b \quad \text{and} \quad w \dagger (n_0 \oplus n_1)^\frown w_1 \simeq b_1.$$

   By the drop/adjoin lemma, we may apply clause 4 with $w^\frown w_1$ in place of w.

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

5, 6, 7. Similar to 4.

■

In contrast to closed instruction lists, for *open* instruction lists, adding code to the end of a flattened version corresponds to $^\smile$:

Lemma 16 **($\simeq$/open-adjoin)** *If* $w_0 \simeq y$ *and* $w_1 \simeq w$*, then* $w_0{}^\frown w_1 \simeq y^\smile w$*.*

The main theorem of this section asserts that the output produced by the flattener bears the code correspondence relation $\simeq$ to its input. Its proof is similar in structure to the proof of Lemma 15.

Theorem 7 **(Flattener establishes $\simeq$)** $F(w) \simeq w$.

Table 22. Closure Conditions for $\sim$.

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. $\langle \texttt{template}\ c\ w\rangle \sim \langle \texttt{template}\ c\ b\rangle$, if $w \simeq b$
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) $t_2(2) \dagger n \simeq b$, 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) $t_2(2) \dagger n \simeq b$, 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 $v_1 = v_2$.

5.5. 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 the FBCM on the left and the BBCM on the right, building up to a notion of correspondence between FBC states and BBC 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 6 the FBCM is a faithful refinement of the BBCM.

Because the two languages share some syntactic objects, we can simplify the definition. Some of the objects that are shared by the two languages are *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 22. 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.

By this definition and Definition 7, which ensures that the stores of initial states are codeless, we may infer that if an initial BBC and an initial FBC state share the same store, then they correspond if their code corresponds. That is, suppose $t_F = \langle \texttt{template}\ c\ w\rangle$; $t = \langle \texttt{template}\ c'\ b\rangle$; and $w \simeq b$. Then

$$\langle t_F, 0, \text{UNSPECIFIED}, \langle\rangle, \text{EMPTY-ENV}, \text{HALT}, s\rangle$$

corresponds to

$$\langle t, t(2), \text{UNSPECIFIED}, \langle\rangle, \text{EMPTY-ENV}, \text{HALT}, s\rangle.$$

By Theorem 7, we can always take w to be $F(b)$, so condition 1 in Definition 11 (storage layout relations) is satisfied. Moreover, condition 4 follows immediately from clause 8 for $\sim$; hence:

LEMMA 17 *$\sim$ satisfies the initial condition (1) and the halt condition (4) 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 BBC 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 18 **(Determinacy of Transitions)**
If M is either the BBCM or the FBCM, and M is in state Σ, then there is at most one state Σ' such that M can proceed by one rule transition from Σ to Σ'.

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 cases when one must also consider the value register or the argument register. Given a particular 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. ∎

For instance, the rules for primitives allocating new storage stipulate that the new locations selected lie immediately after the previously active portion of the store.

Clearly the BBC rules are closely paralleled by the FBC rules: let us say that a BBC 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 Branchf/True, and finally that Closed Branch/False and Open Branch/False both *correspond* to Branchf/False. No BBC rule corresponds to Branch. A *branch* rule means one whose name ends in /True or /False—four for BBC and two for FBC.

THEOREM 8 **(Preservation of State Correspondence)**
Let Σ be the state of the BBCM and Σ_F the state of the FBCM, and assume that $\Sigma_F \sim \Sigma$. Then

1. *if the FBCM proceeds by the Branch rule to state Σ'_F, then $\Sigma'_F \sim \Sigma$ and the FBCM cannot proceed from Σ'_F by the Branch rule;*

2. *if the FBCM cannot proceed by the Branch 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 17, Lemma 12, and Theorem 6, we can infer:

THEOREM 9 $\sim$ *is a storage layout relation for the* FBCM *and the* BBCM. *Hence, the* FBCM *refines the* BBCM.

The VLISP Scheme compiler is structured as the composition of five stages: the byte code compiler, tabulator, flattener, linker, and image builder. We have now completed our presentation of the algorithms and correctness proofs of the byte code compiler and the flattener. We have ignored the tabulator and the linker since they do not raise significant new verification issues. The image builder, which produces the final SBC program as a binary image (i.e., a flat sequence of bits), uses a storage layout relation defined in a different way. However, since this same technique is used in the more interesting proof of the garbage collector (see Section 6.3), we will not describe the image builder in this paper.

6. The Virtual Machine

The VLISP Scheme compiler translates any Scheme program p to a program *sbc* in a language called Stored Byte Code (SBC), while the VLISP Virtual Machine (VM) evaluates SBC programs to answers. Together, these evaluate Scheme programs to answers.

The semantics of SBC is given in terms of a state machine called the Stored Byte Code Machine (SBCM); the meaning of *sbc* is given by the answer computed by the SBCM when started in an initial state Σ_0 constructed from *sbc*. The correctness of the compiler as a whole asserts that if the SBCM computes an answer a when started in Σ_0, then a is the result of applying the denotation of p to suitable initial parameters $\vec{x}$.

The correctness of the VLISP Virtual Machine (VM) requires that when the VM evaluates *sbc* to an answer a, then the SBCM should compute a when started in an initial state constructed from *sbc*. In this section, we present the algorithms and correctness proofs of parts of the VLISP Virtual Machine relative to the SBCM.

There are three main reasons why the SBCM must be refined before being embodied in concrete program code. First, the size of the active store in the SBCM is unbounded in size, whereas any concrete implementation will have a limited address space. As a consequence, a garbage collector must be introduced. Second, other values are also bounded by the finite resources available to a concrete implementation. Third, the declarative specifications of some of the rules are not trivial to embody in concrete, procedural code. For this last reason, we will instantiate these rules using programs that manipulate a set of primitive operations that can be directly and reliably coded. We thus *refine* and *instantiate* the state machine description until we obtain a concrete implementation of the VM. *Refinement* is a binary relation between state machines; as formalized in Section 4, these state machines may have completely different state spaces.

Instantiation is a binary relation between machines sharing the same state space. Instantiation may reduce the non-determinism of a machine; however, it is useful even if the two state machines are identical, as is the case in Section 6.2. The reason is that the *description* of the state machine has changed; in place of the declarative rules, we have substituted algorithms calling simple primitive operations, and have thus moved closer to a concrete program.

We use refinement and instantiation in contrasting ways. We use refinement to alter the states and action rules of state machines. We use instantiation to specify algorithms that implement the action rules. In both cases, the minimal correctness criterion is that we should refine the observable behavior of the state machines, in the sense that every observable behavior of the more concrete machine should also be an observable behavior of the more abstract machine.

We will briefly describe the state machine SBCM. We proceed by alternately instantiating and refining the state machine as follows:

1. We define a collection of primitive state operations that observe and modify states. We *instantiate* the SBCM action rule descriptions by coding them in terms of these state operations. The state machine remains the same, while its description is altered (made more "concrete"). The correctness proof shows that the implementation of each action rule meets its specification; this is long and tedious.

2. We justify garbage collection in two main steps:

   (A) We *refine* the SBCM to the Garbage-Collected Stored Byte Code Machine (GCSM). This machine is similar to the SBCM except that it includes a new primitive state-modifying operation (`gc`). The refinement proof uses the storage layout relation technique to show that the GCSM refines the SBCM. The main difference between this proof and the proof of the flattener in Section 5 is in the definition of the storage layout relation.

   (B) We *instantiate* the `gc` primitive by presenting a concrete implementation that is defined in terms of the primitive state operations. The instantiation proof shows that the garbage collector meets the specification of the action rule.

3. We *weakly refine* the GCSM to a Finite, Garbage-Collected Stored Byte Code Machine (FGCSM). The FGCSM state set is a subset of the GCSM state set, namely the set of all states with components of bounded size. The length of stacks, the length of stores, and the values of registers, stack cells, and store cells receive upper bounds. The FGCSM initial states, halt states, action rules, and answer function are restrictions of the corresponding GCSM components. The proof of weak refinement is trivial.

4. We directly encode the FGCSM description in PreScheme; this is the concrete implementation of the VLISP Virtual Machine.

Our implementation also incorporates several optimizations, mentioned in Section 6.6; we have not justified these optimizations in this paper since they add complexity to the proofs but do not involve any new techniques.

6.1. Stored Byte Code and its State Machine

Stored Byte Code (SBC) language represents the image builder's output. Its operational semantics is given in terms of a deterministic state machine called the *Stored Byte Code Machine* (SBCM).

The abstract syntax of SBC is presented in Table 23. An SBC program is an SBC store together with an index into the store. A store is a sequence of cells, with each cell being one of:

1. A header $\langle \text{HEADER}\ h\ p\ m\rangle$ where h (the *header tag*) is a tag that describes the object occupying the storage that follows; p (the *mutability flag*) is a boolean value that describes whether the object is mutable; and m (the *header size*) is the number of cells in the stored object;
2. A pointer $\langle \text{PTR}\ l\rangle$ where l is an index into the store;
3. A number $\langle \text{FIXNUM}\ n\rangle$ where n is an integer;
4. An immediate value $\langle \text{IMM}\ imm\rangle$ where imm is a simple (unstructured) value such as a character, boolean, etc.
5. A byte b where b is a number that represents either a character or an instruction opcode.

A term of the form 2, 3, or 4 above is called a value descriptor. For each byte that represents an instruction opcode, we define a named constant and refer to the byte by this name, e.g., `call`, `return`, etc.

We will write a typical state of the SBCM in the form

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

where n is a number, t, v, u, k, and r are value descriptors, a is a stack of value descriptors and s is a store. The components of the state are called, in order, its template, offset, value, argument stack, environment, continuation, store, and spare.

The stores in accessible states of the SBCM, where again by an *accessible* state we mean a state that occurs in some trace of the machine, 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 l at which the first value is stored is called the address of the stored object, and the term $\langle \text{PTR}\ \ell\rangle$ is called a pointer to the stored object. We name the stored object by its header tag; thus we call a stored object with header tag TEMPLATE a "template" stored object.

Table 23. Abstract Syntax of Stored Byte Code (SBC).

$$
\begin{array}{rcl}
\mathit{program} & ::= & \langle s\ v\rangle \\
s & ::= & \mathit{cell}^* \\
\mathit{cell} & ::= & \mathit{desc} \mid \mathit{byte} \\
\mathit{desc} & ::= & \langle \text{HEADER}\ \mathit{htag}\ \mathit{bool}\ n\rangle \mid v \\
v & ::= & \langle \text{PTR}\ n\rangle \mid \langle \text{FIXNUM}\ n\rangle \mid \langle \text{IMM}\ \mathit{imm}\rangle \\
\mathit{imm} & ::= & \text{FALSE} \mid \text{TRUE} \mid \langle \text{CHAR}\ n\rangle \mid \text{NULL} \mid \text{UNDEFINED} \\
 & & \text{UNSPECIFIED} \mid \text{EMPTY-ENV} \mid \text{HALT} \mid \text{EOF} \\
\mathit{htag} & ::= & \mathit{bhtag} \mid \mathit{dhtag} \\
\mathit{bhtag} & ::= & \text{STRING} \mid \text{CODEVECTOR} \\
\mathit{dhtag} & ::= & \text{PAIR} \mid \text{SYMBOL} \mid \text{VECTOR} \mid \text{LOCATION} \mid \text{TEMPLATE} \mid \\
 & & \text{CLOSURE} \mid \text{PORT} \mid \text{CONTINUATION} \mid \text{ENVIRONMENT} \\
\mathit{byte} & ::= & n \\
a & ::= & v^* \\
\Sigma & ::= & \langle v_0\ n\ v_1\ a\ v_2\ v_3\ s\ v_4\rangle
\end{array}
$$

DEFINITION 13 **(Stored Objects)** *s contains a* stored object *at ℓ (or alternatively, ℓ* is a pointer to an object in *s) 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.*

If s contains a stored object at ℓ with $s(\ell - 1) = \langle \text{HEADER}\ h\ p\ m\rangle$ for some h, p, and m, then $\mathcal{S}(\langle \text{PTR}\ l\rangle, s)$ extracts the stored object in s at l:

$$\mathcal{S}(\langle \text{PTR}\ l\rangle, s) = \langle h\ p\ \langle s(l) \ldots s(l+m-1)\rangle\rangle$$

$\mathcal{H}(\langle \text{PTR}\ l\rangle, s)$ gives its header type $h = (\mathcal{S}(\langle \text{PTR}\ l\rangle, s))(0)$.

DEFINITION 14 **(Current Instruction, Parameters)**
Let $\Sigma = \langle t, n, v, a, u, k, s, r\rangle$ and let

$$\mathcal{S}(t, s) = \langle \text{TEMPLATE FALSE}\ c{::}v^*\rangle$$
$$\mathcal{S}(c, s) = \langle \text{CODEVECTOR FALSE}\ b^*\rangle$$

for some sequences v^*, b^*. Then,

Table 24. Action Rule for the `call` Instruction.

Rule 0**: call** m
Domain conditions:
$current\_inst(n, t, s) = \texttt{call}$
$current\_inst\_param(n, 1, t, s) = m$
$\mathcal{S}(v, s) = \langle \text{CLOSURE FALSE } \langle t_1 \ u_1 \ d_1 \rangle \rangle$
Changes:
$t' = t_1$
$n' = 0$
$u' = u_1$

$$current\_inst(n, t, s) = b^*(n)$$
$$current\_inst\_param(n, i, t, s) = b^*(n + i)$$

$current\_inst(n, t, s)$ is the current instruction, while $current\_inst\_param(n, i, t, s)$ is the ith parameter to the current instruction.

An SBCM state is initial if it is of the form

$$\langle t, 0, \langle \text{IMM UNSPECIFIED} \rangle, \langle \rangle, \langle \text{IMM EMPTY-ENV} \rangle, \langle \text{IMM HALT} \rangle, s, r \rangle$$

and satisfies the two state invariants in Lemma 19 below. A state $\langle t, n, v, a, u, k, s, r \rangle$ is a halt state if:

- $current\_inst(n, t, s) = \texttt{return}$
- $k = \langle \text{IMM HALT} \rangle$

As before, *acts* is the union of subfunctions called *(action) rules.* Table 24 presents the `call` rule as an example. As with the flattened byte code machine FBCM (Lemma 18), the SBCM is deterministic: for every state of the SBCM, 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. Two basic invariants hold for SBCM-accessible states.

LEMMA 19 **(SBCM Invariants)** *If an accessible state Σ with store s contains a term of the form $\langle \text{PTR } \ell \rangle$, then ℓ points to an object in s.*

If $\Sigma = \langle t, n, v, a, u, k, s, r \rangle$ is accessible, then

1. *$\mathcal{H}(t) = \text{TEMPLATE}$;*
2. *$n < \#(\mathcal{S}(c, s)(2))$, where $c = \mathcal{S}(t, s)(2)(1)$ is the code component of the template;*
3. *either $u = \langle \text{IMM EMPTY-ENV} \rangle$ or $\mathcal{H}(u) = \text{ENVIRONMENT}$; and*
4. *either $k = \langle \text{IMM HALT} \rangle$ or $\mathcal{H}(k) = \text{CONTINUATION}$.*

Table 25. SBCM State Observers for $\Sigma = \langle t, n, v, a, u, k, s, r\rangle$.

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	$= a(0)$	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` $d\ \Sigma$	$= m$

6.2. The Instantiated SBCM Description

The action rules of the SBCM are instantiated in terms of some primitive operations on states.

We define a set of primitive operations on the state, and we instantiate each action rule of the SBCM in terms of these operations. The state operations are presented in Tables 25 and 26. These operations 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 procedures that modify a component of a state. A "state modifier expression" consists of a state modifier name applied to observer expressions.

Each SBCM action rule is instantiated as a program composed from state observer and modifier 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. We will

Table 26. SBCM State Modifiers for $\Sigma = \langle t, n, v, a, u, k, s, r \rangle$.

Register Modifiers

`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 Modifiers

`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 Modifiers

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

Stored Object Modifiers

`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` $\langle \text{PTR}\ l \rangle\ i\ vd\ \Sigma$ $=$ $\Sigma[s' = s[(l + i) \mapsto vd]]$

```
(assert (closure-stob? (value-ref))
        "op/call: value register does not contain a closure")
(template-set
 (stob-ref (value-ref) 0))    ;; Closure's template
(env-set
 (stob-ref (value-ref) 1))    ;; Closure's env
(offset-set 0)
```

Figure 3. Instantiation of the `call` Action Rule.

call such a program a *rule program.* We will say that a rule program is *applicable* in a state Σ if the domain conditions of the associated rule are true in Σ.

The instantiations are exceedingly simple programs. In fact, only the rule *make-rest-list*, which makes an argument list—when calling a Scheme procedure with a dotted argument list—requires the `while` construct; the others can be instantiated using only a primitive recursive `for`-loop. The majority of rules require no iteration whatever. Thus, the instantiated programs are easily verified using Hoare logic.

Figure 3 presents the instantiation of the `call` action rule. The proof of its correctness consists of simple Hoare-style reasoning about the effects of this straight-line code.

THEOREM 10 *The instantiation of the* SBCM *action rules is correct with respect to the specification of the action rules.*

The proof is simple, but highly tedious. A detailed examination of the correctness of all the action rules was carried out. This process would have been eased had we had reasonable mechanized proof support, particularly during phases of development when the rule specifications and instantiations changed.

The SBCM is deterministic, so the instantiation is identically the same state machine: it has (extensionally) the same transition relation. Although the state machine remains the same, the instantiation provides a much more informative description of it. The instantiation determines the majority of the code for the VM.

6.3. The Garbage-Collected Stored Byte Code Machine (GSBCM)

In this section we define the Garbage Collected Stored Byte Code Machine (GCSM) and prove that it is a faithful refinement of the SBCM. As a consequence, we justify adding garbage collection to the instantiated form of the GCSM.

The SBCM and GCSM share the same set of states, the same initial states, halt states, and answer function. The only difference between them is that the GCSM includes a new primitive state modifier (called `gc`). It is not fully specified, so that the GCSM is not a deterministic state machine. We will define `gc` in terms of a relation $\cong$ of similarity between GCSM states and SBCM states. We define $\cong$ in turn using a variant of the storage-layout relations approach. The treatment in [18] assumes that terms are tree-structured, and storage layout relations are defined by structural induction over such terms. However, our 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 Σ and Σ^A 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 understandable 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 match 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.

6.3.1. State Similarity

We formalize locations as natural numbers, but use ℓ-like variables when we are regarding the numbers as store locations. We will typically indicate objects from the more abstract SBCM by variables with a superscript A, leaving variables representing GCSM objects unadorned. Since the SBCM 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. The superscript only connects the object with the machine we regard it as arising in.

DEFINITION 15 **(Store Correspondence)** *Suppose $\sim$ is a binary relation on natural numbers, regarded as representing concrete and abstract store locations. Then $\sim$ is a* store correspondence between *concrete store s* and *abstract store s^A if for all values of the location variables, the following three conditions are met:*

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

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

Definition 16 **(Term Correspondence)** *Let $\sim$ be a store correspondence between stores s and s^A.*

The term correspondence generated by $\sim$ *is a relation $\simeq$ between cells (i.e., value descriptors or bytes)* cell *and* cell^A*, defined by taking cases on* cell^A*.* $\mathrm{cell} \simeq \mathrm{cell}^A$ *holds just in case one of the following holds:*

1. $\mathrm{cell}^A = \langle \textsc{ptr}\ l^A \rangle$ *and*
   (A) $\langle \textsc{ptr}\ l \rangle = \mathrm{cell}$*;*
   (B) $s^A(l^A - 1) = s(l-1) = \langle \textsc{header}\ h\ p\ m \rangle$*, for some header type h, mutability bit p and stored object length m; and*
   (C) *for all $0 \leq i < m$, $(l+i) \sim (l^A + i)$.*
2. $\mathrm{cell}^A = \langle \textsc{fixnum}\ m \rangle = \mathrm{cell}$*, for some number m.*
3. $\mathrm{cell}^A = imm = \mathrm{cell}$*, for some immediate value imm.*
4. $\mathrm{cell}^A = b = \mathrm{cell}$*, for some byte b.*

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

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

Definition 17 **(State Similarity)** *Let $\sim$ be a store correspondence between concrete* GCSM *store s and abstract* SBCM *store s^A, with generated term correspondence $\simeq$.*

Σ and Σ^A, of the forms

$$\langle t, n, v, a, u, k, s, r \rangle \quad \textit{and} \quad \langle t^A, n^A, v^A, a^A, u^A, k^A, s^A, r^A \rangle,$$

respectively, are similar with respect to $\sim$*, written $\Sigma \approx \Sigma^A$, just in case the invariants on* SBCM *states given in Lemma 19 hold and:*

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

Finally, Σ and Σ^A are similar*, written $\Sigma \cong \Sigma^A$, just in case there exists a store correspondence $\sim$ such that Σ and Σ^A 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 we assume they are generated by the same omitted $\sim$. If Σ and Σ^A are similar with respect to $\sim$, then we will call it a *witness* for $\Sigma \cong \Sigma^A$. By inspecting the structure of states we can verify:

LEMMA 20 **(Properties of $\cong$)**

1. *$\cong$ is an equivalence relation;*
2. *$\Sigma \cong \Sigma^A \Rightarrow ans(\Sigma){=}{=}ans(\Sigma^A)$, where ans is defined as in Section 3.1, and $s{=}{=}t$ means that s and t are equal if either of them is well defined (see Section 4.2); hence,*
3. *$\Sigma \cong \Sigma^A \Rightarrow \Sigma \in halts$ iff $\Sigma^A \in halts$.*

6.3.2. Legitimacy of Garbage Collection

We now define the GCSM to be a state machine having the same states, initial states, halt states, and answer function as SBCM. It differs from the SBCM only in the tuple component *acts*. We regard any transition from Σ to Σ' where $\Sigma \cong \Sigma'$ as a potential garbage collection, and allow garbage collection to occur non-deterministically in the course of executing any rule program immediately before executing the storage-allocating modifier `make-stob`.

To formalize this idea, we add one state modifier to the ones that were used in the SBCM, and we also add one keyword to the programming language for building up rule programs. The modifier is `gc`. Its specification is non-deterministic: `gc` Σ returns some state Σ' such that $\Sigma \cong \Sigma'$. The keyword is `maybe`; if e is a modifier expression, then so is `maybe` e. It also introduces non-determinism. If the modifier expression e, executed in state Σ, may return Σ', then so may `maybe` e, and moreover `maybe` e may do nothing and return Σ unchanged.

Because $\cong$ happens to be a reflexive relation, there is in fact no difference between `gc` and `maybe gc`. Nevertheless, we introduce both because the two kinds of non-determinism get resolved differently. The non-determinism of `gc` is resolved in Section 6.4, when we select a particular garbage collection algorithm. The non-determinism introduced by the `maybe` is resolved when we pass to a finite machine in Section 6.5. At that point the stores are finite, and garbage collection can occur when they are nearly full.

The transition relation of the GCSM is defined implicitly by means of rule programs. If `P` is one of the rule programs of the SBCM, then `P`$'$ is defined to be the result of replacing every occurence of (`make-stob` h p m) by

```
(maybe (gc))
(make-stob h p m)
```

If `P`$'$ derives from a SBCM rule program `P` in this way, we will say that `P`$'$ *corresponds to* `P`. We define *acts* to be the set of pairs $\langle \Sigma, \Sigma' \rangle$ such that for some SBCM rule program `P`, `P` is applicable in Σ, and the corresponding program $\mathtt{P}'$, when started in Σ, may terminate in Σ'. Thus, the domain conditions for the SBCM rules are carried over in the GCSM; the changes of state are the same, except that `gc` is permitted.

We want to establish that $\cong$ is a storage layout relation between SBCM and GCSM. Clause 1 of Lemma 20 implies Clause 1 of Definition 11, while Clause 2 of the lemma is the same as Clause 4 of Definition 11. By Lemma 12 and Theorem 6, to show that GCSM refines SBCM it suffices to show the safety and liveness conditions hold.

The safety condition asserts that for every pair of similar states $\Sigma \cong \Sigma^A$, if Σ can proceed to Σ_1, then either $\Sigma_1 \cong \Sigma^A$ or else Σ^A can proceed to some Σ_1^A such that $\Sigma_1 \cong \Sigma_1^A$. In fact, the latter always holds, so the liveness condition is also satisfied.

To prove this, we will show

1. If $\Sigma \cong \Sigma^A$, then the domain conditions for a rule either hold in both states or else fail in both states (Theorem 11); and
2. If $\Sigma_0 \cong \Sigma_0^A$ and $\mathtt{P}'$ corresponds to `P`, then either both programs fail to terminate when started in those states (respectively) or else both programs terminate, and the results satisfy $\cong$ (Theorem 12).

For both these results, the real heart of the matter is that the individual observers and modifiers are invariant under $\cong$. We will separate observers and modifiers in the following two lemmas, each of which is proved [16] by perfectly straightforward (but lengthy) applications of the definitions:

LEMMA 21 **(Observers respect $\simeq$)** *Let $\sim$ be a store correspondence, with generated term correspondence $\simeq$; let e be any of the state observers listed in Table 25, taking state parameter and possibly other parameters. Suppose:*

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

2. the respective components of $\vec{x}$ and $\vec{x}^A$ satisfy $\simeq$.

Then $e(\Sigma, \vec{x}) \simeq e(\Sigma^A, \vec{x}^A)$.

LEMMA 22 **(State modifiers preserve $\cong$)** *Let $\sim$ be a store correspondence, with generated term correspondence $\simeq$; let e be any of the state modifiers listed in Table 26, taking state parameter and possibly other parameters. Suppose:*

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

2. the respective components of $\vec{x}$ and $\vec{x}^A$ satisfy $\simeq$.

Then $e(\Sigma, \vec{x}) \cong e(\Sigma^A, \vec{x}^A)$.

In the proof of Lemma 22, the underlying store correspondence $\sim$ is used unaltered for every modifier 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 23 *Let $\sim$ be a store correspondence, with generated term correspondence $\simeq$. Suppose:*

1. *e and e' are observer expressions;*

2. *$\Sigma \approx \Sigma^A$; and*

3. *the respective components of $\vec{x}$ and $\vec{x}^A$ satisfy $\simeq$;*

4. *the respective components of $\vec{y}$ and $\vec{y}^A$ satisfy $\simeq$;*

Then $e(\Sigma, \vec{x}) = e'(\Sigma, \vec{y})$ iff $e(\Sigma^A, \vec{x}^A) = e'(\Sigma^A, \vec{y}^A)$.

Proof: By structural induction on observer expressions, using Lemma 21 and the definition of $\approx$. ■

Theorem 11 *Let $\sim$ be a store correspondence. Suppose:*

1. *R is any* SBCM *rule (with auxiliary variables);*

2. *The domain condition for R is satisfied in Σ using the witnessing values $\vec{x}$ for the auxiliary variables;*

3. *$\Sigma \approx \Sigma^A$;*

4. *Σ consists of stored objects.*

Then there is an $\vec{x}^A$ such that:

1. *$\#\vec{x} = \#\vec{x}^A$;*

2. *$\vec{x}(i) \simeq \vec{x}^A(i)$, for each i where $0 \leq i < \#\vec{x}$;*

3. *The domain condition for R is satisfied in Σ^A using the witnessing values $\vec{x}^A$.*

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

On the assumption there is such an ℓ', then it follows that conditions 1 and 2 are satisfied. Condition 3 follows from Lemma 23, because the domain conditions are always equations between observer expressions.

Thus, it suffices to show that there is an ℓ' such that $\ell \sim \ell'$, whenever $\langle \text{PTR } \ell \rangle$ is a witnessing value for a domain condition auxiliary variable. By Definition 15 clause 1, there is at most one such ℓ', so $\vec{x}^A$ is actually unique.

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 17 clause 3 to infer that, if v is the value of r_0 in Σ^A, then $\langle \text{PTR } \ell \rangle \simeq v$. By Definition 16, v is of the form $\langle \text{PTR } \ell^A \rangle$, and by clause 1c, $\ell \sim \ell^A$. The remaining cases are similar, except that Definitions 17 and 16 must be used repeatedly. ■

THEOREM 12 **(Corresponding programs produce corresponding results)** *Suppose that $\Sigma_0 \cong \Sigma_0^A$, and that* `P′` *and* `P` *are corresponding rule programs. Then, when* `P′` *and* `P` *are started in states Σ_0 and Σ_0^A respectively, then either*

1. *both fail to terminate; or else*
2. *both terminate, and in states Σ_1 and Σ_1^A, respectively, such that $\Sigma_1 \cong \Sigma_1^A$.*

Proof: By induction on the sequence of states traversed as `P′` and `P` compute, using Lemmas 22 and 23. ■

From Lemma 20 and Theorems 11 and 12, we may conclude:

THEOREM 13 *The* GCSM *refines the* SBCM.

6.4. The Instantiated GSBCM Description

We instantiate the `gc` primitive by a specific garbage collection algorithm *gc* and we prove that it satisfies the property that $\text{gc}(\Sigma) \cong \Sigma$. 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 Table 4. 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. It in effect translates the store to the left, adjusting all of the pointers to match.

DEFINITION 18 **(Shift_store, Shift)** *If s is a store, then*

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

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

The result of *shift*(Σ, j) is a well-formed state if $j \leq \ell$ whenever $\langle \text{PTR}\ \ell \rangle$ is a component of Σ, and $j \leq \ell$ whenever $\langle \text{PTR}\ \ell \rangle$ occurs at a location ℓ_0 in its store such that $j \leq \ell_0$. This condition ensures that a pointer always has a nonnegative value after the translation.

The garbage collector, which implements the algorithm given in Figure 4, uses several auxiliary functions. 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 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 pointer to the new object, called a *broken heart*; all other pointers to the old stored object will eventually be modified to point to the new object. If the object has already been copied to the new store, *gc-convert* returns 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. 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.

We regard the garbage collector as an algorithm rather than simply a function. In the course of executing, the garbage collector modifies the store component of the state repeatedly. We say somewhat informally—for instance, in Lemma 25—that a store s *occurs* in a run of the garbage collection algorithm. By this we mean that *gc-trace* returns a pair $\langle d\ s \rangle$ on some "call" from gc_0, *gc-trace-stack*, or *gc-scan-heap*.

Garbage collection is indeed a non-trivial algorithm and it transforms the store through a succession of intermediate values. Therefore, the proof requires the use of a relation similar to $\cong$ to express an invariant maintained by the algorithm. 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 19 **(GC Term Correspondence, GC State Similarity)**
Let $\sim$ be a store correspondence between GCSM *stores s and s'.*

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

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

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

$$
\begin{array}{l}
\textit{gc-trace-stack}(a, s) = \langle a'\ s_{\#a}\rangle \\
\quad \text{where } s_0 = s \text{ and } \textit{gc-trace}(a(i), s_i) = \langle a'_i\ s_{i+1}\rangle \\
\qquad \text{for } 0 \leq i < \#a \\
\quad \text{and } a' = \langle a'_0\ a'_1\ \ldots a'_{\#a-1}\rangle
\end{array}
$$

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

$$
\begin{array}{l}
gc_0(\Sigma, n_0) = \langle t', n, v', a', u', k', s_7, r'\rangle \\
\quad \text{where } \textit{gc-trace}(t, s) = \langle t'\ s_1\rangle \\
\quad \text{and } \textit{gc-trace}(v, s_1) = \langle v'\ s_2\rangle \\
\quad \text{and } \textit{gc-trace}(u, s_2) = \langle u'\ s_3\rangle \\
\quad \text{and } \textit{gc-trace}(k, s_3) = \langle k'\ s_4\rangle \\
\quad \text{and } \textit{gc-trace-stack}(a, s_4) = \langle a'\ s_5\rangle \\
\quad \text{and } \textit{gc-trace}(r, s_5) = \langle r'\ s_6\rangle \\
\quad \text{and } \textit{gc-scan-heap}(s_6, n_0) = s_7
\end{array}
$$

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

Figure 4. Garbage Collection Algorithm

The GC term correspondence generated by $\sim$ *is a relation* $\simeq^{gc}$ *between cells (i.e., value descriptors or bytes)* cell *and* cell$'$, *defined by taking cases on* cell$'$. cell $\simeq^{gc}$ cell$'$ *holds just in case one of the following holds:*

1. cell$' = \langle \text{PTR}\ l' \rangle$ *and* cell $= \langle \text{PTR}\ l \rangle$. *Let* l_1 *be* l *unless* $s(l-1)$ *is a broken heart* $\langle \text{PTR}\ l_0 \rangle$, *in which case let* l_1 *be* l_0. *Let* l'_1 *be* l' *unless* $s'(l'-1)$ *is a broken heart* $\langle \text{PTR}\ l'_0 \rangle$, *in which case let* l'_1 *be* l'_0. *We require further:*
   - *(A)* $s'(l'_1 - 1) = s(l_1 - 1) = \langle \text{HEADER}\ h\ p\ m \rangle$, *for some header type* h, *mutability bit* p *and stored object length* m; *and*
   - *(B) for all* $0 \leq i < m$, $(l_1 + i) \sim (l'_1 + i)$.
2. cell$' = \langle \text{FIXNUM}\ m \rangle =$ cell, *for some number* m.
3. cell$' = imm =$ cell, *for some immediate value* imm.
4. cell$' = b =$ cell, *for some byte* 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. $\#a' = \#a$ *and* $\forall 0 \leq i < \#a \,.\, a(i) \simeq^{gc} a'(i)$;
2. *for all locations* l' *and* l, $l \sim l' \Rightarrow s(l) \simeq^{gc} s'(l')$;
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* $\sim$.

LEMMA 24 $\cong^{gc}$ *is a transitive relation. That is,*

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

Proof: Compose the witness store correspondences. ■

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

DEFINITION 20 **(GC Stored Objects)** *If* $s(l-1)$ *is a broken heart* $\langle \text{PTR}\ l_0 \rangle$, *let* $l_1 = l_0$; *let* $l_1 = l$ *otherwise. Then* s *contains a* GC stored object *at* l *if:*

1. $s(l_1 - 1) = \langle \text{HEADER}\ h\ p\ m \rangle$, *for some* h, p, *and* m;
2. $s(j)$ *is a well-defined non-header value for each* j *for* $l_1 \leq j < l_1 + m$.

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

LEMMA 25 *1. Suppose that s is a store encountered (i.e., returned by* gc-trace*) during garbage collection of a* GCSM *accessible state; then s consists of GC stored objects.*

2. If Σ consists of stored objects and $\Sigma' = \mathrm{gc}(\Sigma)$, then Σ' consists of stored objects.

Proof: 1. This holds of every accessible GCSM store prior to garbage collection. During garbage collection, all mutations of the store are made via the function *gc-trace* which preserves this property.

2. Since Σ consists of stored objects, it consists (*a fortiori*) of GC stored objects. Using part 1 inductively, the same is true of $\mathrm{gc}_0(\Sigma)$. The property of consisting of GC stored objects is preserved by *shift*, so $\Sigma' = \mathit{shift}(\mathrm{gc}_0(\Sigma), \#s)$ consists of GC stored objects. Finally, Σ' contains no broken hearts, so it also consists of stored objects. ■

This establishes that *gc* preserves the first invariant of Lemma 19; the properties given in its second clause are straightforward consequences of the algorithm.

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 26 *Let s be a store encountered (i.e., returned by* gc-trace*) during garbage collection of a* GCSM *accessible state Σ, and let d be a component of Σ. Let*

$$\text{gc-trace}(d, s) = \langle d'\ s' \rangle$$

for some d', s'. Suppose d is not a pointer, or that $d = \langle \text{PTR}\ l \rangle$ and s contains a GC stored object at l.

Then there exists a store correspondence $\sim$ between s and s' with generated GC term correspondence $\simeq^{gc}$ such that

1. $d' \simeq^{gc} d$

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

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

Case: $d = \langle \text{PTR}\ l \rangle$ and $s(l-1) = \langle \text{HEADER}\ h\ p\ m \rangle$, so, by definition of *gc-trace*, $d' = \langle \text{PTR}\ l_0 \rangle$, $l_0 = \#s + 1$, and

$$\begin{aligned} s' = s&[l_0 - 1 \mapsto s(l-1)][l_0 \mapsto s(l)] \ldots [l_0 + m - 1 \mapsto s(l + m - 1)] \\ &[l - 1 \mapsto \langle \text{PTR}\ l_0 \rangle] \end{aligned}$$

Define $\sim$ as follows:

$$\begin{array}{rl} l_0 + i \sim l + i & \text{for } 0 \le i < m \\ x \sim x & \text{for all } x \in \mathit{dom}(s) \text{ such that } s(x) \text{ is not a header} \\ & \text{and } x \neq l + i \text{ for } 0 \le i < m. \end{array}$$

1. $s(l-1) = \langle \text{HEADER } h\ p\ m\rangle$ by case condition. $s'(l_0-1) = s(l-1) = \langle \text{HEADER } h\ p\ m\rangle$ by definition of s'. For all $0 \leq i < m$, $l_0 + i \sim l + i$ holds by definition of $\sim$. Thus, by definition of $\simeq^{gc}$,

   $$\langle \text{PTR } l_0\rangle \simeq^{gc} \langle \text{PTR } l\rangle.$$

2. We first show that if vd is a value in one of the registers, or a component of the stack, then $vd \simeq^{gc} vd$. First, if vd is not a pointer, $vd \simeq^{gc} vd$ holds by definition of $\simeq^{gc}$.

   Second, suppose $vd = \langle \text{PTR } y\rangle$ where $y \neq l$. Let $x = l'$ if $s(y-1)$ is a broken heart $\langle \text{PTR } l'\rangle$ and let $x = y$ otherwise. Then $s(x-1) = s'(x-1) = \langle \text{HEADER } h'\ p'\ m'\rangle$ for some h', p', m', and $x+i \sim x+i$ for all $0 \leq i < m'$. Thus $\langle \text{PTR } y\rangle \simeq^{gc} \langle \text{PTR } y\rangle$ holds by definition of $\simeq^{gc}$.

   Finally, if $vd = \langle \text{PTR } l\rangle$, then we know that $s(l-1) = \langle \text{HEADER } h\ p\ m\rangle$ by the case condition. Further, $s'(l-1) = \langle \text{PTR } l_0\rangle$ and $s'(l_0-1) = \langle \text{HEADER } h\ p\ m\rangle$ by definition of s'. Finally, for all $0 \leq i < m$, $l_0 + i \sim l + i$ holds by definition of $\sim$. Thus, $\langle \text{PTR } l\rangle \simeq^{gc} \langle \text{PTR } l\rangle$ holds by definition of $\simeq^{gc}$.

   To complete the proof that $\langle t, n, v, a, u, k, s', r\rangle \cong^{gc} \Sigma$, we need to show that:

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

   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, $x' = l_0 + i$ by definition of $\sim$. Now $s'(x') = s'(l_0 + i) = s(l + i) = s(x)$ by definition of s'. The result then follows since $vd \simeq^{gc} vd$ was shown above for any component of the state.

■

LEMMA 27 *Let s_0 and s consist of stored objects, where s_0 is a prefix of s and $\#s_0 \leq l$. Let $\Sigma = \langle t, n, v, a, u, k, s, r\rangle$, 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, $\langle t, n, v, a, u, k, \text{gc-scan-heap}(s, \#s_0), r\rangle \cong^{gc} \Sigma$.

Proof: By Lemma 26, $\langle t, n, v, a, u, k, s', r\rangle \cong^{gc} \Sigma$ with $\sim$ as the witness store correspondence. Let $\sim$ extend to the term correspondence $\simeq^{gc}$. 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 $s(l) = \langle \text{PTR } l_0 \rangle$ for some l_0, then by definition of *gc-trace*, $s(l) = s'(l) = \langle \text{PTR } l_0 \rangle$ since *gc-trace* only modifies $s(l_0 - 1)$ if it is a header and hence $l_0 - 1 \neq l$. Thus, we have that $d' \simeq^{gc} s'(l) = \langle \text{PTR } l_0 \rangle$. 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 24).

Let $\sim$ be the identity function on all defined locations of s' that are not headers of stored objects, and generate the 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 $\langle \text{PTR } l_0 \rangle$ it depends on $s'(l_0 - 1)$ and $s'[l \mapsto d'](l_0 - 1)$ being the same header value. This holds since we know that $s'(l)$ is not a header and hence $l \neq l_0 - 1$.

Moreover, $s'[l \mapsto d'](l) \simeq^{gc} s'(l)$, because, as we showed above, $d' \simeq^{gc} s'(l) = \langle \text{PTR } l_0 \rangle$.

Finally, we must show that *gc-scan-heap* terminates. We assume that the original store s_0 consists of stored objects, and that s has the property that if $i < \#s_0$ and $s_0(i)$ is not a header, then $s(i) = s_0(i)$. Let $\langle d, s' \rangle = \textit{gc-trace}(s(m), s)$, let $s'' = s'[m \mapsto d]$, and let $\#s_0 \leq m$. We have:

1. s, s', and s'' all contain the same total number of headers;
2. Assuming
$$\forall n \geq m, \forall l' \;.\; s(n) = \langle \text{PTR } l' \rangle \Rightarrow l' < \#s_0,$$
then we may infer
$$\forall n \geq m+1, \forall l' \;.\; s''(n) = \langle \text{PTR } l' \rangle \Rightarrow l' < \#s_0,$$
using the assumptions that s_0 consists of stored objects and agrees with s on non-headers.
3. If $\#s < \#s''$, then s'' has one fewer header below $\#s_0$ than did s.

■

The correctness of gc_0 now follows from Lemmas 24, 26, and 27.

LEMMA 28 *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 29 *Let* $\Sigma = \langle t, n, v, a, u, k, s, r \rangle$ *be a* GCSM *state and let*

$$\mathrm{gc}_0(\Sigma) = \langle t_1, n_1, v_1, a_1, u_1, k_1, s_1, r_1 \rangle.$$

If, for some l_0, *one of* t_1, v_1, u_1, k_1, r_1, $a_1(i)$ *for* $0 \leq i < \#a$, *or* $s_1(i)$ *for* $\#s \leq i$ *is the value* $\langle \text{PTR } l_0 \rangle$, *then*

1. $l_0 \geq \#s$;

2. $s_1(l_0 - 1) = \langle \text{HEADER } h\ p\ m \rangle$ *for some* h, p, m.

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

Let s' be a store encountered during garbage collection of Σ such that if s' contains a header of the form $\langle \text{PTR } l_0 \rangle$, then $l_0 > \#s$ and $s'(l_0 - 1) = \langle \text{HEADER } h\ p\ m \rangle$ for some h, p, m. If d' is the value of a state component of Σ, and $gc\text{-}trace(d', s') = (d'', s'')$, then we prove that s'' also satisfies the above property. That is, we prove that if s'' contains a header of the form $\langle \text{PTR } l_0 \rangle$, then $l_0 > \#s$ and $s'(l_0 - 1) = \langle \text{HEADER } h\ p\ m \rangle$ for some h, p, m. If s' also contains the same header $\langle \text{PTR } l_0 \rangle$, this follows by induction hypothesis. Otherwise, by definition of $gc\text{-}trace$,

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

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 $gc\text{-}trace$).

By induction, we have that all intermediate stores during garbage collection satisfy the property. But it follows from the definition of $gc\text{-}trace$ that given such stores, if $gc\text{-}trace$ returns a pointer $\langle \text{PTR } l_0 \rangle$, then $l_0 \geq \#s$ and $s_1(l_0 - 1) = \langle \text{HEADER } h\ p\ m \rangle$ for some h, p, m. ■

Finally, combining Lemmas 25, 28 and 29, we have:

THEOREM 14 **(Garbage Collection preserves $\cong^{gc}$, $\cong$)** *If Σ consists of stored objects, then*

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

and

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

Implementation Note. Our proofs are proofs about algorithms, while of course our implementation consists of code. Thus, we have taken care to ensure that our code reflects the algorithms it implements in as lucid a way as possible. There is, however, one complication in matching the VM program to its specification. It is due to the fact that VLISP PreScheme, in the version used in this work, compiles to a language in which all procedure calls are in tail recursive position. Thus, where a procedure call is not in tail recursive position, the Front End must substitute the code for the called procedure in line. There are several instructions in the SBCM that cause memory to be allocated, and each of them, when coded in the obvious way, contains a non-tail-recursive 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, because the original procedure must 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 only a little lucidity as a consequence of this programming approach. Our PreScheme implementation has since been improved; procedure calls in non-tail-recursive position need no longer be expanded in line [13].

6.5. The Finite Stored Byte Code Machine (FSBCM)

A finite state machine (with any fixed word size, bounded store size, and bounded stack size) cannot simulate all the computation histories of the GCSM, because there will always be computations that require more pointers than can be stored in words, stores, or stacks of the given size. We define a Finite Stored Byte Code Machine (FGCSM) and prove that it is a *weak* refinement (see Section 4) of the GCSM.

Table 27 contains the abstract syntax for the states of the FGCSM. The definition ensures that they form a proper subset of the states of SBC. The initial (halt) FGCSM states are the FGCSM states that are GCSM initial (halt) states. The FGCSM action rules are the GCSM action rules with domain and range restricted to FGCSM states. Thus, if an FGCSM action rule is defined on an FGCSM state, then its value at that state is the same as that of the corresponding GCSM action rule. The implemented values of `max-store-size` and of the upper and lower bounds for fixed point numbers reflect a 32-bit word size in which two bits are reserved for tags.

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

THEOREM 15 *Let $\sim$ be the identity relation on* FGCSM *states. Then $\sim$ is a weak storage layout relation between* FGCSM *and* GCSM, *and* FGCSM *weakly refines* GCSM.

6.6. Concrete Virtual Machine

A concrete implementation of the Virtual Machine is obtained by directly encoding the FGCSM as a PreScheme program. The actual VLISP Virtual Machine implementation differs from the implementation described here in several ways:

1. The actual implementation is designed to run on machines with 32-bit physical memory words. Since there are fewer than 256 distinct FSBC bytes, the implementation optimizes the representations of "string" and "codevector" stored

Table 27. Abstract Syntax of Finite Stored Byte Code (FSBC).

$$
\begin{array}{rcl}
\mathit{program} & ::= & \mathit{term} \\
\mathit{term} & ::= & \langle \mathit{store}\ \mathit{vdesc} \rangle \\
\mathit{store} & ::= & \mathit{cell}^* \qquad \mathit{with}\ \#\mathit{cell}^* < \texttt{max-store-size} \\
\mathit{stack} & ::= & \mathit{vdesc}^* \quad \mathit{with}\ \#\mathit{vdesc}^* < \texttt{max-stack-size} \\
\mathit{cell} & ::= & \mathit{desc} \mid \mathit{byte} \\
\mathit{desc} & ::= & \langle \textsc{header}\ \mathit{htag}\ \mathit{bool}\ \mathit{nat}_1 \rangle \mid \mathit{vdesc} \\
\mathit{vdesc} & ::= & \langle \textsc{ptr}\ \mathit{nat}_2 \rangle \mid \langle \textsc{fixnum}\ \mathit{int} \rangle \mid \langle \textsc{imm}\ \mathit{imm} \rangle \\
\mathit{imm} & ::= & \textsc{false} \mid \textsc{true} \mid \langle \textsc{char}\ \mathit{nat}_3 \rangle \mid \textsc{null} \mid \textsc{undefined} \\
 & & \textsc{unspecified} \mid \textsc{empty-env} \mid \textsc{halt} \mid \textsc{eof} \\
\mathit{htag} & ::= & \mathit{bhtag} \mid \mathit{dhtag} \\
\mathit{bhtag} & ::= & \textsc{string} \mid \textsc{codevector} \\
\mathit{dhtag} & ::= & \textsc{pair} \mid \textsc{symbol} \mid \textsc{vector} \mid \textsc{location} \mid \textsc{template} \mid \\
 & & \textsc{closure} \mid \textsc{port} \mid \textsc{continuation} \mid \textsc{environment} \\
\mathit{byte} & ::= & \mathit{nat}_4 \\
\mathit{nat}_1 & ::= & 0..\texttt{max-object-size} \\
\mathit{nat}_2 & ::= & 0..\texttt{max-pointer-size} \\
\mathit{nat}_3 & ::= & 0..127 \\
\mathit{nat}_4 & ::= & 0..255 \\
\mathit{int} & ::= & \texttt{min-fixnum-size}..\texttt{max-fixnum-size}
\end{array}
$$

objects by packing the bytes within them; specifically, four bytes are packed into a single store cell.

2. The actual implementation represents both the argument stack and the store within a single vector (called the *heap*). The representation is non-trivial and is designed to optimize certain expensive action rules. Specifically, this optimization permits the action rules `make-env` and `make-cont` to create environment and continuation stored objects respectively, without copying the contents of the argument stack to the store. This results in a much faster interpreter.

3. The actual implementation contains two heaps, only one of which is in active use. During garbage collection, it copies the reachable contents of the active heap to the inactive heap and switches the roles of the heaps. In this paper, the state machines have a single store, and the garbage collector copies reachable objects to the end of the current store.

4. The correctness proof of the garbage collector requires that all stored objects in the store have active value fields. To meet this condition, the state operation `make-stob` is specified (see Table 26) to create a new stored object and to initialize the value fields of this object to a distinctive value ($\langle$IMM UNDEFINED$\rangle$). Since this is time-consuming, the actual implementation does not initialize objects during allocation, but explicitly writes active values into new objects immediately after allocation. The garbage collector is not invoked until these writes have been performed.

5. The actual implementation represents FSBC *cells* as 32-bit binary numerals.

The proofs presented here have been extended to accommodate these optimizations.

7. Conclusion

Although the VLISP verification is quite comprehensive, and covers the great majority of the detailed implementation, there are a number of unverified aspects:

- 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 PreScheme Front End [12, 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.

To have fully verified these aspects would have greatly increased the scale of the work.

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 [7].

List of Tables

List of Figures

References

1. William Clinger. The Scheme 311 compiler: An exercise in denotational semantics. In *1984 ACM Symposium on Lisp and Functional Programming*, pages 356–364, New York, August 1984. The Association for Computing Machinery, Inc.
2. William M. Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, 55(3):1269–91, 1990. Also MITRE Corporation technical report M88-52, 1988; revised 1990.
3. William M. Farmer, Joshua D. Guttman, Leonard G. Monk, John D. Ramsdell, and Vipin Swarup. The faithfulness of the VLISP operational semantics. M 92B093, The MITRE Corporation, September 1992.
4. William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: an Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11(2):213–248, October 1993.
5. Joshua D. Guttman, Leonard G. Monk, William M. Farmer, John D. Ramsdell, and Vipin Swarup. The VLISP byte-code compiler. M 92B092, The MITRE Corporation, September 1992.
6. Joshua D. Guttman, Leonard G. Monk, William M. Farmer, John D. Ramsdell, and Vipin Swarup. The VLISP flattener. M 92B094, The MITRE Corporation, 1992.
7. Joshua D. Guttman, John D. Ramsdell, and Mitchell Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Computation*, 8(1/2):???–???, 1995.
8. C. A. R. Hoare. Notes on data structuring. In O.-J. Dahl, editor, *Structured Programming*. Academic Press, 1972.
9. IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.
10. Richard A. Kelsey and Jonathan A. Rees. Scheme48 progress report. Manuscript in preparation, 1992.
11. Robin Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs, 1989.
12. Dino P. Oliva, John D. Ramsdell, and Mitchell Wand. The VLISP verified PreScheme compiler. *Lisp and Symbolic Computation*, 8(1/2):???–???, 1995.
13. John D. Ramsdell. The revised VLISP PreScheme front end. M 93B095, The MITRE Corporation, August 1993.
14. Jonathan A. Rees, Norman I. Adams, and James R. Meehan. *The T Manual*. Computer Science Department, Yale University, fifth edition edition, 1988.
15. Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, 1977.
16. Vipin Swarup, William M. Farmer, Joshua D. Guttman, Leonard G. Monk, and John D. Ramsdell. The VLISP byte-code interpreter. M 92B097, The MITRE Corporation, September 1992.
17. Mitchell Wand. Semantics-directed machine architecture. In *Conf. Rec. 9th ACM Symp. on Principles of Prog. Lang.*, pages 234–241, 1982.
18. Mitchell Wand and Dino P. Oliva. Proving the correctness of storage representations. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 151–160, New York, 1992. ACM Press.