VLISP Byte Code Interpreter

Vipin Swarup William M. Farmer Joshua D. Guttman
Leonard G. Monk John D. Ramsdell

The MITRE Corporation[1]
M92B097
September 1992

[1]This work was supported by Rome Laboratories of the United States Air Force, contract No. F19628-89-C-0001.
Authors' address: The MITRE Corporation, 202 Burlington Road, Bedford MA, 01730-1420.

Abstract

The Verified Programming Language Implementation project has developed a formally verified implementation of the Scheme programming language. This report documents the byte-code interpreter, which executes a user program presented in the form of a binary image. It specifies several intermediate abstract machines, and proves that they are (successive) faithful implementations of the Vlisp operational semantics. Garbage collection and the finiteness of computer memories are among the issues handled.

Contents

1 Introduction

The VLISP byte code compiler translates Scheme programs into VLISP Stored Byte Code (SBC) programs. This report presents and justifies the translation of SBC programs to VLISP Finite Stored Byte Code (FSBC) programs, and the VLISP implementation of an interpreter for FSBCprograms. FSBC is a "finite" subset of SBC. The syntax of SBC is presented in the VLISP report on the image builder [5]. That report also presents an operational semantics of SBC (and hence of FSBC) in terms of a state machine called the Stored Byte Code Machine (SBCM). We do not reproduce the syntax and SBCM semantics of Stored Byte Code but refer the reader to the VLISP report that describes that language [5].

This report shall address several issues related to the implementation of a practical interpreter:

1. The interpreter has operations that create environments and continuations by copying an argument stack. By changing the representation of stores and stacks, it is possible to eliminate this copying. A minor modification to this representation permits continuations to be created and restored in a stack-like fashion; the result is a much faster interpreter.

2. A physical computer has finite memory, i.e., it has a fixed number of memory cells. Since the VLISP interpreter makes heavy use of memory, its practicality depends on automatic storage reclamation; this is provided via a garbage-collector that reclaims memory cells whose contents do not affect the behavior of the interpreter.

3. Another factor in the finiteness of a physical computer's memory is that each memory cell can only contain a number within fixed bounds. The VLISP interpreter terminates gracefully (with an error message) if its memory bounds are ever exceeded.

This report presents:

1. an operational semantics of SBC in terms of a state machine called the microcoded stored byte code machine (MSBCM). This machine incorporates a storage optimization which permits the implementation to create environments and continuations without copying the argument stack. It also refines the SBCM so that the action rules of the state machine can be expressed as the composition of a small set of primitive functions.

2. a proof that the operational semantics of SBC as given by the MSBCM is equivalent to the operational semantics of SBC as given by the SBCM.

3. an operational semantics of SBC in terms of a state machine called the garbage-collected stored byte code machine (GSBCM). This machine incorporates a storage optimization which permits the implementation to reclaim memory via a standard garbage-collection algorithm.

4. a proof that the operational semantics of SBC programs in terms of the GSBCM is equivalent to the operational semantics of SBC programs in terms of the MSBCM.

5. the syntax of VLISP Finite Stored Byte Code (FSBC) (defined as a subset of SBC), and a function that checks whether a SBC program is an FSBC program.

6. an operational semantics of FSBC in terms of a state machine called the finite stored byte code machine (FSBCM). This machine has states whose components are either finite numbers (with fixed bounds) or fixed-length sequences of such finite numbers. This permits the implementation to be realized on a physical machine with finite storage capacity.

7. a proof that the operational semantics of FSBC as given by the FSBCM is faithful to the operational semantics of FSBC as given by the GSBCM. That is, if the FSBCM operational semantics assigns a meaning to an FSBC program, then the GSBCM operational semantics assigns the same meaning to the program.

8. a map from FSBC programs to binary images.

1.1 Notation

We will use the notation contained in Table 1. Additional explanation of our notation can be found in the VLISP report [1].

$\langle \ldots \rangle$	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 $s(\#s - 1) = x$
$s \frown t$	concatenation of sequences s and t
$\Sigma_{i=0}^{n-1} \alpha_i$	concatenation of sequences α_i for $0 \leq i < n$
$s \dagger k$	drop the first k members of sequence s
$s \ddagger k$	the sequence of only the first k members of s
$\rho[i \mapsto x]$	the function which is the same as ρ except that it takes the value x at i

Figure 1: Some Notation

2 Microcoded SBC State Machine

The VLISP report [5] defined the syntax and operational semantics of stored byte code (SBC), and justified the VLISP translation from linked byte code to SBC. The operational semantics of SBC was presented in terms of a state machine called the stored byte code machine (SBCM). In this chapter, we describe the operational semantics of SBC in terms of a new state machine called the microcoded stored byte code machine (MSBCM), and we show that this semantics is equivalent to the semantics presented in terms of the SBCM. The MSBCM refines the SBCM in two ways:

1. The MSBCM alters the representation of stores and stacks to permit a more efficient implementation of some of the action rules; in particular, the action rules `make-env` and `make-cont` can be implemented without copying the argument stack. It is possible to modify this algorithm to permit continuations to be created and restored in a stack-like fashion. This optimization can be added and verified at this level and, based on preliminary benchmarks, would speed up the interpreter substantially.

2. The MSBCM includes four temporary registers in its states; this permits the MSBCM action rules to be expressed as the composition of a small set of primitive functions. In subsequent chapters, we shall refine these primitive functions, thus obtaining an implementation of an SBC interpreter. The actual implementation is written in a subset of Scheme called PreScheme and it is compiled by a verified PreScheme compiler [4, 3].

2.1 SBC Syntax

We reproduce the syntax of the SBC from the VLISP Image Builder report [5].

2.1.1 Abstract Syntax

We express the syntax of SBC in Backus-Naur form (BNF). The terminal symbols of SBC are:

primitives: natural numbers (i.e., 0,1,2,...), booleans (i.e., #t and #f), and identifiers (i.e., alphanumeric symbols that begin with an alphabetic letter);

constructors: HEADER, PTR, FIXNUM, IMM, FALSE, TRUE, CHAR, NULL, UNDEFINED, PAIR, SYMBOL, STRING, VECTOR, LOCATION, TEMPLATE, CODEVECTOR;

keywords: `empty`, `call`, `return`, `make-cont`, `literal`, `closure`, `global`, `local`, `set-global!`, `set-local!`, `push`, `make-env`, `make-restlist`, `unspecified`, `jump`, `jump-if-false`, `check-args=`, `check-args>=`

symbols: "<", ">"

The nonterminal symbols are as follows:

nat	for *natural numbers*,
bool	for *booleans*,
ident	for *identifiers*,
program	for (SBC) *programs*,
term	for (SBC) *terms*,
store	for (SBC) *stores*,
cell	for (SBC) *cells*,
desc	for (SBC) *descriptors*,
vdesc	for (SBC) *value descriptors*,
imm	for (SBC) *immediates*,
htag	for (SBC) *header tags*,
bhtag	for (SBC) *byte header tags*,
dhtag	for (SBC) *descriptor header tags*,
byte	for (SBC) *bytes*,
loc	for (SBC) *locations*, and
obj	for (SBC) *stored object data*

We use extended BNF with productions having the form $N ::= S_1 \mid \ldots \mid S_n$ where N is a nonterminal, and the S_i's are possible forms for phrases in the set denoted by N. We use a variant of the conventional notation for repetition. We write S^k (where k is a nonnegative integer) to stand for the sequence $\langle SS \ldots S \rangle$ containing k occurrences of S. We write S^* to stand for the sequence $\langle SS \ldots \rangle$ containing a finite number of occurrences of S (including, possibly, no occurrence of S).

We assume *bpw* to be some constant natural number; in the implementation, *bpw* will be either 1 or 4 (due to the architecture of the physical machines we shall be working with).

The abstract syntax of Stored Byte Code (SBC) is as follows:

$$
\begin{array}{rcl}
\mathit{program} & ::= & \mathit{term} \\
\mathit{term} & ::= & \langle \mathit{store}\ \mathit{vdesc} \rangle \\
\mathit{store} & ::= & \mathit{cell}^* \\
\mathit{cell} & ::= & \mathit{desc} \mid \mathit{byte}^{\mathit{bpw}} \\
\mathit{desc} & ::= & \langle \text{HEADER}\ \mathit{htag}\ \mathit{bool}\ \mathit{nat} \rangle \mid \mathit{vdesc} \\
\mathit{vdesc} & ::= & \langle \text{PTR}\ \mathit{loc} \rangle \mid \langle \text{FIXNUM}\ \mathit{nat} \rangle \mid \langle \text{IMMEDIATE}\ \mathit{imm} \rangle \\
\mathit{imm} & ::= & \text{FALSE} \mid \text{TRUE} \mid \langle \text{CHAR}\ \mathit{nat} \rangle \mid \text{NULL} \mid \text{UNDEFINED} \\
\mathit{htag} & ::= & \mathit{bhtag} \mid \mathit{dhtag} \\
\mathit{bhtag} & ::= & \text{STRING} \mid \text{CODEVECTOR} \\
\mathit{dhtag} & ::= & \text{PAIR} \mid \text{SYMBOL} \mid \text{VECTOR} \mid \text{LOCATION} \mid \text{TEMPLATE} \\
\mathit{byte} & ::= & \mathit{nat} \mid \texttt{empty} \mid \texttt{call} \mid \texttt{return} \mid \texttt{make-cont} \mid \texttt{literal} \mid \\
 & & \texttt{closure} \mid \texttt{global} \mid \texttt{local} \mid \texttt{set-global!} \mid \texttt{set-local!} \mid \\
 & & \texttt{push} \mid \texttt{make-env} \mid \texttt{make-restlist} \mid \texttt{unspecified} \mid \texttt{jump} \mid \\
 & & \texttt{jump-if-false} \mid \texttt{check-args=} \mid \texttt{check-args>=} \mid \mathit{ident} \\
\mathit{loc} & ::= & \mathit{nat} \\
\mathit{obj} & ::= & \mathit{vdesc} \mid \mathit{byte}
\end{array}
$$

We will use i and m-like variables for natural numbers, p-like variables for booleans, and r-like variables for identifiers. Similarly for the other nonterminals, with

P	for *(SBC) programs*,
T	for *(SBC) terms*,
s	for *(SBC) stores*,
c	for *(SBC) cells*,
d	for *(SBC) descriptors*,
vd	for *(SBC) value descriptors*,
imm	for *(SBC) immediates*,
h	for *(SBC) header tags*,
bh	for *(SBC) byte header tags*,
dh	for *(SBC) descriptor header tags*,
b	for *(SBC) bytes*,
l	for *(SBC) locations*, and
o	for *(SBC) stored object data*

We will define an SBC *store* in such a way that it can be viewed as a succession of *stored objects*. Each stored object is a sequence of *cells*, with the first cell being a *header* and the remaining cells being *data*. Stored objects are classified into two broad classes: *descriptor objects* and *byte objects*.

The data cells of descriptor objects are *value descriptors*, which are either *constants* (numbers, booleans, characters, etc.) or *pointers* to stored objects. A pointer is restricted to index the first data cell of some stored object. A header cell of a descriptor object has the form $\langle$HEADER *dh p m*$\rangle$ where *dh* (the *(descriptor) header tag*) is a tag that describes the stored object, *p* (the *mutability flag*) is a boolean value that describes whether the object is mutable, and *m* (the *header size*) is a multiple of the number of value descriptors in the stored object, the multiple being the constant *bpw*.

The data cells of byte objects are fixed length sequences of *bytes*, bytes being *numbers*, *keywords*, or *(special) identifiers*. These sequences of bytes can be concatenated to obtain a single sequence of bytes. Only a prefix of this sequence is considered to be useful, and the bytes in this prefix are called *useful bytes*. A header cell of a byte object has the form $\langle$HEADER *bh p m*$\rangle$ where *bh* (the *(byte) header tag*) is a tag that describes the stored object, *p* (the *mutability flag*) is a boolean value that describes whether the object is mutable, and *m* (the *header size*) is the number of useful bytes in the stored object.

We shall identify a stored object by the name of its header tag. Thus we call a stored object with header tag TEMPLATE to be a template. Templates, codevectors, and symbols are immutable and hence their immutability flag is always false ($\#f$). Locations are always mutable, while pairs, strings, and vectors may be either mutable or immutable. Likewise, the header size of symbols is always $1 * bpw$, the header size of locations and pairs is always $2 * bpw$, and the header size of templates, codevectors, strings, and vectors varies.

An SBC *term* is an SBC store together with a value descriptor. If the value descriptor is a constant, then the store is irrelevant; otherwise, the value descriptor will be a pointer to a stored object in the store. An SBC *program* is an SBC term that consists of an SBC store and a pointer to a "template object" in the store.

The header of a stored object contains the object size in units of the number of bytes — we call this the *size in bytes*. The object size in units of cells (i.e., the *size in cells*) is obtained by means of a function $\mho$ that converts a size in bytes to a size in cells. Thus if a stored object's header size is n, then the object consists of $\mho(n)$ data cells. The function Ω converts

a size in cells to a size in bytes. These functions use the constant *bpw* which represents the number of bytes per cell.

Definition 1 *Let* `div` *be the integer division (i.e., quotient) operation. Then* $\mho$ *and* Ω *are functions (from natural numbers to natural numbers) defined as follows:*

$$\mho n \stackrel{\text{def}}{=} (n + bpw - 1)\ \texttt{div}\ bpw$$

$$\Omega n \stackrel{\text{def}}{=} n * bpw$$

2.1.2 Stores

A store s is represented as a sequence of cells. Thus $\#s$ is the domain of s, and is also the least l such that $s(l)$ is not defined. $s[l \mapsto v]$ is the function whose domain is $\#s \cup l$ and which is identical to s at all arguments except possibly l where its value is v. This is a store if and only if $l \leq \#s$. An SBC store is defined to be a store that satisfies the following invariants:

1. If $s(l) = \langle \text{HEADER } h\ p\ m \rangle$ for some l, h, p, m, then for all $1 \leq x \leq \mho m$,
   - If $h \in bhtag$, then $s(l + x) = \langle b_1 \ldots b_{bpw} \rangle$ for some $b_1 \ldots b_{bpw}$,
   - If $h \in dhtag$, then $s(l + x) = vd$ for some vd;
2. For all l such that $s(l)$ is defined,

$$\begin{array}{l} s(l)(1) \neq \text{HEADER} \Rightarrow \\ \quad (\exists l', h, p, m)\ \ (s(l') = \langle \text{HEADER } h\ p\ m \rangle) \wedge \\ \qquad\qquad\qquad (l' < l \leq l' + \mho m) \end{array}$$

3. If $s(l) = \langle \text{PTR } l' \rangle$ for some l, l', then $s(l' - 1) = \langle \text{HEADER } h\ p\ m \rangle$ for some h, p, m.

These invariants ensure that the store is a succession of stored objects. Thus every header cell in the store is the header of a stored object and should be followed by the appropriate number and kind of data cells (invariant 1), every non-header cell in the store is a data cell of some stored object (invariant 2), and every pointer cell is a pointer to the first data cell of a stored object (invariant 3).

2.1.3 Programs

An SBC program is defined to be a term $\langle s\ vd \rangle$ such that

1. s is an SBC store;
2. all location objects have UNDEFINED values. That is, if $s(l-1) = \langle$HEADER LOCATION $\#\ 2{*}bpw\rangle$, then $s(l) = \langle$IMMEDIATE UNDEFINED$\rangle$; and
3. vd is a pointer to a template object in the store s, i.e., $vd = \langle$PTR $l\rangle$ and $s(l-1) = \langle$HEADER TEMPLATE $\#f\ m\rangle$ for some l, m. vd represents the root template of the program.

2.2 MSBC State Machines (MSBCM)

The operational semantics of stored byte code is defined in terms of a deterministic state machine with concrete states. The state machine is called the *Microcoded Stored Byte Code Machine (*MSBCM*)*. We define the syntax of MSBC states by augmenting the syntax of the Stored Byte Code. We then define the states and action rules of a deterministic state machine. We adopt the notation and conventions presented in the VLISP Image Builder report [5].

2.2.1 Microcoded Stored Byte Code (MSBC)

The Microcoded Stored Byte Code (MSBC) provides the syntactic objects that form the states of the MSBCM. It expands the SBC to provide representations of objects such as environments, closures, continuations, etc. These objects are not part of SBC as they do not occur in the code obtained by compiling Scheme programs; they are put in the initial state of the MSBCM and are also generated during the course of computation.

The new MSBC tokens are as follows:

constructors: UNSPECIFIED, EMPTY-ENV, HALT, EOF, CLOSURE, PORT, CONTINUATION, ENVIRONMENT

The new MSBC syntactic categories are:

t for (MSBC) *templates*,
n for (MSBC) *offsets*,
c for (MSBC) *codevectors*,
v for (MSBC) *values*,
u for (MSBC) *environments*,
k for (MSBC) *continuations*,
a for (MSBC) *argument stacks*

The abstract syntax of Microcoded Stored Byte Code (MSBC) is defined below. This grammar redefines several syntactic categories of SBC.

$$
\begin{array}{rcl}
\mathit{program} & ::= & \mathit{term} \\
\mathit{term} & ::= & \langle \mathit{store}\ \mathit{vdesc} \rangle \\
\mathit{store} & ::= & \langle \mathit{storeseg}\ \mathit{storeseg}\ \mathit{storeseg}\ \mathit{storeseg} \rangle \\
\mathit{storeseg} & ::= & \mathit{cell}^* \\
\mathit{cell} & ::= & \mathit{desc} \mid \mathit{byte}^{\mathit{bpw}} \\
\mathit{desc} & ::= & \langle \textsc{header}\ \mathit{htag}\ \mathit{bool}\ \mathit{nat} \rangle \mid \mathit{vdesc} \\
\mathit{vdesc} & ::= & \langle \textsc{ptr}\ \mathit{loc} \rangle \mid \langle \textsc{fixnum}\ \mathit{nat} \rangle \mid \langle \textsc{immediate}\ \mathit{imm} \rangle \\
\mathit{imm} & ::= & \textsc{false} \mid \textsc{true} \mid \langle \textsc{char}\ \mathit{nat} \rangle \mid \textsc{null} \mid \textsc{undefined} \\
 & & \textsc{unspecified} \mid \textsc{empty-env} \mid \textsc{halt} \mid \textsc{eof} \\
\mathit{htag} & ::= & \mathit{bhtag} \mid \mathit{dhtag} \\
\mathit{bhtag} & ::= & \textsc{string} \mid \textsc{codevector} \\
\mathit{dhtag} & ::= & \textsc{pair} \mid \textsc{symbol} \mid \textsc{vector} \mid \textsc{location} \mid \textsc{template} \mid \\
 & & \textsc{closure} \mid \textsc{port} \mid \textsc{continuation} \mid \textsc{environment} \\
\mathit{byte} & ::= & \mathit{nat} \mid \texttt{empty} \mid \texttt{call} \mid \texttt{return} \mid \texttt{make-cont} \mid \texttt{literal} \mid \\
 & & \texttt{closure} \mid \texttt{global} \mid \texttt{local} \mid \texttt{set-global!} \mid \texttt{set-local!} \mid \\
 & & \texttt{push} \mid \texttt{make-env} \mid \texttt{make-restlist} \mid \texttt{unspecified} \mid \texttt{jump} \mid \\
 & & \texttt{jump-if-false} \mid \texttt{check-args=} \mid \texttt{check-args>=} \mid \mathit{ident} \\
\mathit{loc} & ::= & \langle \mathit{nat}\ \mathit{nat} \rangle \\
\mathit{obj} & ::= & \mathit{vdesc} \mid \mathit{byte} \\
\mathit{stack} & ::= & \mathit{vdesc}^*
\end{array}
$$

We extend the naming conventions of SBC for variables. Thus, we will use m-like variables for numbers, p-like variables for booleans, and r-like variables for identifiers. Similarly for the other nonterminals, with

P	for *(*MSBC*) programs*,
T	for *(*MSBC*) terms*,
s	for *(*MSBC*) stores*,
s_i	for *(*MSBC*) store segments* with $i < 4$,
cell	for *(*MSBC*) cells*,
d	for *(*MSBC*) descriptors*,
vd	for *(*MSBC*) value descriptors*,
imm	for *(*MSBC*) immediates*,
h	for *(*MSBC*) header tags*,
bh	for *(*MSBC*) byte header tags*,
dh	for *(*MSBC*) descriptor header tags*,
b	for *(*MSBC*) bytes*,
l	for *(*MSBC*) locations*,
o	for *(*MSBC*) stored object data*, and
a	for *(*MSBC*) stacks*

In addition, we will use t, c, v, u, k, and r-like variables for *(*MSBC*) value descriptors*, and i,m,n-like variables for numbers.

Stores

A store s is represented as a sequence of four store segments, where each store segment is a sequence of cells. By abuse of notation, we shall write $s(\langle i, m\rangle)$ to mean $s(i)(m)$. The first and third store segments are indexed by the nonnegative integers, while the second and fourth are indexed by the nonpositive integers.

Let $0 \leq i < 4$ and let $s_i = s(i)$. Then, for $i \in \{0, 2\}$, $\#s_i$ is the domain of s_i, and is also the least nonnegative m such that $s_i(m)$ is not defined. For $i \in \{1, 3\}$, $-\#s_i$ is the domain of s_i, and is also the greatest nonpositive m such that $s_i(m)$ is not defined. $s_i[m \mapsto v]$ is the function whose domain is $\#s_i \cup m$ and which is identical to s_i at all arguments except possibly m where its value is v. This is a store if and only if $i \in \{0, 2\}$ and $m \leq \#s_i$ or $i \in \{1, 3\}$ and $m \geq -\#s_i$. We shall write $s[\langle i, m\rangle \mapsto v]$ to mean $s(i)[m \mapsto v]$.

A location l is a pair of numbers $\langle i, m\rangle$ such that

- $m \geq 0$ if $i \in \{0, 2\}$
- $m \leq 0$ if $i \in \{1, 3\}$

Let $l = \langle i, m\rangle$ and $l' = \langle i', m'\rangle$. By the above notation, $s(l) = s(i)(m)$ and

$s(l') = s(i')(m')$. We shall also use the following notation:

$$\begin{array}{lcl} l +_A n & \stackrel{\text{def}}{=} & \langle i, (m+n)\rangle \\ l -_A n & \stackrel{\text{def}}{=} & \langle i, (m-n)\rangle \\ l <_A l' & \stackrel{\text{def}}{=} & (i = i') \wedge (m < m') \\ l \leq_A l' & \stackrel{\text{def}}{=} & (i = i') \wedge (m \leq m') \end{array}$$

An MSBC store is defined to be a store that satisfies the following invariants:

1. If $s(l) = \langle \text{HEADER } h\ p\ m\rangle$ for some l, h, p, m, then for all $1 \leq x \leq \mho m$,
   - If $h \in bhtag$, then $s(l +_A x) = \langle b_1 \ldots b_{bpw}\rangle$ for some $b_1 \ldots b_{bpw}$.
   - If $h \in dhtag$, then $s(l +_A x) = vd$ for some vd.
2. For all l such that $s(l)$ is defined,

$$\begin{array}{l} s(l)(1) \neq \text{HEADER} \Rightarrow \\ \quad (\exists l', h, p, m)\ \ (s(l') = \langle \text{HEADER } h\ p\ m\rangle) \wedge \\ \qquad\qquad\qquad\ \ (l' <_A l \leq_A l' +_A \mho m) \end{array}$$

3. If $s(l) = \langle \text{PTR } l'\rangle$ for some l, l', then $s(l' -_A 1) = \langle \text{HEADER } h\ p\ m\rangle$ for some h, p, m.

These invariants ensure that each store segment is a succession of stored objects. Thus every header cell in the store is the header of a stored object and should be followed by the appropriate number and kind of data cells (invariant 1), every non-header cell in the store is a data cell of some stored object (invariant 2), and every pointer cell is a pointer to the first data cell of a stored object (invariant 3).

Stored Objects

An MSBC store consists of a succession of *stored objects*. Each stored object is a sequence of cells, with the first cell being a *header* and the remaining cells being *data*. A stored object is always identified by the location of its first data cell. If l is a location such that $s(l)$ is the first data cell of a stored object, then $\langle \text{PTR } l\rangle$ is called a *pointer* to the stored object in store s.

Let d be a pointer into store s. Then $\mathcal{S}(d, s)$ is defined to be a sequence of the contents of the stored object pointed to by d in s. The first element of this sequence is the header tag, the second element is the mutability flag, and the remaining elements are the data elements of the object. Note that

the sequences of bytes within a byte object are flattened into a single byte sequence; this resulting sequence may be larger than the object size, so the extra bytes are truncated from the end of the sequence. The result is the sequence of "useful" bytes in the stored object.

Definition 2 *For all* d, s, h, p, m, o^*, l *such that* $s(l +_A (\mho m - 1))$ *is defined,* $\mathcal{S}(d, s) \stackrel{\text{def}}{=} \langle h\ p\ o^* \rangle$ *if*

- $d = \langle \text{PTR}\ l \rangle$
- $s(l -_A 1) = \langle \text{HEADER}\ h\ p\ m \rangle$
- *if* $h \in dhtag$ *then*
  $o^* = \langle s(l) \ldots s(l +_A (\mho m - 1)) \rangle$
  else
  $o^* = (s(l) \frown \ldots \frown s(l +_A (\mho m - 1))) \ddagger m$

The following notation provides a concise way of appending stored objects to stores.

Definition 3 *Let* $s = \langle s_0\ s_1\ s_2\ s_3 \rangle$ *be an* MSBC *store. If* $i \in \{0, 2\}$ *then*

$$
\begin{array}{lll}
s +_i \langle h\ p\ o^* \rangle & \stackrel{\text{def}}{=} & \textit{if } h \in dhtag \textit{ then} \\
& & \quad s[i \mapsto s_i \frown \langle \langle \text{HEADER}\ h\ p\ (\Omega \# o^*) \rangle \rangle \frown o^*] \\
& & \textit{else} \\
& & \quad s[i \mapsto s_i \frown \langle \langle \text{HEADER}\ h\ p\ \# o^* \rangle \rangle \frown o'^*] \\
& & \quad \textit{where } \# o'^* = \mho \# o^*, \\
& & \quad \textit{and } (\forall 0 \leq i < \# o'^*)\ \# o'^*(i) = bpw \\
& & \quad \textit{and } o^* = (\Sigma_{i=0}^{\# o'^* - 1} o'^*(i)) \ddagger \# o^*
\end{array}
$$

If $i \in \{1, 3\}$ *then*

$$
\begin{array}{lll}
s +_i \langle h\ p\ o^* \rangle & \stackrel{\text{def}}{=} & \textit{if } h \in dhtag \textit{ then} \\
& & \quad s[i \mapsto \langle \langle \text{HEADER}\ h\ p\ (\Omega \# o^*) \rangle \rangle \frown o^* \frown s_i] \\
& & \textit{else} \\
& & \quad s[i \mapsto \langle \langle \text{HEADER}\ h\ p\ \# o^* \rangle \rangle \frown o'^* \frown s_i] \\
& & \quad \textit{where } \# o'^* = \mho \# o^*, \\
& & \quad \textit{and } (\forall 0 \leq i < \# o'^*)\ \# o'^*(i) = bpw \\
& & \quad \textit{and } o^* = (\Sigma_{i=0}^{\# o'^* - 1} o'^*(i)) \ddagger \# o^*
\end{array}
$$

The following lemma asserts that appending a stored object to the ith segment of a store s (for $i \in \{0, 2\}$) results in a new store s' such that $\langle \text{PTR}\ \langle i\ (\# s_i + 1) \rangle \rangle$ is a pointer to the stored object in the new store s'.

Lemma 4 $\mathcal{S}(\langle \text{PTR } \langle i\ (\#s_i + 1)\rangle\rangle, s +_i \langle h\ p\ o^*\rangle) = \langle h\ p\ o^*\rangle$

Proof: Immediate from definitions.

□

The following lemma asserts that appending a stored object to the ith segment of a store s (for $i \in \{1, 3\}$) results in a new store s' such that $\langle \text{PTR } \langle i\ (-\#s_i - \#o^* + 1)\rangle\rangle$ is a pointer to the stored object in the new store s'.

Lemma 5 $\mathcal{S}(\langle \text{PTR } \langle i\ (-\#s_i - \#o^* + 1)\rangle\rangle, s +_i \langle h\ p\ o^*\rangle) = \langle h\ p\ o^*\rangle$

Proof: Immediate from definitions.

□

The following notation provides a concise way of creating a new stored object and appending it to a store. Let $i \in \{0, 2\}$ and let s be an MSBC store. Then,

$$s \oplus_i \langle h\ p\ m\rangle \quad = \quad \begin{array}{ll} s +_i \langle h\ p\ d_u^*\rangle & \text{if } h \in dhtag \text{ and } \#d_u^* = m \\ s +_i \langle h\ p\ b^*\rangle & \text{if } h \in bhtag \text{ and } \#b^* = m \end{array}$$

2.2.2 Stored Object Predicates

We define a family of predicates that characterize the types of stored objects in an MSBC store. These relations are defined recursively.

Let d, t, c, v, u, k range over value descriptors. The free variables in each formula are (implicitly) existentially quantified over the entire formula. Thus, for example,

$$\text{String}(d, s) \stackrel{\text{def}}{=} \mathcal{S}(d, s) = \langle \text{STRING } p\ b^*\rangle$$

means

$$\text{String}(d, s) \stackrel{\text{def}}{=} (\exists p, b^*)\mathcal{S}(d, s) = \langle \text{STRING } p\ b^*\rangle.$$

$$
\begin{array}{lcl}
\text{Pair}(d,s) & \iff & \mathcal{S}(d,s) = \langle \textsc{pair}\ p\ \langle o_1\ o_2\rangle\rangle \\
 & & \quad \text{and Value}(o_1,s) \text{ and Value}(o_2,s) \\
\text{Symbol}(d,s) & \iff & \mathcal{S}(d,s) = \langle \textsc{symbol}\ \#f\ \langle o\rangle\rangle \text{ and String}(o,s) \\
\text{String}(d,s) & \iff & \mathcal{S}(d,s) = \langle \textsc{string}\ p\ b^*\rangle \\
\text{Vector}(d,s) & \iff & \mathcal{S}(d,s) = \langle \textsc{vector}\ p\ o^*\rangle \\
 & & \quad \text{and } (\forall 0 \le i < \#o^*)\ \text{Value}(o^*(i),s) \\
\text{Location}(d,s) & \iff & \mathcal{S}(d,s) = \langle \textsc{location}\ \#t\ \langle o_1\ o_2\rangle\rangle \\
 & & \quad \text{and Value}(o_1,s) \text{ and Symbol}(o_2,s) \\
\text{Template}(t,s) & \iff & \mathcal{S}(t,s) = \langle \textsc{template}\ \#f\ c{::}o^*\rangle \\
 & & \quad \text{and Codevector}(c,s) \\
 & & \quad \text{and } (\forall 0 \le i < \#o^*)\ \text{Value}(o^*(i),s) \\
 & & \\
\text{Codevector}(c,s) & \iff & \mathcal{S}(c,s) = \langle \textsc{codevector}\ \#f\ b^*\rangle \\
\text{Offset}(n,t,s) & \iff & \text{Template}(t,s) \\
 & & \quad \text{and } \mathcal{S}(t,s) = \langle \textsc{template}\ \#f\ c{::}o^*\rangle \\
 & & \quad \text{and } \mathcal{S}(c,s) = \langle \textsc{codevector}\ \#f\ b^*\rangle \\
 & & \quad \text{and } 0 \le n < \#b^* \\
\text{Closure}(d,s) & \iff & \mathcal{S}(d,s) = \langle \textsc{closure}\ \#f\ \langle t\ u\rangle\rangle \\
 & & \quad \text{and Template}(t,s) \\
 & & \quad \text{and Environment}(u,s) \\
\text{Continuation}(k,s) & \iff & k = \langle \textsc{immediate halt}\rangle \text{ or} \\
 & & \mathcal{S}(k,s) = \langle \textsc{continuation}\ \#f\ \langle t\ n\ u\ k\rangle \frown vd^*\rangle \\
 & & \quad \text{and Template}(t,s) \text{ and Offset}(n,t,s) \\
 & & \quad \text{and Environment}(u,s) \\
 & & \quad \text{and Continuation}(k,s) \\
 & & \quad \text{and } (\forall 0 \le i < \#vd^*)\ \text{Value}(vd^*(i),s) \\
\text{Environment}(u,s) & \iff & u = \langle \textsc{immediate empty-env}\rangle \text{ or} \\
 & & \mathcal{S}(u,s) = \langle \textsc{environment}\ \#t\ u'{::}o^*\rangle \\
 & & \quad \text{and Environment}(u',s) \\
 & & \quad \text{and } (\forall 0 \le i < \#o^*)\ \text{Value}(o^*(i),s) \\
\text{Value}(v,s) & \iff & v = \langle \textsc{fixnum}\ n\rangle \text{ or } v = \langle \textsc{immediate}\ imm\rangle \\
 & & \quad \text{or } \mathcal{S}(v,s) \text{ is defined}
\end{array}
$$

2.2.3 States

The states of a microcoded SBC state machine (MSBCM) are the sequences of the form

$$\langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4\rangle$$

that satisfy the following (MSBCM) state invariants:

1. s is an MSBC store
2. Template(t, s)
3. Codevector(c, s)
4. $\mathcal{S}(t, s) = \langle$TEMPLATE $\#f\ c{::}o^*\rangle$ for some o^*.
5. Offset(n, t, s)
6. Value(v, s)
7. Environment(u, s)
8. Continuation(k, s)
9. $(\forall 0 \leq i < \#a)$ Value$(a(i), s)$

The components of a state are called, in order, its *template*, *offset*, *codevector*, *value*, *argument stack*, *environment*, *continuation*, *store*, *spare1*, *spare2*, *spare3*, and *spare4*, and we may informally speak of them as being held in registers. The first three state invariants ensure that the store is an MSBC store. The remaining invariants restrict the values of the register components of states. For example, the register t is restricted to hold pointers to templates (i.e., stored objects with header tag TEMPLATE).

Note that the MSBCM state invariants are the same as the SBCM state invariants.

2.2.4 Initial and Halt States

The initial states of the MSBCM are the states $\langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4\rangle$ such that

- $n = 0$,
- $v = \langle$IMMEDIATE UNSPECIFIED$\rangle$,
- $a = \langle\rangle$,
- $u = \langle$IMMEDIATE EMPTY-ENV$\rangle$,
- $k = \langle$IMMEDIATE HALT$\rangle$, and

- $s(1) = s(2) = s(3) = \langle\rangle$.

The halt states of the MSBCM are the states $\langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4\rangle$ such that for some b^*,

- $\mathcal{S}(c, s) = \langle$CODEVECTOR $\#f\ b^*\rangle$,
- $b^*(n) =$ `return`, and
- $k = \langle$IMMEDIATE HALT$\rangle$.

2.2.5 State Observers and Mutators

We shall implement the action rules of the MSBCM as the composition of functions drawn from a small set of functions (called *microcode* functions). Microcode functions are classified into two categories. Microcode functions that map MSBCM states to non-state values are called *(*MSBCM*) observers* since they only observe the state and do not mutate it. Microcode functions that map MSBCM states to MSBCM states are called *(*MSBCM*) mutators*. MSBCM observers are specified in Figure 2 and MSBCM mutators are specified in Figures 3 and 4.

2.2.6 Stored Object Manipulators

In Figures 2, 3, and 4 we specified a collection of microcode functions. These functions were classified into register, stack, stored object, and miscellaneous functions. In this section, we specify a fairly large collection of auxiliary functions that will be used to manipulate stored objects in an MSBCM store. These auxiliary functions (called *stored object manipulators*) are implemented in terms of the microcode functions; their primary purpose is to abstract away from representation details of how stored objects are represented in the store. The MSBCM action rules will be implemented in terms of the register and stack microcode functions, the stored object manipulators, and other miscellaneous auxiliary functions. As with the microcode functions, we distinguish between stored object manipulators on the basis of their being *observers* or *mutators*.

For each *stored object observer* we give names to the function and its arguments, one or more conditions determining when the function is defined (and possibly introducing new locally bound variables for later use), a specification of the value returned by the function, and an "implementation" of the function in terms of microcode observers and other auxiliary functions.

Let $\Sigma = \langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4 \rangle$ be an MSBCM state. Then,

Register Observers

`template-ref` Σ	$=$	t
`codevector-ref` Σ	$=$	c
`offset-ref` Σ	$=$	n
`value-ref` Σ	$=$	v
`env-ref` Σ	$=$	u
`cont-ref` Σ	$=$	k
`spare1-ref` Σ	$=$	r_1
`spare2-ref` Σ	$=$	r_2
`spare3-ref` Σ	$=$	r_3
`spare4-ref` Σ	$=$	r_4

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 $s(l -_A 1) = \langle h\ p\ m \rangle$. Let $\mathcal{S}(d, s) = \langle h\ p\ o^* \rangle$. Then

`stob-desc-ref` $d\ i\ \Sigma$	$=$	$o^*(i)$ if $h \in dhtag$
`stob-byte-ref` $d\ i\ \Sigma$	$=$	$o^*(i)$ if $h \in bhtag$
`stob-tag` $d\ \Sigma$	$=$	h
`stob-mutable?` $d\ \Sigma$	$=$	p
`stob-size-in-bytes` $d\ \Sigma$	$=$	m
`stob-size-in-cells` $d\ \Sigma$	$=$	$\mho m$

Figure 2: MSBCM State Observers.

Let $\Sigma = \langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4 \rangle$ be an MSBCM state. Then,

Register Mutators

`template-set` $vd\ \Sigma$	$=$	$\Sigma[t' = vd]$	if Template(vd)
`codevector-set` $vd\ \Sigma$	$=$	$\Sigma[c' = vd]$	if Codevector(vd)
`offset-set` $i\ \Sigma$	$=$	$\Sigma[n' = i]$	if Offset(i, t, s)
`offset-inc` $i\ \Sigma$	$=$	$\Sigma[n' = n + i]$	if Offset$(n + i, t, s)$
`value-set` $vd\ \Sigma$	$=$	$\Sigma[v' = vd]$	if Value(vd)
`env-set` $vd\ \Sigma$	$=$	$\Sigma[u' = vd]$	if Environment(vd)
`cont-set` $vd\ \Sigma$	$=$	$\Sigma[k' = vd]$	if Continuation(vd)
`spare1-set` $vd\ \Sigma$	$=$	$\Sigma[r_1' = vd]$	
`spare2-set` $vd\ \Sigma$	$=$	$\Sigma[r_2' = vd]$	
`spare3-set` $i\ \Sigma$	$=$	$\Sigma[r_3' = i]$	
`spare4-set` $d\ \Sigma$	$=$	$\Sigma[r_4' = d]$	

Stack Mutators

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

Miscellaneous Mutators

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

Figure 3: MSBCM State Mutators.

Stored Object Mutators

$$
\begin{array}{lcl}
\texttt{make-desc-stob}\ h\ p\ m\ \Sigma & = & \Sigma[s' = s \oplus_0 \langle h\ p\ m\rangle] \\
 & & \quad [r'_1 = \langle 0, \#s(0)\rangle] \\
 & & \text{if } h \in \mathit{dhtag} \\
\texttt{make-desc-stob-from-stack}\ h\ p\ \Sigma & & \\
 & = & \Sigma[s' = s +_1 \langle h\ p\ a\rangle] \\
 & & \quad [a' = \langle\rangle] \\
 & & \quad [r'_1 = \langle 1, -\#s(1) - \#a + 1\rangle] \\
 & & \text{if } h \in \mathit{dhtag} \\
\texttt{make-byte-stob}\ h\ p\ m\ \Sigma & = & \Sigma[s' = s \oplus_0 \langle h\ p\ m\rangle] \\
 & & \quad [r'_1 = \langle 0, \#s(0)\rangle] \\
 & & \text{if } h \in \mathit{bhtag}
\end{array}
$$

Let $d = \langle \textsc{ptr}\ l\rangle$ and $\mathcal{S}(d, s) = \langle h\ p\ o^*\rangle$. Then

$$
\begin{array}{lcl}
\texttt{stob-desc-set}\ d\ i\ vd\ \Sigma & = & \Sigma[s' = s[(l +_A i) \mapsto vd]] \\
 & & \quad \text{if } h \in \mathit{dhtag} \\
 & & \quad \text{and } 0 \le i < \#o^* \\
\texttt{stob-byte-set}\ d\ i\ b\ \Sigma & = & \Sigma[s' = s[(l +_A i') \mapsto b^*]] \\
 & & \quad \text{if } h \in \mathit{bhtag} \\
 & & \quad \text{and } 0 \le i < \#o^* \\
 & & \quad \text{and } i' = \mho(i+1) - 1 \\
 & & \quad \text{and } b^* = s(l +_A i')[(i \bmod \mathit{bpw}) \mapsto b]
\end{array}
$$

Figure 4: MSBCM State Mutators (continued).

For each *stored object mutator* we give names to the function and its arguments, one or more conditions determining when the function is defined (and possibly introducing new locally bound variables for later use), a specification of the new values of some registers, and an "implementation" of the function in terms of microcode mutators and other auxiliary functions.

Often the domain of a stored object manipulator is specified by equations giving "the form" of certain registers or values. In all specifications, the original values of the various registers are designated by conventional variables used exactly as in the definition of a state: t, n, c, v, a, u, k, s, r_1, r_2, r_3, and r_4. Call these the original register variables. In specifications of stored object mutators, the new values of the registers are indicated by the same variables with primes attached: t', n', c', v', a', u', k', s', r_1', r_2', r_3', and r_4'. Call these the new register variables. New register variables occur only as the left hand sides of equations specifying 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 function arguments, and may introduce new variables (not among the function arguments or 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 $j + 12$ places

(†) $R(t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4, x_1, \ldots, x_j)$.

The domain condition really is this: the rule can be applied in a given state if there exist $x_1, \ldots, x_j$ such that (†). Furthermore, in the value and 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.

The "implementation" of the action rules expresses each rule as the composition of stored object manipulators, microcode functions, and other auxiliary functions. The implementation is expressed as a Scheme program. Note that Scheme is a call-by-value language that implicitly threads the global state through function calls; hence the global state does not appear as an explicit parameter.

Function: make-closure-stob!
Domain conditions:

Changes:

$s' = s \oplus_0 \langle \text{CLOSURE } \#f\ 2 \rangle$

Implementation:

```
(make-desc-stob! hdr/closure #f 2)
```

Function: closure-stob? d
Domain conditions:

Return value:

$\#t$ if $d = \langle \text{PTR } l \rangle$ and $\mathcal{S}(d, s) = \langle \text{CLOSURE } \#f\ \langle d_0\ d_1 \rangle \rangle$
$\#f$ otherwise

Implementation:

```
(and (pointer? d) (= (stob-type d) hdr/closure))
```

Function: closure-stob-template d
Domain conditions:

$\mathcal{S}(d, s) = \langle \text{CLOSURE } \#f\ \langle d_0\ d_1 \rangle \rangle$

Return value:

d_0

Implementation:

```
(stob-desc-ref d 0)
```

Function: closure-stob-env d
Domain conditions:

$\mathcal{S}(d, s) = \langle \text{CLOSURE } \#f\ \langle d_0\ d_1 \rangle \rangle$

Return value:

d_1

Implementation:

`(stob-desc-ref d 1)`

Function: closure-stob-template-init! d *val*
Domain conditions:
$d = \langle \text{PTR } l \rangle$
$\mathcal{S}(d, s) = \langle \text{CLOSURE } \#f \ \langle d_0 \ d_1 \rangle \rangle$
Changes:
$s' = s[l \mapsto val]$

Implementation:

`(stob-desc-set! d 0` *val*`)`

Function: closure-stob-env-init! d *val*
Domain conditions:
$d = \langle \text{PTR } l \rangle$
$\mathcal{S}(d, s) = \langle \text{CLOSURE } \#f \ \langle d_0 \ d_1 \rangle \rangle$
Changes:
$s' = s[l +_A 1 \mapsto val]$

Implementation:

`(stob-desc-set! d 1` *val*`)`

Function: make-codevector-stob! m
Domain conditions:

Changes:
$s' = s \oplus_0 \langle \text{CODEVECTOR } \#f \ m \rangle$

Implementation:

`(make-desc-stob! hdr/codevector #f m)`

Function: codevector-stob? d
Domain conditions:

Return value:

$\#t$ if $d = \langle \text{PTR } l \rangle$ and $\mathcal{S}(d, s) = \langle \text{CODEVECTOR } \#f \ b^* \rangle$

$\#f$ otherwise

Implementation:

```
(and (pointer? d) (= (stob-type d) hdr/codevector))
```

Function: codevector-stob-length d

Domain conditions:

$\mathcal{S}(d, s) = \langle \text{CODEVECTOR } \#f \ b^* \rangle$

Return value:

$\#b^*$

Implementation:

```
(stob-size-in-bins d)
```

Function: codevector-stob-ref d i

Domain conditions:

$\mathcal{S}(d, s) = \langle \text{CODEVECTOR } \#f \ b^* \rangle$

$0 \leq i < \#b^*$

Return value:

$b^*(i)$

Implementation:

```
(stob-binarray-ref d i)
```

Function: codevector-stob-init! d i b

Domain conditions:

$d = \langle \text{PTR } l \rangle$

$\mathcal{S}(d, s) = \langle \text{CODEVECTOR } \#f \ b^* \rangle$

$0 \leq i < \#b^*$

$i' = \mho(i + 1) - 1$

$b'^* = s(l +_A i')[(i \bmod bpw) \mapsto b]$

Changes:

$$s' = s[(l +_A i') \mapsto b'^*]$$

Implementation:

```
(stob-binarray-set! d i b)
```

Function: make-continuation-stob!
Domain conditions:

Changes:

$$s' = s +_1 \langle \text{CONTINUATION } \#f\ a \rangle$$

Implementation:

```
(make-desc-stob-from-stack! hdr/continuation #f)
```

Function: continuation-stob? d
Domain conditions:

Return value:

$\#t$	if $d = \langle \text{IMMEDIATE HALT} \rangle$
$\#t$	if $d = \langle \text{PTR } l \rangle$
	and $\mathcal{S}(d, s) = \langle \text{CONTINUATION } \#f\ \langle d_0\ d_1\ d_2\ d_3 \rangle \frown d_4^* \rangle$
$\#f$	otherwise

Implementation:

```
(or (= d halt)
    (and (pointer? d) (= (stob-type d) hdr/continuation)))
```

Function: continuation-stob-template d
Domain conditions:

$$\mathcal{S}(d, s) = \langle \text{CONTINUATION } \#f\ \langle d_0\ d_1\ d_2\ d_3 \rangle \frown d_4^* \rangle$$

Return value:

$$d_0$$

Implementation:

```
(stob-desc-ref d 0)
```

Function: continuation-stob-offset d
Domain conditions:
$\mathcal{S}(d, s) = \langle \text{CONTINUATION } \#f \ \langle d_0 \ d_1 \ d_2 \ d_3 \rangle \frown d_4^* \rangle$
Return value:
d_1

Implementation:

```
(stob-desc-ref d 1)
```

Function: continuation-stob-env d
Domain conditions:
$\mathcal{S}(d, s) = \langle \text{CONTINUATION } \#f \ \langle d_0 \ d_1 \ d_2 \ d_3 \rangle \frown d_4^* \rangle$
Return value:
d_2

Implementation:

```
(stob-desc-ref d 2)
```

Function: continuation-stob-cont d
Domain conditions:
$\mathcal{S}(d, s) = \langle \text{CONTINUATION } \#f \ \langle d_0 \ d_1 \ d_2 \ d_3 \rangle \frown d_4^* \rangle$
Return value:
d_3

Implementation:

```
(stob-desc-ref d 3)
```

Function: continuation-stob-template-init! d *val*
Domain conditions:

$d = \langle \text{PTR } l \rangle$
$\mathcal{S}(d, s) = \langle \text{CONTINUATION } \#f \ \langle d_0 \ d_1 \ d_2 \ d_3 \rangle \frown d_4^* \rangle$
Changes:
$s' = s[l \mapsto val]$

Implementation:

`(stob-desc-set! d 0` *val*`)`

Function: continuation-stob-offset-init! d *val*
Domain conditions:
$d = \langle \text{PTR } l \rangle$
$\mathcal{S}(d, s) = \langle \text{CONTINUATION } \#f \ \langle d_0 \ d_1 \ d_2 \ d_3 \rangle \frown d_4^* \rangle$
Changes:
$s' = s[l +_A 1 \mapsto val]$

Implementation:

`(stob-desc-set! d 1` *val*`)`

Function: continuation-stob-env-init! d *val*
Domain conditions:
$d = \langle \text{PTR } l \rangle$
$\mathcal{S}(d, s) = \langle \text{CONTINUATION } \#f \ \langle d_0 \ d_1 \ d_2 \ d_3 \rangle \frown d_4^* \rangle$
Changes:
$s' = s[l +_A 2 \mapsto val]$

Implementation:

`(stob-desc-set! d 2` *val*`)`

Function: continuation-stob-cont-init! d *val*
Domain conditions:
$d = \langle \text{PTR } l \rangle$
$\mathcal{S}(d, s) = \langle \text{CONTINUATION } \#f \ \langle d_0 \ d_1 \ d_2 \ d_3 \rangle \frown d_4^* \rangle$
Changes:

$$s' = s[l +_A 3 \mapsto val]$$

Implementation:

```
(stob-desc-set! d 3 val)
```

Function: make-environment-stob!
Domain conditions:

Changes:

$$s' = s +_1 \langle \text{ENVIRONMENT } \#t\ a \rangle$$

Implementation:

```
(make-desc-stob-from-stack! hdr/environment #t)
```

Function: environment-stob? d
Domain conditions:

Return value:

$\#t$	if $d = \langle \text{IMMEDIATE EMPTY-ENV} \rangle$
$\#t$	if $d = \langle \text{PTR } l \rangle$ and $\mathcal{S}(d, s) = \langle \text{ENVIRONMENT } \#t\ v^* \rangle$
$\#f$	otherwise

Implementation:

```
(or (= d emptyenv)
    (and (pointer? d) (= (stob-type d) hdr/environment)))
```

Function: environment-stob-length d
Domain conditions:

$$\mathcal{S}(d, s) = \langle \text{ENVIRONMENT } \#t\ v^* \rangle$$

Return value:

$$\#v^*$$

Implementation:

```
(stob-size-in-cells d)
```

Function: environment-stob-parent d
Domain conditions:
$\mathcal{S}(d, s) = \langle \text{ENVIRONMENT } \#t\ v^* \rangle$
Return value:
$v^*(0)$

Implementation:

```
(stob-desc-ref d 0)
```

Function: environment-stob-ref d i
Domain conditions:
$\mathcal{S}(d, s) = \langle \text{ENVIRONMENT } \#t\ v^* \rangle$
$1 \leq i < \#v^*$
Return value:
$v^*(i)$

Implementation:

```
(stob-desc-ref d i)
```

Function: environment-stob-set! d i *val*
Domain conditions:
$d = \langle \text{PTR } l \rangle$
$\mathcal{S}(d, s) = \langle \text{ENVIRONMENT } \#t\ v^* \rangle$
$1 \leq i < \#v^*$
Changes:
$s' = s[l +_A i \mapsto val]$

Implementation:

```
(stob-desc-set! d i val)
```

Function: make-location-stob!
Domain conditions:

Changes:
$s' = s \oplus_0 \langle \text{LOCATION } \#t\ 2 \rangle$

Implementation:

```
(make-desc-stob! hdr/location #t 2)
```

Function: location-stob? d
Domain conditions:

Return value:
$\#t$ if $d = \langle \text{PTR } l \rangle$ and $\mathcal{S}(d, s) = \langle \text{LOCATION } \#t\ \langle d_0\ d_1 \rangle \rangle$
$\#f$ otherwise

Implementation:

```
(and (pointer? d) (= (stob-type d) hdr/location))
```

Function: location-stob-value d
Domain conditions:
$\mathcal{S}(d, s) = \langle \text{LOCATION } \#t\ \langle d_0\ d_1 \rangle \rangle$
Return value:
d_0

Implementation:

```
(stob-desc-ref d 0)
```

Function: location-stob-name d
Domain conditions:
$\mathcal{S}(d, s) = \langle \text{LOCATION } \#t\ \langle d_0\ d_1 \rangle \rangle$
Return value:
d_1

Implementation:

```
(stob-desc-ref d 1)
```

Function: location-stob-value-set! d *val*
Domain conditions:

$d = \langle \text{PTR } l \rangle$
$\mathcal{S}(d, s) = \langle \text{LOCATION } \#t \ \langle d_0 \ d_1 \rangle \rangle$

Changes:

$s' = s[l \mapsto val]$

Implementation:

```
(stob-desc-set! d 0 val)
```

Function: make-mutable-pair-stob!
Domain conditions:

Changes:

$s' = s \oplus_0 \langle \text{PAIR } \#\ 2 \rangle$

Implementation:

```
(make-desc-stob! hdr/pair #t 2)
```

Function: make-immutable-pair-stob!
Domain conditions:

Changes:

$s' = s \oplus_0 \langle \text{PAIR } \#f\ 2 \rangle$

Implementation:

```
(make-desc-stob! hdr/pair #f 2)
```

Function: pair-stob? d
Domain conditions:

Return value:
$\#t$ if $d = \langle \text{PTR } l \rangle$ and $\mathcal{S}(d, s) = \langle \text{PAIR } p \langle d_0\ d_1 \rangle \rangle$
$\#f$ otherwise

Implementation:

```
(and (pointer? d) (= (stob-type d) hdr/pair))
```

Function: pair-stob-fst d
Domain conditions:
$\mathcal{S}(d, s) = \langle \text{PAIR } p \langle d_0\ d_1 \rangle \rangle$
Return value:
d_0

Implementation:

```
(stob-desc-ref d 0)
```

Function: pair-stob-snd d
Domain conditions:
$\mathcal{S}(d, s) = \langle \text{PAIR } p \langle d_0\ d_1 \rangle \rangle$
Return value:
d_1

Implementation:

```
(stob-desc-ref d 1)
```

Function: pair-stob-fst-set! d *val*
Domain conditions:
$d = \langle \text{PTR } l \rangle$
$\mathcal{S}(d, s) = \langle \text{PAIR } \#t \langle d_0\ d_1 \rangle \rangle$
Changes:

$$s' = s[l \mapsto val]$$

Implementation:

```
(stob-desc-set! d 0 val)
```

Function: pair-stob-snd-set! d *val*
Domain conditions:

$$d = \langle \text{PTR } l \rangle$$
$$\mathcal{S}(d, s) = \langle \text{PAIR } \#t \ \langle d_0 \ d_1 \rangle \rangle$$

Changes:

$$s' = s[l +_A 1 \mapsto val]$$

Implementation:

```
(stob-desc-set! d 1 val)
```

Function: make-immutable-string-stob! m
Domain conditions:

Changes:

$$s' = s \oplus_0 \langle \text{STRING } \#f \ m \rangle$$

Implementation:

```
(make-desc-stob! hdr/string #f m)
```

Function: make-mutable-string-stob! m
Domain conditions:

Changes:

$$s' = s \oplus_0 \langle \text{STRING } \#t \ m \rangle$$

Implementation:

```
(make-desc-stob! hdr/string #t m)
```

Function: string-stob? d
Domain conditions:

Return value:
$\#t$ if $d = \langle \text{PTR } l \rangle$ and $\mathcal{S}(d, s) = \langle \text{STRING } p\ b^* \rangle$
$\#f$ otherwise

Implementation:

```
(and (pointer? d) (= (stob-type d) hdr/string))
```

Function: string-stob-length d
Domain conditions:
$\mathcal{S}(d, s) = \langle \text{STRING } p\ b^* \rangle$
Return value:
$\#b^*$

Implementation:

```
(stob-size-in-bins d)
```

Function: string-stob-ref d i
Domain conditions:
$\mathcal{S}(d, s) = \langle \text{STRING } p\ b^* \rangle$
$0 \leq i < (\#b^* - 1)$
Return value:
$b^*(i)$

Implementation:

```
(char->immediate (integer->char (stob-binarray-ref d i)))
```

Function: string-stob-set! d i c
Domain conditions:

$d = \langle \text{PTR } l \rangle$
$\mathcal{S}(d, s) = \langle \text{STRING } \#t\ b^* \rangle$
$0 \leq i < (\#b^* - 1)$
$c = \langle \text{IMMEDIATE } \langle \text{CHAR } b \rangle \rangle$
$i' = \mho(i + 1) - 1$
$b'^* = s(l +_A i')[(i \bmod bpw) \mapsto b]$

Changes:

$s' = s[(l +_A i') \mapsto b'^*]$

Implementation:

```
(stob-binarray-set! d i (char->integer (immediate->char c)))
```

Function: make-symbol-stob!
Domain conditions:

Changes:

$s' = s \oplus_0 \langle \text{SYMBOL } \#f\ 1 \rangle$

Implementation:

```
(make-desc-stob! hdr/symbol #f 1)
```

Function: symbol-stob-name-init! d *val*
Domain conditions:

$d = \langle \text{PTR } l \rangle$
$\mathcal{S}(d, s) = \langle \text{SYMBOL } \#f\ \langle v \rangle \rangle$

Changes:

$s' = s[l \mapsto val]$

Implementation:

```
(stob-desc-set! d 0 val)
```

Function: symbol-stob? d
Domain conditions:

Return value:

$\#t$ if $d = \langle \text{PTR } l \rangle$ and $\mathcal{S}(d, s) = \langle \text{SYMBOL } \#f \ \langle v \rangle \rangle$

$\#f$ otherwise

Implementation:

```
(and (pointer? d) (= (stob-type d) hdr/symbol))
```

Function: symbol-stob-name d

Domain conditions:

$\mathcal{S}(d, s) = \langle \text{SYMBOL } \#f \ \langle v \rangle \rangle$

Return value:

v

Implementation:

```
(stob-desc-ref d 0)
```

Function: make-template-stob! m

Domain conditions:

Changes:

$s' = s \oplus_0 \langle \text{TEMPLATE } \#f \ m \rangle$

Implementation:

```
(make-desc-stob! hdr/template #f m)
```

Function: template-stob? d

Domain conditions:

Return value:

$\#t$ if $d = \langle \text{PTR } l \rangle$ and $\mathcal{S}(d, s) = \langle \text{TEMPLATE } \#f \ v^* \rangle$

$\#f$ otherwise

Implementation:

```
(and (pointer? d) (= (stob-type d) hdr/template))
```

Function: template-stob-length d
Domain conditions:
$\mathcal{S}(d, s) = \langle \text{TEMPLATE } \#f\ v^* \rangle$
Return value:
$\#v^*$

Implementation:

```
(stob-size-in-cells d)
```

Function: template-stob-codevector d
Domain conditions:
$\mathcal{S}(d, s) = \langle \text{TEMPLATE } \#f\ v^* \rangle$
Return value:
$v^*(0)$

Implementation:

```
(stob-desc-ref d 0)
```

Function: template-stob-ref d i
Domain conditions:
$\mathcal{S}(d, s) = \langle \text{TEMPLATE } \#f\ v^* \rangle$
$1 \leq i < \#v^*$
Return value:
$v^*(i)$

Implementation:

```
(stob-desc-ref d i)
```

Function: template-stob-init! d i *val*
Domain conditions:

$d = \langle \text{PTR } l \rangle$
$\mathcal{S}(d, s) = \langle \text{TEMPLATE } \#f\ v^* \rangle$
$0 \leq i < \#v^*$

Changes:

$s' = s[l +_A i \mapsto val]$

Implementation:

`(stob-desc-set! d i` *val*`)`

Function: make-mutable-vector-stob! m
Domain conditions:

Changes:

$s' = s \oplus_0 \langle \text{VECTOR } \#t\ m \rangle$

Implementation:

`(make-desc-stob! hdr/vector #t m)`

Function: make-immutable-vector-stob! m
Domain conditions:

Changes:

$s' = s \oplus_0 \langle \text{VECTOR } \#f\ m \rangle$

Implementation:

`(make-desc-stob! hdr/vector #f m)`

Function: vector-stob? d
Domain conditions:

Return value:

$\#t$ if $d = \langle \text{PTR } l \rangle$ and $\mathcal{S}(d, s) = \langle \text{VECTOR } p\ v^* \rangle$
$\#f$ otherwise

Implementation:

```
(and (pointer? d) (= (stob-type d) hdr/vector))
```

Function: vector-stob-length d
Domain conditions:
$\mathcal{S}(d, s) = \langle \text{VECTOR } p\ v^* \rangle$
Return value:
$\#v^*$

Implementation:

```
(stob-size-in-cells d)
```

Function: vector-stob-ref d i
Domain conditions:
$\mathcal{S}(d, s) = \langle \text{VECTOR } p\ v^* \rangle$
$0 \leq i < \#v^*$
Return value:
$v^*(i)$

Implementation:

```
(stob-desc-ref d i)
```

Function: vector-stob-set! d i *val*
Domain conditions:
$d = \langle \text{PTR } l \rangle$
$\mathcal{S}(d, s) = \langle \text{VECTOR } \#t\ v^* \rangle$
$0 \leq i < \#v^*$
Changes:
$s' = s[l +_A i \mapsto val]$

Implementation:

```
(stob-desc-set! d i val)
```

2.2.7 Miscellaneous Auxiliary Functions

Before presenting the action rules, we present several auxiliary functions.

The function *env-frame*(u, n, s) returns the environment that is nested n levels deep within the environment pointed to by u in store s.

$$\begin{aligned}&\textit{env-frame}(u, 0, s) = u, \text{ and}\\ &\textit{env-frame}(u, n+1, s) = \textit{env-frame}(u', n, s)\\ &\qquad\qquad\qquad\qquad \textit{if } \mathcal{S}(u, s) = \langle \textsc{environment } \#t\ u'{::}vd^* \rangle\end{aligned}$$

The function *stack-to-list*(lp, n, a, s) pops the first n values of a stack a and stores them as a linked list in store s. The tail of this linked list is the list pointed to by lp in store s. The function returns a pointer to the new linked list, together with the modified stack and store.

$$\begin{aligned}&\textit{stack-to-list}(lp, 0, a, s) = \langle lp\ a\ s \rangle, \text{ and}\\ &\textit{stack-to-list}(lp, n+1, a, s) =\\ &\qquad \textit{stack-to-list}(\langle \textsc{ptr } \langle 0\ \#s_0 + 1 \rangle \rangle, n, a \dagger 1, s +_0 \langle \textsc{pair } \#f\ \langle a(0)\ lp \rangle \rangle)\end{aligned}$$

The function *list-to-stack*(p, a, s) returns the stack $a' \frown a$, where a' is the stack represented by the linked list pointed to by p in store s.

$$\begin{aligned}&\textit{list-to-stack}(\langle \textsc{immediate null} \rangle, a, s) = a, \text{ and}\\ &\textit{list-to-stack}(p, a, s) = \textit{list-to-stack}(p', v :: a, s)\\ &\qquad\qquad\qquad\qquad \text{if } \mathcal{S}(p, s) = \langle \textsc{pair } p\ \langle v\ p' \rangle \rangle\end{aligned}$$

The function *compute-offset* takes two integers and returns an integer representing an offset they are together encoding. For $0 \leq n_0, n_1$,

$$\textit{compute-offset}(n_0, n_1) = (256 * n_0) + n_1.$$

We will usually write $n_0 \otimes n_1$ for *compute-offset*(n_0, n_1).

2.2.8 Action Rules

We present *actions* as the union of subfunctions called *(action) rules*. The action rules are functions from pairwise disjoint subsets of *states* $\times$ *inputs* $\times$ *outputs* into *states* $\times$ *inputs* $\times$ *outputs* where *states* is the set of SBCM states, and *inputs* and *outputs* are sets of finite sequences of characters. Since the domains of these functions are disjoint, the union of these functions is a partial function from *states* $\times$ *inputs* $\times$ *outputs* into *states* $\times$ *inputs* $\times$ *outputs*. We denote this function by $\mathcal{R}$.

Let Σ, i^*, and o^* be a state, input sequence, and output sequence respectively. The value of the function $\mathcal{R}$ at $\langle \Sigma, i^*, o^* \rangle$ only depends on Σ and possibly on the first element of i^*. Let $\mathcal{R}(\langle \Sigma, i^*, o^* \rangle) = \langle \Sigma', i'^*, o'^* \rangle$. Then, one of the following hold:

- The input and output sequences are unchanged, i.e., $i'^* = i^*$ and $o'^* = o^*$. We call such rules *pure rules.*
- The rule drops the first element of the argument input sequence, i.e., $i'^* = i^* \dagger 1$ and $o'^* = o^*$. We call such rules *input port rules.*
- The rule appends a sequence of values to the argument output sequence, i.e., $i'^* = i^*$ and $o'^* = o^* \frown o''^*$. We call such rules *output port rules.*

We define a function $\mathcal{R}^*$ as follows:

$$\mathcal{R}^* \stackrel{\text{def}}{=} \begin{array}{ll} \mathcal{R}^*(\mathcal{R}(\langle \Sigma, i^*, o^* \rangle)) & \text{if } \mathcal{R}(\langle \Sigma, i^*, o^* \rangle) \text{ is defined} \\ \langle \Sigma, i^*, o^* \rangle & \text{otherwise} \end{array}$$

If, when started in state $\langle \Sigma, i^*, o^* \rangle$, the state machine halts, then $\mathcal{R}^*(\langle \Sigma, i^*, o^* \rangle)$ is defined and is the state the machine halts in. Otherwise, $\mathcal{R}^*(\langle \Sigma, i^*, o^* \rangle)$ is undefined.

We present the definition of the action rules below. We also present an "implementation" of the action rules of the MSBCM, with the state transformations of the action rules being specified as the composition of several microcode functions.

We modify the presentation format for pure action rules that is described in the VLISP Flattener report [2]. For each pure 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), a specification of the new values of some registers, and an "implementation" of the rule. 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, n, c, v, a, u, k, s, r_1, r_2, r_3, and r_4. Call these the original register variables. The new values of the registers are indicated by the same variables with primes attached: t', n', c', v', a', u', k', s', r'_1, r'_2, r'_3, and r'_4. Call these the new register variables. New register variables occur only as the left hand sides of equations specifying new register values. Registers for which no new value is given are tacitly asserted to remain

unchanged. Input and output must both be null. We use the two auxiliary functions defined below. Let $\mathcal{S}(c,s) = \langle \text{CODEVECTOR } \#f\ b^* \rangle$. Then,

$$\begin{array}{rcll} \mathit{current\_inst}(n,c,s) & \stackrel{\text{def}}{=} & b^*(n) & \text{if } 0 \le n < \#b^* \\ \mathit{current\_inst\_param}(n,i,c,s) & \stackrel{\text{def}}{=} & b^*(n+i) & \text{if } 0 \le n+i < \#b^* \end{array}$$

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 $j + 12$ places

(†) $R(t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4, x_1, \ldots, x_j)$.

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

The "implementation" of the action rules expresses each rule as the composition of stored object manipulators, microcode functions, and other auxiliary functions. The implementation is expressed as a Scheme program. Note that Scheme is a call-by-value language that implicitly threads the global state through function calls; hence the global state does not appear as an explicit parameter.

Rule 1: return

Domain conditions:

$\mathit{current\_inst}(n,c,s) =$ `return`

$k =$ HALT

Changes:

Halt in final state Σ

Implementation:

```
(if (= (cont-ref) halt)
    (halt!)
    (begin
      (spare2-set! (cont-ref))
      (template-set! (continuation-stob-template (spare2-ref)))
```

```
(codevector-set! (template-stob-codevector (template-ref)))
(offset-set! (fixnum->int
              (continuation-stob-offset (spare2-ref))))
(env-set! (continuation-stob-env (spare2-ref)))
(cont-set! (continuation-stob-cont (spare2-ref)))
(stack-restore! (spare2-ref))
))
```

Rule 2: return

Domain conditions:

$current\_inst(n, c, s) = \texttt{return}$

$\mathcal{S}(k, s) = \langle \textsc{continuation}\ \#f\ \langle t_1\ n_1\ u_1\ k_1 \rangle \frown a_1 \rangle$

$\mathcal{S}(t_1, s) = \langle \textsc{template}\ \#f\ c_1 :: o^* \rangle$

$n_1 = \langle \textsc{fixnum}\ m \rangle$

Changes:

$t' = t_1$

$n' = m$

$c' = c_1$

$a' = a_1$

$u' = u_1$

$k' = k_1$

$r'_2 = k$

Implementation:

```
(if (= (cont-ref) halt)
    (halt!)
    (begin
      (spare2-set! (cont-ref))
      (template-set! (continuation-stob-template (spare2-ref)))
      (codevector-set! (template-stob-codevector (template-ref)))
      (offset-set! (fixnum->int
                    (continuation-stob-offset (spare2-ref))))
      (env-set! (continuation-stob-env (spare2-ref)))
      (cont-set! (continuation-stob-cont (spare2-ref)))
      (stack-restore! (spare2-ref))
      ))
```

Rule 3: call m

Domain conditions:

$current\_inst(n, c, s) = \texttt{call}$
$current\_inst\_param(n, 1, c, s) = m$
$\mathcal{S}(v, s) = \langle \textsc{closure}\ \#f\ \langle t_1\ u_1\ d_1 \rangle \rangle$
$\mathcal{S}(t_1, s) = \langle \textsc{template}\ \#f\ c_1 :: o^* \rangle$

Changes:

$t' = t_1$
$n' = 0$
$c' = c_1$
$u' = u_1$

Implementation:

```
(assert! (closure-stob? (value-ref))
         "op/call: value register does not contain a closure")
(template-set! (closure-stob-template (value-ref)))
(env-set! (closure-stob-env (value-ref)))
(codevector-set! (template-stob-codevector (template-ref)))
(offset-set! 0)
```

Rule 4**: jump-if-false** m_1 m_2

Domain conditions:

$current\_inst(n, c, s) = \texttt{jump-if-false}$
$current\_inst\_param(n, 1, c, s) = m_1$
$current\_inst\_param(n, 2, c, s) = m_2$
$v \neq \langle \textsc{immediate}\ \textsc{false} \rangle$

Changes:

$n' = n + 3$

Implementation:

```
(if (= (value-ref) false)
    (offset-inc! (+ 3 (compute-offset (current-inst-param 1)
                                      (current-inst-param 2))))
    (offset-inc! 3))
```

Rule 5**: jump-if-false** m_1 m_2

Domain conditions:

$current\_inst(n, c, s) =$ `jump-if-false`
$current\_inst\_param(n, 1, c, s) = m_1$
$current\_inst\_param(n, 2, c, s) = m_2$
$v = \langle$IMMEDIATE FALSE$\rangle$

Changes:

$n' = n + 3 + (m_1 \otimes m_2)$

Implementation:

```
(if (= (value-ref) false)
    (offset-inc! (+ 3 (compute-offset (current-inst-param 1)
                                      (current-inst-param 2))))
    (offset-inc! 3))
```

Rule 6: jump m_1 m_2

Domain conditions:

$current\_inst(n, c, s) =$ `jump`
$current\_inst\_param(n, 1, c, s) = m_1$
$current\_inst\_param(n, 2, c, s) = m_2$

Changes:

$n' = n + 3 + (m_1 \otimes m_2)$

Implementation:

```
(offset-inc! (+ 3 (compute-offset (current-inst-param 1)
                                  (current-inst-param 2))))
```

Rule 7: make-cont m_1 m_2 m_3

Domain conditions:

$current\_inst(n, c, s) =$ `make-cont`
$current\_inst\_param(n, 1, c, s) = m_1$
$current\_inst\_param(n, 2, c, s) = m_2$
$current\_inst\_param(n, 3, c, s) = m_3$
$m = n + 4 + (m_1 \otimes m_2)$

Changes:

$n' = n + 4$
$a' = \langle\rangle$
$k' = \langle \text{PTR } \langle 1 \ -\#s - \#a - 3 \rangle \rangle$
$s' = s +_1 \langle \text{CONTINUATION } \#f \ \langle t \ \langle \text{FIXNUM } m \rangle \ u \ k \rangle \frown a \rangle$
$r'_1 = \langle \text{PTR } \langle 1 \ -\#s - \#a - 3 \rangle \rangle$

Implementation:

```
(stack-push! (cont-ref))
(stack-push! (env-ref))
(stack-push! (int->fixnum
              (+ (offset-ref) 4
                 (compute-offset (current-inst-param 1)
                                 (current-inst-param 2)))))
(stack-push! (template-ref))
(make-continuation-stob!)
(cont-set! (spare1-ref))
(offset-inc! 4)
```

Rule 8: literal m

Domain conditions:

$current\_inst(n, c, s) =$ `literal`
$current\_inst\_param(n, 1, c, s) = m$
$\mathcal{S}(t, s) = \langle \text{TEMPLATE } \#f \ c{::}o^* \rangle$
$1 \leq m < (\#o^* + 1)$
$o_m = (c{::}o^*)(m)$

Changes:

$n' = n + 2$
$v' = o_m$

Implementation:

```
(assert! (in-bounds? (current-inst-param 1)
                     1 (template-stob-length (template-ref)))
         "op/literal: index is outside template bounds")
(value-set! (template-stob-ref (template-ref)
                               (current-inst-param 1)))
(offset-inc! 2)
```

Rule 9: closure m

Domain conditions:

$current\_inst(n, c, s) = \texttt{closure}$
$current\_inst\_param\,(n, 1, c, s) = m$
$\mathcal{S}(t, s) = \langle \textsc{template}\ \#f\ c{::}o^* \rangle$
$1 \leq m < (\#o^* + 1)$
$o_m = (c{::}o^*)(m)$
$\mathcal{S}(o_m, s) = \langle \textsc{template}\ \#f\ c_1{::}o_1^* \rangle$

Changes:

$n' = n + 2$
$v' = \langle \textsc{ptr}\ \langle 0\ (\#s_0 + 1) \rangle \rangle$
$s' = s +_0 \langle \textsc{closure}\ \#f\ \langle o_m\ u\ \langle \textsc{immediate unspecified} \rangle \rangle \rangle$
$r_1' = \langle \textsc{ptr}\ \langle 0\ (\#s_0 + 1) \rangle \rangle$
$r_2' = o_m$

Implementation:

```
(assert! (in-bounds? (current-inst-param 1)
                     1 (template-stob-length (template-ref)))
         "op/closure: index is outside template bounds")
(make-closure-stob!)
(spare2-set! (template-stob-ref (template-ref)
                                (current-inst-param 1)))
(assert! (template-stob? (spare2-ref))
         "op/closure: indexed datum is not a template stob")
(closure-stob-template-init! (spare1-ref) (spare2-ref))
(closure-stob-env-init! (spare1-ref) (env-ref))
(value-set! (spare1-ref))
(offset-inc! 2)
```

Rule 10**: global** m

Domain conditions:

$current\_inst(n, c, s) = \texttt{global}$
$current\_inst\_param\,(n, 1, c, s) = m$
$\mathcal{S}(t, s) = \langle \textsc{template}\ \#f\ c{::}o^* \rangle$
$1 \leq m < (\#o^* + 1)$
$o_m = (c{::}o^*)(m)$
$\mathcal{S}(o_m, s) = \langle \textsc{location}\ \#\ \langle v_1\ x \rangle \rangle$
$v_1 \neq \langle \textsc{immediate undefined} \rangle$

Changes:

$n' = n + 2$
$v' = v_1$
$r_2' = o_m$

Implementation:

```
(assert! (in-bounds? (current-inst-param 1)
                     1 (template-stob-length (template-ref)))
 "op/global: index is outside template bounds")
(spare2-set! (template-stob-ref (template-ref)
                                (current-inst-param 1)))
(assert! (location-stob? (spare2-ref))
         "op/global: indexed datum is not a location stob")
(value-set! (location-stob-value (spare2-ref)))
(assert! (not (immediate-undefined? (value-ref)))
         "op/global: location's value is undefined")
(offset-inc! 2)
```

Rule 11**: set-global!** m

Domain conditions:

$current\_inst(n, c, s) =$ `set-global!`
$current\_inst\_param(n, 1, c, s) = m$
$\mathcal{S}(t, s) = \langle \text{TEMPLATE } \#f\ c{::}o^* \rangle$
$1 \leq m < (\#o^* + 1)$
$o_m = (c{::}o^*)(m) = \langle \text{PTR } l \rangle$
$\mathcal{S}(o_m, s) = \langle \text{LOCATION } \#t\ \langle v_1\ x \rangle \rangle$

Changes:

$n' = n + 2$
$s' = s[l \mapsto v]$
$v' = \langle \text{IMMEDIATE UNSPECIFIED} \rangle$
$r_2' = o_m$

Implementation:

```
(assert! (in-bounds? (current-inst-param 1)
                     1 (template-stob-length (template-ref)))
         "op/set-global!: index is outside template bounds")
(spare2-set! (template-stob-ref (template-ref)
                                (current-inst-param 1)))
(assert! (location-stob? (spare2-ref))
         "op/set-global!: indexed datum is not a location stob")
(location-stob-value-set! (spare2-ref) (value-ref))
```

```
(value-set! unspecified)
(offset-inc! 2)
```

Rule 12**: local** m_1 m_2

Domain conditions:

$current\_inst(n, c, s) =$ `local`
$current\_inst\_param(n, 1, c, s) = m_1$
$current\_inst\_param(n, 2, c, s) = m_2$
$u_1 = env\text{-}frame(u, m_1, s)$
$\mathcal{S}(u_1, s) = \langle \text{ENVIRONMENT } \#t\ u_2 :: vd^* \rangle$
$v_1 = (u_2 :: vd^*)(m_2)$
$v_1 \neq \langle \text{IMMEDIATE UNDEFINED} \rangle$

Changes:

$n' = n + 3$
$v' = v_1$

Implementation:

```
(value-set! (environment-stob-ref
             (env-frame! (env-ref) (current-inst-param 1))
             (current-inst-param 2)))
(assert! (not (immediate-undefined? (value-ref)))
         "op/local: indexed local variable has undefined value")
(offset-inc! 3)
```

Rule 13**: set-local!** m_1 m_2

Domain conditions:

$current\_inst(n, c, s) =$ `set-local!`
$current\_inst\_param(n, 1, c, s) = m_1$
$current\_inst\_param(n, 2, c, s) = m_2$
$u_1 = env\text{-}frame(u, m_1, s) = \langle \text{PTR } l_1 \rangle$
$S(u_1, s) = \langle \text{ENVIRONMENT } \#t\ u_2 :: vd^* \rangle$
$l = l_1 + m_2$

Changes:

$n' = n + 3$
$v' = \langle \text{IMMEDIATE UNSPECIFIED} \rangle$
$s' = s[l \mapsto v]$

Implementation:

```
(environment-stob-set!
 (env-frame! (env-ref) (current-inst-param 1))
 (current-inst-param 2)
 (value-ref))
(value-set! unspecified)
(offset-inc! 3)
```

Rule 14**: push**
Domain conditions:

$current\_inst(n, c, s) =$ `push`

Changes:

$n' = n + 1$
$a' = v::a$

Implementation:

```
(stack-push! (value-ref))
(offset-inc! 1)
```

Rule 15**: make-env** m
Domain conditions:

$current\_inst(n, c, s) =$ `make-env`
$current\_inst\_param(n, 1, c, s) = m$
$m = \#a$

Changes:

$n' = n + 2$
$a' = \langle\rangle$
$u' = \langle \text{PTR } \langle 1 \ - \#s - \#a \rangle \rangle$
$s' = s +_1 \langle \text{ENVIRONMENT } \#t \ u::a \rangle r'_1 = \langle \text{PTR } \langle 1 \ - \#s - \#a \rangle \rangle$

Implementation:

```
(stack-push! (env-ref))
(make-environment-stob)
(offset-inc! 2)
(env-set! (spare1-ref))
```

Rule 16**: make-rest-list** m
Domain conditions:

$current\_inst(n, c, s) = \texttt{make-rest-list}$
$current\_inst\_param\,(n, 1, c, s) = m$
$\#a \geq m$
$\langle v_1\ a_1\ s_1 \rangle = stack\text{-}to\text{-}list(\langle \text{IMMEDIATE NULL} \rangle, (\#a - m), a, s)$

Changes:

$n' = n + 2$
$v' = v_1$
$a' = a_1$
$s' = s_1$
$r'_1 = \texttt{if}\ (\#a - m) = 0\ \texttt{then}\ r_1$ else v_1
$r'_2 = v_1$
$r'_3 = 0$

Implementation:

```
(let ((m (current-inst-param 1)))
  (assert! (>= (stack-length) m)
    "op/make-rest-list: argument is larger than stack size")
  (spare3-set! (- (stack-length) m))
  (spare2-set! null)
  (let loop ()
    (if (= (spare3-ref) 0)
        (begin
          (value-set! (spare2-ref))
          (offset-inc! 2))
        (begin
          (make-mutable-pair-stob!)
          (pair-stob-fst-set! (spare1-ref) (stack-top))
          (stack-pop!)
          (pair-stob-snd-set! (spare1-ref) (spare2-ref))
          (spare2-set! (spare1-ref))
          (spare3-set! (- (spare3-ref) 1))
          (loop))
        )))
```

Rule 17: unspecified

Domain conditions:

$current\_inst(n, c, s) = \texttt{unspecified}$

Changes:

$n' = n + 1$
$v' = \langle \text{IMMEDIATE UNSPECIFIED} \rangle$

Implementation:

```
(value-set! unspecified)
(offset-inc! 1)
```

Rule 18**: check-args=** m
Domain conditions:

$current\_inst(n, c, s) =$ `check-args=`
$current\_inst\_param(n, 1, c, s) = m$
$\#a = m$

Changes:

$n' = n + 2$

Implementation:

```
(assert! (= (stack-length) (current-inst-param 1))
         "op/check-args=: Incorrect number of arguments.")
(offset-inc! 2)
```

Rule 19**: check-args**$>=$ m
Domain conditions:

$current\_inst(n, c, s) =$ `check-args>=`
$current\_inst\_param(n, 1, c, s) = m$
$\#a \geq m$

Changes:

$n' = n + 2$

Implementation:

```
(assert! (>= (stack-length) (current-inst-param 1))
         "op/check-args>=: Incorrect number of arguments.")
(offset-inc! 2)
```

Rule 20**: primitive-throw**
Domain conditions:

$current\_inst(n, c, s) = \texttt{primitive-throw}$
$\mathcal{S}(v, s) = \langle \text{CONTINUATION } \#f\ \langle t_1\ n_1\ u_1\ k_1 \rangle \frown a_1 \rangle$

Changes:

$n' = n + 1$
$k' = v$

Implementation:

```
(assert! (continuation-stob? (value-ref))
         "op/primitive-throw: value is not a continuation")
(cont-set! (value-ref))
(offset-inc! 1)
```

2.3 SBC Operational Semantics

We first define a function *load* that maps SBC stores to MSBC stores. An SBC store is represented as the first store segment within an MSBC store, and all SBC pointers within the SBC store are mapped to corresponding MSBC pointers. Let s^{sbc} be an SBC store. Then

$$relocate(cell) \stackrel{\text{def}}{=} \begin{array}{ll} \langle \text{PTR } \langle 0\ m \rangle \rangle & \text{if } cell = \langle \text{PTR } m \rangle \\ cell & \text{otherwise} \end{array}$$

$$load(s^{sbc}) \stackrel{\text{def}}{=} \langle s_0\ \langle\rangle\ \langle\rangle\ \langle\rangle \rangle$$
where $\#s_0 = \#s^{sbc}$
and for all $0 \le m < \#s_0$
$s_0(m) = relocate(s^{sbc}(m))$

The MSBCM *Loader* L_{msbcm} is defined to be a function that maps SBC programs to MSBCM initial states such that:

$$L_{msbcm}(\langle s^{sbc}\ t^{sbc} \rangle) \stackrel{\text{def}}{=} \begin{array}{l} \langle t, 0, c, d_u, \langle\rangle, \langle \text{IMMEDIATE EMPTY-ENV} \rangle, \\ \quad \langle \text{IMMEDIATE HALT} \rangle, s, d_u, d_u, 0, d_u \rangle \\ \text{if } s = load(s^{sbc}) \text{ and } t = relocate(t^{sbc}) \\ \text{and } \mathcal{S}(t, s) = \langle \text{TEMPLATE } \#f\ c{::}o^* \rangle \\ \text{and } d_u = \langle \text{IMMEDIATE UNSPECIFIED} \rangle. \end{array}$$

Note that SBCM initial states are MSBC terms, and hence the loader involves coercing SBC terms to MSBC terms. We can easily show that the loader maps SBC programs to MSBCM initial states:

Lemma 6 *If $\mathcal{P}^{sbc} = \langle s^{sbc}\ t^{sbc} \rangle$ is an* SBC *program, then $L_{msbcm}(\mathcal{P}^{sbc})$ is an initial* MSBCM *state.*

Proof: Let $L_{msbcm}(\langle s^{sbc}\ t^{sbc}\rangle)$ be defined as above. All conditions for it to be an initial MSBCM state are trivial except for the conditions that s be an MSBC store and t be a pointer to a template object in s.

By definition of s, if $s(l)$ is defined for some l, then l has the form $\langle 0, m\rangle$ for some $0 \leq m < \#s^{sbc}$, and further $s(l) = \mathit{relocate}(s^{sbc}(l))$. Since $\mathcal{P}^{sbc}$ is an SBC program, we have by definition that s^{sbc} is an SBC store. Using these facts, it is trivial to show that s is an MSBC store.

Since $\mathcal{P}^{sbc}$ is an SBC program, we have by definition that t^{sbc} is a pointer to a template object in s^{sbc}. Then it is trivial to show that $t = \mathit{relocate}(t^{sbc})$ is a pointer to a template object in s.

□

The MSBCM *Answer* A_{msbcm} is defined to be a partial function that maps MSBCM halt states to integers such that:

$$A_{msbcm}(\langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4\rangle) \stackrel{\text{def}}{=} m \qquad \text{if } v = \langle \text{FIXNUM } m\rangle$$

Definition 7 *Let $\mathcal{P}$ be an* SBC *program. Then $L_{msbcm}(\mathcal{P})$ is an initial state of the* MSBCM. *If $\mathcal{R}^*_{msbcm}(L_{msbcm}(\mathcal{P}))$ is defined, then the meaning of program $\mathcal{P}$ as given by the* SBC *operational semantics is*

$$\mathcal{O}_{msbcm}[\![\mathcal{P}]\!] \stackrel{\text{def}}{=} A_{msbcm}(\mathcal{R}^*_{msbcm}(L_{msbcm}(\mathcal{P})))$$

Otherwise $\mathcal{O}_{msbcm}[\![\mathcal{P}]\!]$ is undefined.

Theorem 8 *If p is an* SBC *program, then its operational semantics as given by the* SBCM *yields the same answer as its operational semantics as given by the* MSBCM. *That is,*

$$\mathcal{O}_{sbcm}[\![p]\!] = \mathcal{O}_{msbcm}[\![p]\!]$$

Proof: The loader function L_{msbcm} is an extension of the function L_{sbcm} and differs only in the values it assigns the temporary registers r_1, r_2, r_3, and r_4. The action rules of the MSBCM are defined as extensions of the action rules of the SBCM, and differ only in the values of the temporary registers r_1, r_2, r_3, and r_4. The answer functions A_{sbcm} and A_{msbcm} do not depend on the values of the temporary registers. Since the operational semantics is defined as the composition of these three functions, it follows that the two semantics are equivalent.

□

Theorem 9 *The implementation of the action rules of the* MSBCM *is correct with respect to the specification of the action rules.*

Proof: This follows immediately from the definitions of the action rules, stored object manipulators, microcode functions, and auxiliary functions. In this report, we only present the proof for an example action rule. The proof is very simple, and suggests that the interested reader can verify the correctness of the other rules by perusal. A detailed examination of the correctness of the other action rules has been carried out. A manuscript presenting detailed proofs for (earlier versions) of the action rules is available.

Rule 21**: call** m
Domain conditions:

$current\_inst(n, c, s) = \texttt{call}$
$current\_inst\_param(n, 1, c, s) = m$
$\mathcal{S}(v, s) = \langle \textsc{closure}\ \#f\ \langle t_1\ u_1\ d_1 \rangle \rangle$
$\mathcal{S}(t_1, s) = \langle \textsc{template}\ \#f\ c_1 :: o^* \rangle$

Changes:

$t' = t_1$
$n' = 0$
$c' = c_1$
$u' = u_1$

Implementation:

```
(assert! (closure-stob? (value-ref))
         "op/call: value register does not contain a closure")
(template-set! (closure-stob-template (value-ref)))
(env-set! (closure-stob-env (value-ref)))
(codevector-set! (template-stob-codevector (template-ref)))
(offset-set! 0)
```

Let Σ be an MSBCM state such that the above domain conditions hold. We rewrite the implementation to make the global state explicit, and simulta-

neously verify the implementation.

Σ_1 = `(assert! (closure-stob? (value-ref` Σ`)` Σ`)`
`"op/call: value register does not contain a closure")`
Σ_2 = `(template-set!`
`(closure-stob-template (value-ref` Σ_1`)` Σ_1`)` Σ_1`)`
Σ_3 = `(env-set! (closure-stob-env (value-ref` Σ_2`)` Σ_2`)` Σ_2`)`
Σ_4 = `(codevector-set!`
`(template-stob-codevector (template-ref` Σ_3`)` Σ_3`)` Σ_3`)`
Σ_5 = `(offset-set! 0` Σ_4`)`

From the definitions of the above functions, we have that:

$$\begin{aligned}
\Sigma_1 &= \Sigma \\
\Sigma_2 &= \Sigma_1[t' = t_1] \\
\Sigma_3 &= \Sigma_2[u' = u_1] \\
\Sigma_4 &= \Sigma_3[c' = c_1] \\
\Sigma_5 &= \Sigma_4[n' = 0]
\end{aligned}$$

Thus, $\Sigma_5 = \Sigma[t' = t_1][u' = u_1][c' = c_1][n' = 0]$ as desired.

The other rules are verified similarly.

□

2.4 State Correspondence Relation

If $\simeq$ is a four-place relation, we write $(s^S, s \vdash x^S \simeq x)$ to mean that (s^S, x^S, s, x) is in the relation $\simeq$.

Let s^S be a SBCM store. Then

$$ldom(s^S) \stackrel{\text{def}}{=} dom(s^S)$$

is the set of SBCM locations l^S such that $s^S(l^S)$ is defined.

Let s be an MSBCM store. Then

$$ldom(s) \stackrel{\text{def}}{=} \{\langle i\ m\rangle \mid 0 \leq i < 4 \wedge m \in dom(s(i))\}$$

is the set of MSBCM locations l such that $s(l)$ is defined.

Definition 10 A *location correspondence* $\simeq_0 \subseteq (s^S \times l^S) \times (s \times l)$ is defined to be a relation such that for all s^S, s, the set $\{(l^S, l) \mid (s^S, s \vdash l^S \simeq_0 l)\}$ is a 1-to-1 map from a subset of $ldom(s^S)$ to a subset of $ldom(s)$. Further, if $(s^S, s \vdash l^S \simeq_0 l)$ then $s^S(l^S)$ and $s(l)$ are not headers.

Definition 11 Let $\simeq_0 \subseteq (s^S \times l^S) \times (s \times l)$ be a location correspondence. A *term correspondence* $\simeq \subseteq (s^S \times cell^S) \times (s \times cell)$ induced by $\simeq_0$ is defined as follows:

$(s^S, s \vdash cell^S \simeq cell)$ holds if one of the following hold:

- 1. $cell^S = \langle \text{PTR}\ l^S\rangle$
  2. $cell = \langle \text{PTR}\ l\rangle$
  3. $s^S(l^S - 1) = \langle \text{HEADER}\ h^S\ p^S\ m^S\rangle$
  4. $s(l -_A 1) = \langle \text{HEADER}\ h\ p\ m\rangle$
  5. $h^S = h$
  6. $p^S = p$
  7. $m^S = m$
  8. for all $0 \leq i < \mho m$, $(s^S, s \vdash (l^S + i) \simeq_0 (l +_A i))$
- $cell^S = \langle \text{FIXNUM}\ m\rangle = cell$
- $cell^S = imm = cell$
- $cell^S = b^* = cell$

Definition 12 Let $\Sigma_{sbcm} = \langle t^S, n^S, c^S, v^S, a^S, u^S, k^S, s^S \rangle$ be an SBCM state and let $\Sigma_{msbcm} = \langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4 \rangle$ be an MSBCM state. The *state correspondence relation* $\cong \subset state^S \times state$ is defined as follows:

$$\Sigma_{sbcm} \cong \Sigma_{msbcm}$$

if there exists a location correspondence $\simeq_0$ and a term correspondence $\simeq$ induced by $\simeq_0$ such that

- for all $l^S \in ldom(s^S)$ and $l \in ldom(s)$,
  $(s^S, s \vdash l^S \simeq_0 l) \Rightarrow (s^S, s \vdash s^S(l^S) \simeq s(l))$
- $(s^S, s \vdash t^S \simeq t)$
- $n^S = n$
- $(s^S, s \vdash v^S \simeq v)$
- $\#a^S = \#a$ and $(\forall 0 \leq i < \#a)(s^S, s \vdash a^S(i) \simeq a(i))$
- $(s^S, s \vdash u^S \simeq u)$
- $(s^S, s \vdash k^S \simeq k)$

2.5 Establishing State Correspondence

Let $d_u \stackrel{\text{def}}{=} \langle$IMMEDIATE UNSPECIFIED$\rangle$. The SBCM *Loader*, L_{sbcm}, is a function that maps SBC programs to SBCM initial states and is defined as follows:

$$\begin{aligned} L_{sbcm}(\langle s^{sbc}\ t^{sbc} \rangle) \stackrel{\text{def}}{=}\ & \langle t^{sbc}, 0, c^S, d_u, \langle\rangle, \langle\text{IMMEDIATE EMPTY-ENV}\rangle, \\ & \quad \langle\text{IMMEDIATE HALT}\rangle, s^{sbc} \rangle \\ & \text{if } \mathcal{S}(t^{sbc}, s^{sbc}) = \langle\text{TEMPLATE } \#f\ c^S :: o^* \rangle. \end{aligned}$$

The MSBCM *Loader*, L_{msbcm}, is a function that maps SBC programs to MSBCM initial states and is defined as follows:

$$\begin{aligned} L_{msbcm}(\langle s^{sbc}\ t^{sbc} \rangle) \stackrel{\text{def}}{=}\ & \langle t, 0, c, d_u, \langle\rangle, \langle\text{IMMEDIATE EMPTY-ENV}\rangle, \\ & \quad \langle\text{IMMEDIATE HALT}\rangle, s, d_u, d_u, 0, d_u \rangle \\ & \text{if } s = load(s^{sbc}) \text{ and } t = relocate\,(t^{sbc}) \\ & \text{and } \mathcal{S}(t, s) = \langle\text{TEMPLATE } \#f\ c :: o^* \rangle. \end{aligned}$$

where *load* and *relocate* are as defined in Section 2.3.

Lemma 13 *The* SBCM *Loader and the* MSBCM *Loader map* SBC *programs to initial states that are related by the state correspondence relation. That is, if* $\mathcal{P}$ *is an* SBC *program, then*

$$L_{sbcm}(\mathcal{P}) \cong L_{msbcm}(\mathcal{P})$$

Proof: Let

$$L_{sbcm}(\mathcal{P}) = \langle t^S, n^S, c^S, v^S, a^S, u^S, k^S, s^S \rangle$$
$$L_{msbcm}(\mathcal{P}) = \langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4 \rangle$$

Let the location correspondence $\simeq_0$ be defined as follows:

$$(s^S, s \vdash m \simeq_0 \langle 0, m \rangle) \text{ if } 0 \leq m < \#s^S \text{ and } s^S(m)(0) \neq \text{HEADER}$$

Let $\simeq$ be the term correspondence induced by $\simeq_0$.

Now, by definition of a term correspondence (Definition 11), we have that for all m, imm, and b^*,

$$(s^S, s \vdash \langle \text{FIXNUM } m \rangle \simeq \langle \text{FIXNUM } m \rangle),$$
$$(s^S, s \vdash imm \simeq imm), \text{ and}$$
$$(s^S, s \vdash b^* \simeq b^*)$$

Let $l^S < \#s^S$ be such that $s^S(l^S - 1) = \langle \text{HEADER } h^S \ p^S \ m^S \rangle$ for some h^S, p^S, m^S. Since s^S is an SBCM store, $s^S(l^S + i)$ is not a header for all $0 \leq i < \mho m^S$. Let $l = \langle 0, l^S \rangle$. By definition of the MSBCM loader, $s(l -_A 1) = \langle \text{HEADER } h^S \ p^S \ m^S \rangle$. Also, by definition of $\simeq_0$ above, we have that for all $0 \leq i < \mho m$, $(s^S, s \vdash (l^S + i) \simeq_0 (l +_A i))$. Then, by definition of a term correspondence (Definition 11), we have that

$$(s^S, s \vdash \langle \text{PTR } l^S \rangle \simeq \langle \text{PTR } l \rangle)$$

It is now straightforward to show from the definition of state correspondence (Definition 12) that $\simeq_0$ and $\simeq$ are witness to the state correspondence:

$$L_{sbcm}(\mathcal{P}) \cong L_{msbcm}(\mathcal{P})$$

□

2.6 Preserving State Correspondence

We prove that each action rule preserves state correspondence. That is, if Σ^S and Σ are corresponding SBCM and MSBCM states, then an SBCM action rule maps Σ^S to a state Σ'^S if and only if an MSBCM action rule maps Σ to a state Σ'; furthermore, the states Σ'^S and Σ' correspond.

The following lemma will be used extensively in the proof. The lemma states that if corresponding values are stored in corresponding (or new) locations of stores in corresponding states, then the resulting states also correspond.

Lemma 14 Let $\simeq_0$ be a location correspondence and $\simeq$ be the term correspondence induced by $\simeq_0$. Let $\Sigma^S = \langle t^S, n^S, v^S, a^S, u^S, k^S, s^S \rangle$ and $\Sigma = \langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4 \rangle$ be corresponding states of the SBCM and MSBCM respectively, i.e., $\Sigma^S \cong \Sigma$ with witnesses $\simeq_0$ and $\simeq$.

Let e^S, e be corresponding terms, i.e., $(s^S, s \vdash e^S \simeq e)$. Let l^S, l either be corresponding locations or new locations, i.e., either $(s^S, s \vdash l^S \simeq_0 l)$ or else $l^S = \#s^S$ and $l = \#s$. Let $s'^S = s^S[l^S \mapsto e^S]$ and $s' = s[l \mapsto e]$.

Let $\simeq'_0$ be a location correspondence such that for all l_1^S, l_1, $(s'^S, s' \vdash l_1^S \simeq'_0 l_1)$ holds if and only if either $(s^S, s \vdash l_1^S \simeq_0 l_1)$ holds or else $l_1^S = l^S$ and $l_1 = l$. Let $\simeq'$ be the term correspondence induced by $\simeq'_0$.

Then,

1. for all $cell^S, cell$,
   $(s^S, s \vdash cell^S \simeq cell) \Rightarrow (s'^S, s' \vdash cell^S \simeq' cell)$

2. for all $l^S \in ldom(s'^S)$ and $l \in ldom(s')$
   $(s'^S, s' \vdash l^S \simeq'_0 l) \Rightarrow (s'^S, s' \vdash s'^S(l^S) \simeq' s'(l))$

3. $\langle t^S, n^S, v^S, a^S, u^S, k^S, s'^S \rangle \cong \langle t, n, c, v, a, u, k, s', r_1, r_2, r_3, r_4 \rangle$ with witnesses $\simeq'_0$ and $\simeq'$.

Proof:

1. If $(s^S, s \vdash l^S \simeq_0 l)$, we have from definition 10 that $s^S(l^S)$ and $s(l)$ are not headers. By inspection, we see that the definition of a term correspondence relation does not depend on the contents of such locations. Thus modifying the contents of the locations l^S and l does not affect the term correspondence relation $\simeq$, hence the result.

Otherwise, l^S and l are new locations. Now, $\simeq$ and $\simeq'$ are induced (from the same set of monotonic closure rules) by $\simeq_0$ and $\simeq'_0$ respectively. But, by definition of $\simeq'_0$, for all l_1^S, l_1, $(s^S, s \vdash l_1^S \simeq_0 l_1)$ implies $(s'^S, s' \vdash l_1^S \simeq'_0 l_1)$. The result thus follows.

2. Let $l_1^S \in ldom(s'^S)$, $l_1 \in ldom(s')$, and $(s'^S, s' \vdash l_1^S \simeq'_0 l_1)$. The proof is by cases:

   Case $(l_1^S \neq l^S)$ **and** $(l_1 \neq l)$**:**

$(s'^S, s' \vdash l_1^S \simeq'_0 l_1)$	by assumption
$\Rightarrow (s^S, s \vdash l_1^S \simeq_0 l_1)$	by definition of $\simeq'_0$ and case condition
$\Rightarrow (s^S, s \vdash s^S(l_1^S) \simeq s(l_1))$	since $\Sigma^S \cong \Sigma$
$\Rightarrow (s'^S, s' \vdash s^S(l_1^S) \simeq' s(l_1))$	by part 1 of this lemma
$\Rightarrow (s'^S, s' \vdash s'^S(l_1^S) \simeq' s'(l_1))$	by case condition and definitions of s'^S and s'

   Case $(l_1^S = l^S)$ **and** $(l_1 = l)$**:**

$(s^S, s \vdash e^S \simeq e)$	by assumption
$\Rightarrow (s'^S, s' \vdash e^S \simeq' e)$	by part 1 of this lemma
$\Rightarrow (s'^S, s' \vdash s'^S(l^S) \simeq' s'(l))$	by definition of s'^S and s'
$\Rightarrow (s'^S, s' \vdash s'^S(l_1^S) \simeq' s'(l_1))$	by case condition

   There are no other cases by definition of $\simeq'_0$ (since $\simeq_0$ and $\simeq'_0$ are injective on locations and $(s^S, s \vdash l^S \simeq_0 l)$).

3. Follows trivially from Definition 12 and parts (1) and (2) of this lemma.

□

Lemma 15 *Let Σ^S and Σ be corresponding* SBCM *and* MSBCM *states. Then an* SBCM *action rule maps Σ^S to a state Σ'^S if and only if an* MSBCM *action rule maps Σ to a state Σ'; furthermore, the states Σ'^S and Σ' correspond.*

Proof: We prove this in the following subsections by considering each action rule in turn. The following rules have identical definition forms as in the SBCM(except for changes made to the spare registers r_1, r_2, r_3, and r_4): `return`, `call`, `jump-if-false`, `jump`, `literal`, `global`, `set-global!`,

`local`, `set-local!`, `push`, `unspecified`, `check-args=`, `check-args>=`, and `primitive-throw`. Their proof is trivial and is omitted from this report.

The proofs for the remaining four rules (`make-cont`, `make-closure`, `make-env`, and `make-rest-list`) are similar. We present the proof for `make-cont` as a representative proof.

Rule: make-cont $m_1\ m_2\ m_3$

Changes:

$$\begin{array}{rcl}
\Sigma'^S & = & \Sigma^S[a'^S = \langle\rangle] \\
 & & \quad [k'^S = \langle \textsc{ptr}\ \#s^S + 1\rangle] \\
 & & \quad [s'^S = s^S \rhd \langle \textsc{continuation}\ \langle t^S\ \langle \textsc{fixnum}\ m^S\rangle \\
 & & \qquad\qquad u^S\ k^S\rangle \frown a^S\rangle] \\
 & & \quad [n'^S = n^S + 4] \\
 & & \quad [r_1'^S = \langle \textsc{ptr}\ \#s^S + 1\rangle] \\
 & & \text{where } m^S = n^S + 4 + (m_1 \otimes m_2) \\
\Sigma' & = & \Sigma[a' = \langle\rangle] \\
 & & \quad [k' = \langle \textsc{ptr}\ \langle 1\ -\#s(1) - \#a - 3\rangle\rangle] \\
 & & \quad [s' = s +_1 \langle \textsc{continuation}\ \#f\ \langle t\ \langle \textsc{fixnum}\ m\rangle\ u\ k\rangle \frown a\rangle] \\
 & & \quad [n' = n + 4] \\
 & & \quad [r_1' = \langle \textsc{ptr}\ \langle 1\ -\#s(1) - \#a - 3\rangle\rangle] \\
 & & \text{where } m = n + 4 + (m_1 \otimes m_2)
\end{array}$$

Proof Obligations:

Let $\simeq_0$ and $\simeq$ be the location and term correspondences that are witnesses to the state correspondence $\Sigma^S \cong \Sigma$. Let $\simeq_0'$ be a new location correspondence such that for all locations l_1^S, l_1, $(s'^S, s' \vdash l_1^S \simeq_0' l_1)$ holds if either $(s^S, s \vdash l_1^S \simeq_0 l_1)$ holds, or if $l_1^S = \#s^S + i$ and $l_1 = \langle 1, -\#s(1) - \#a - 4) +_A i\rangle$ for some $1 \le i < \#a^S + 5$. Let $\simeq'$ be the term correspondence induced by $\simeq_0$ (by Definition 11). We show that $\simeq_0'$ and $\simeq'$ are witness to the state correspondence $\Sigma'^S \cong \Sigma'$.

Since $\Sigma^S \cong \Sigma$, we have that

$$\begin{array}{l}
(s^S, s \vdash t^S \simeq t) \\
m^S = m \\
(s^S, s \vdash u^S \simeq u) \\
(s^S, s \vdash k^S \simeq k) \\
(s^S, s \vdash a^S(i) \simeq a(i)) \text{ for all } 0 \le i < \#a
\end{array}$$

Now, since

$$\begin{array}{rcl} s'^S & = & s^S \rhd \langle \text{CONTINUATION } \#f\ \langle t^S\ \langle \text{FIXNUM } m^S\rangle\ u^S\ k^S\rangle \frown a^S\rangle \\ s' & = & s +_1 \langle \text{CONTINUATION } \#f\ \langle t\ \langle \text{FIXNUM } m\rangle\ u\ k\rangle \frown a\rangle \end{array}$$

we can use Lemma 14 in proving the obligations.

1. for all $l_1^S \in ldom(s'^S)$ and $l_1 \in ldom(s')$,
   $(s'^S, s' \vdash l_1^L \simeq' l_1) \Rightarrow (s'^S, s' \vdash s'^L(l_1^S) \simeq' s'(l_1))$
   This follows from Lemma 14.

2. $(s'^S, s' \vdash t^L \simeq' t)$ which follows from Lemma 14.

3. $n'^S = n'$ where $n'^S = n^S + 4$ and $n' = n + 4$; this holds since $n^S = n$ by the assumption that $\Sigma^S \cong \Sigma$.

4. $(s'^S, s' \vdash v^L \simeq' v)$ which follows from Lemma 14.

5. $\#a'^S = \#a'$ and $(\forall 0 \le i < \#a')(s'^S, s' \vdash a'^L(i) \simeq' a'(i))$ which holds trivially since $a'^S = a' = \langle\rangle$.

6. $(s'^S, s' \vdash u^L \simeq' u)$ which follows from Lemma 14.

7. $(s'^S, s' \vdash k'^L \simeq' k')$ where $k'^S = \langle \text{PTR } \#s^S + 1\rangle$ and $k' = \langle \text{PTR } \langle 1 - \#s(1) - \#a - 3\rangle\rangle$.
   Now $\mathcal{S}(k'^S, s'^S) = \langle \text{CONTINUATION } \#f\ \langle t^S\ \langle \text{FIXNUM } m^S\rangle\ u^S\ k^S\rangle \frown a^S\rangle$ and $\mathcal{S}(k', s') = \langle \text{CONTINUATION } \#f\ \langle t\ \langle \text{FIXNUM } m\rangle\ u\ k\rangle \frown a\rangle$ by Lemma 5. The result then follows from Lemma 14 since $\Sigma' \cong \Sigma$.

□

2.7 Correspondence of Final Answers

The following lemma asserts that the SBCM and MSBCM answer functions map corresponding states to equal natural numbers.

Lemma 16 *Let Σ_f^S and Σ_f be* SBCM *and* MSBCM *halt states respectively. If $\Sigma_f^S \cong \Sigma_f$, then $A_{sbcm}(\Sigma_f^S) = A_{msbcm}(\Sigma_f)$.*

Proof: Let v^S and v be the value registers of Σ_f^S and Σ_f respectively. Since $\Sigma_f^S \cong \Sigma_f$, we have by definition of state correspondence that v^S and v are related (i.e. $(s^S, s \vdash v^S \simeq v)$). By definition of term correspondence, if $v^S = \langle \text{FIXNUM } m \rangle$ then $v = \langle \text{FIXNUM } m \rangle$ and so $A_{sbcm}(\Sigma_f^S) = A_{msbcm}(\Sigma_f) = m$. If v^S is not of the form $\langle \text{FIXNUM } m \rangle$ then neither is v, and so $A_{sbcm}(\Sigma_f^S)$ and $A_{msbcm}(\Sigma_f)$ are both undefined.

□

2.8 Correspondence of Semantics

We can now prove the main theorem of this chapter.

Theorem 17 *If $\mathcal{P}$ is an* SBC *program, then its operational semantics as given by the* SBCM *yields the same answer as its operational semantics as given by the* MSBCM*. That is,*

$$\mathcal{O}^{sbc}_{sbcm}[\![\mathcal{P}]\!] = \mathcal{O}^{sbc}_{msbcm}[\![\mathcal{P}]\!]$$

Proof: Follows immediately from Lemmas 13, 15, and 16.

□

3 Garbage-collected SBC State Machine

In this chapter, we give Stored Byte Code (SBC) programs an (operational) semantics in terms of a deterministic state machine with concrete states. The state machine is called the *Garbage-collected Stored Byte Code Machine (*GSBCM*)*. We show that this operational semantics is equivalent to the operational semantics in terms of the MSBCM.

This chapter presents:

1. an operational semantics of SBC programs in terms of a state machine called the Garbage-collected Stored Byte Code Machine (GSBCM).

2. a correspondence relation that relates MSBCM states and GSBCM states;

3. a proof that the MSBCM loader and the GSBCM loader map SBC programs to initial states that are related by the above correspondence relation.

4. a proof that the MSBCM and GSBCM action rules preserve state correspondence. That is, if an MSBCM state is related to a GSBCM state, then either the MSBCM and GSBCM action rules are both undefined on the respective states, or else they map the respective related states to states that are also related.

   A crucial ingredient of this proof is a proof that if an MSBCM state is related to a GSBCM state, then the function *garbage-collect* maps the GSBCM state to a state that is also related to the MSBCM state.

5. a proof that the MSBCM and GSBCM answer functions map related states to equal numbers.

6. a proof that the operational semantics of SBC as given by the MSBCM is extensionally equal to the operational semantics of SBC as given by the GSBCM.

3.1 GSBC State Machine (GSBCM)

3.1.1 States

The states of the GSBCM are those of the MSBCM. The initial and halt states of the GSBCM are also those of the MSBCM.

3.1.2 Garbage Collection Algorithm

```
(define (pointer-to-old-space? p)
  (and (pointer? p)
       (let ((a (pointer->addr p)))
         (and (not (addr< a *old-low-heap-base*))
              (addr< a *old-high-heap-limit*)
              ))))

(define (gc-convert! ret p)
  (let ((h (stob-header p)))
    (cond ((pointer? h) (ret h))
          ((header? h)
           (nh-alloc! h (addr+ *hp* (+ (header-size-in-cells h)
                                       1)))
           (heap-set! (pointer->addr p) -1 (spare1-ref))
           (do ((src (pointer->addr p) (addr+ src 1))
                (dst (pointer->addr (spare1-ref)) (addr+ dst 1))
                (ret ret)
                )
               ((addr= dst *hp*) (ret (spare1-ref)))
             (heap-set! dst 0 (heap-ref src 0))))
          (else (error!
                 "gc-trace: invalid heap structure during gc"))
          )))

(define (gc-trace! ret p)
  (if (pointer? p)
      (gc-convert! ret p)
      (ret p)
      ))

(define *stack-scan* *old-stack-base*)
(define (gc-trace-stack!)
  (set! *stack-scan* (addr+ *old-stack-base* -1))
  (do-gc-trace-stack!))
(define (do-gc-trace-stack!)
  (if (addr< *stack-scan* *old-sp*)
      (ret8)
      (gc-trace! ret-gc-trace-stack! (stack-ref *stack-scan*))
      ))
(define (ret-gc-trace-stack! val)
  (stack-push! val)
  (set! *stack-scan* (addr+ *stack-scan* -1))
  (do-gc-trace-stack!))
```

```
(define *scan* heap-low)
(define (gc-scan-heap!)
  (if (addr< *scan* *hp*)
      (let ((desc (heap-ref *scan* 0)))
        (cond ((pointer-to-old-space? desc)
               (gc-convert! ret-gc-scan-heap! desc))
              ((and (header? desc)
                    (binarray-header-tag? (header-tag desc)))
               (set! *scan*
                     (addr+ *scan* (+ (header-size-in-cells desc)
                                      1)))
               (gc-scan-heap!))
              (else
               (set! *scan* (addr+ *scan* 1))
               (gc-scan-heap!))))
      (ret9)))
(define (ret-gc-scan-heap! val)
  (heap-set! *scan* 0 val)
  (set! *scan* (addr+ *scan* 1))
  (gc-scan-heap!))

(define *gc-ret-address* (lambda () 0))
(define (garbage-collect! ret)
  (set! *gc-ret-address* ret)
  (if noisy-vm
      (begin
        (write "[Garbage Collecting ..." (current-output-port))
        (force-output (current-output-port))))
  ;;; Flip

  (set! *old-low-heap-base* *low-heap-base*)
  (set! *old-hp* *hp*)
  (set! *old-sp* *sp*)
  (set! *old-stack-base* *stack-base*)
  (set! *old-low-heap-limit* *low-heap-limit*)
  (set! *old-high-heap-limit* *high-heap-limit*)

  (if (addr= *low-heap-base* heap-low)
      (begin
        (set! *low-heap-base* heap-mid)
        (set! *stack-base* heap-high)
        (set! *high-heap-limit* heap-high))
      (begin
        (set! *low-heap-base* heap-low)
```

```
        (set! *stack-base* heap-mid)
        (set! *high-heap-limit* heap-mid)))
  (set! *hp* *low-heap-base*)
  (set! *sp* *stack-base*)
  (set! *low-heap-limit*
        (addr+ *stack-base* (- stack-size-in-words)))
  (set! *scan* *hp*)
  (gc-trace! ret1 (template-ref))  )
(define (ret1 val)
  (template-set! val)
  (gc-trace! ret2 (value-ref)) )
(define (ret2 val)
  (value-set! val)
  (gc-trace! ret3 (env-ref)) )
(define (ret3 val)
  (env-set! val)
  (gc-trace! ret4 (cont-ref)) )
(define (ret4 val)
  (cont-set! val)
  (gc-trace! ret6 (spare2-ref)) )
(define (ret6 val)
  (spare2-set! val)
  (gc-trace! ret7 (symbol-table-ref)) )
(define (ret7 val)
  (symbol-table-set! val)
  (gc-trace-stack!))
(define (ret8)
  (gc-scan-heap!))
(define (ret9)
  (codevector-set! (template-stob-codevector (template-ref)))
  (let ((free (quotient (* (addr- *sp* *hp*) 100)
                        heap-space-size-in-words)))
    (if noisy-vm
        (begin
          (write " done; " (current-output-port))
          (write-int free (current-output-port))
          (write "% free]" (current-output-port))
          (newline (current-output-port))))
    (assert! (> free 10)
             "Only 10% free after garbage collection."))
  (*gc-ret-address*))
```

3.1.3 State Observers and Mutators

The GSBCM state observer and mutator functions are defined as the corresponding MSBCM state observer and mutator functions, except for three mutator functions which are defined as follows:

Let $\Sigma = \langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4 \rangle$ be a GSBCM state. Let

$$\Sigma' = \texttt{if } C \texttt{ then } \Sigma \texttt{ else } \textit{garbage-collect}(\Sigma)$$

where C is some unspecified boolean condition and the function *garbage-collect* is a partial function from GSBCM states to GSBCM states and is defined in Section 3.1.2. Then,

$$\begin{aligned}
\texttt{make-desc-stob}\ h\ p\ m\ \Sigma \quad &= \quad \Sigma'[s' = s \oplus_0 \langle h\ p\ m \rangle] \\
&\qquad [r_1' = \langle 0, \#s(0) \rangle] \\
&\qquad \text{if } h \in \textit{dhtag} \\
\texttt{make-desc-stob-from-stack}\ h\ p\ \Sigma & \\
&= \quad \Sigma'[s' = s +_1 \langle h\ p\ a \rangle] \\
&\qquad [a' = \langle\rangle] \\
&\qquad [r_1' = \langle 1, -\#s(1) - \#a + 1 \rangle] \\
&\qquad \text{if } h \in \textit{dhtag} \\
\texttt{make-byte-stob}\ h\ p\ m\ \Sigma \quad &= \quad \Sigma'[s' = s \oplus_0 \langle h\ p\ m \rangle] \\
&\qquad [r_1' = \langle 0, \#s(0) \rangle] \\
&\qquad \text{if } h \in \textit{bhtag}
\end{aligned}$$

3.1.4 Action Rules

In Section 2.2.8, we specified the action rules of the MSBCM as the composition of stored object manipulators, microcode functions, and other auxiliary functions. The action rules of the GSBCM are given identical definition forms. Note, however, that since two GSBCM mutator functions differ from the corresponding MSBCM functions, the extensions of the GSBCM action rules are not the same as the extensions of the corresponding MSBCM action rules.

3.2 SBC Operational Semantics

The GSBCM *Loader* L_{gsbcm} is defined to be a function that maps SBC programs to GSBCM initial states such that:

$$L_{gsbcm}(\langle s^{sbc}\ t^{sbc} \rangle) \stackrel{\text{def}}{=} L_{msbcm}(\langle s^{sbc}\ t^{sbc} \rangle)$$

The GSBCM *Answer* A_{gsbcm} is defined to be a partial function that maps GSBCM halt states Σ^f_{gsbcm} to natural numbers such that

$$A_{gsbcm}(\Sigma^f_{gsbcm}) \stackrel{\text{def}}{=} A_{msbcm}(\Sigma^f_{gsbcm})$$

Definition 18 *Let $\mathcal{P}$ be an* SBC *program. Then $L_{gsbcm}(\mathcal{P})$ is an initial state of the* GSBCM. *If $\mathcal{R}^*_{gsbcm}(L_{gsbcm}(\mathcal{P}))$ is defined, then the meaning of program $\mathcal{P}$ as given by the* SBC *operational semantics is*

$$\mathcal{O}_{gsbcm}[\![\mathcal{P}]\!] \stackrel{\text{def}}{=} A_{gsbcm}(\mathcal{R}^*_{gsbcm}(L_{gsbcm}(\mathcal{P})))$$

Otherwise $\mathcal{O}_{gsbcm}[\![\mathcal{P}]\!]$ is undefined.

3.3 State Correspondence Relation

We express the relation between MSBCM and GSBCM states via a state correspondence relation. Our approach is inspired by Wand and Oliva's storage relations [6].

If $\simeq$ is a four-place relation, we write $(s^M, s \vdash x^M \simeq x)$ to mean that (s^M, x^M, s, x) is in the relation $\simeq$.

Let s be an MSBCM (or GSBCM) store. Then

$$ldom(s) \stackrel{\text{def}}{=} \{\langle i, m\rangle \mid 0 \leq i < 4 \wedge m \in dom(s(i))\}$$

is the set of all locations l such that $s(l)$ is defined.

Definition 19 A *location correspondence* $\simeq_0 \subseteq (s^M \times l^M) \times (s \times l)$ is defined to be a relation such that for all s^M, s, the set $\{(l^M, l) \mid (s^M, s \vdash l^M \simeq_0 l)\}$ is a 1-to-1 map from a subset of $ldom(s^M)$ to a subset of $ldom(s)$. Further, if $(s^M, s \vdash l^M \simeq_0 l)$ holds, then $s^M(l^M)$ and $s(l)$ are not headers.

Definition 20 Let $\simeq_0 \subseteq (s^M \times l^M) \times (s \times l)$ be a location correspondence. A *term correspondence* $\simeq \subseteq (s^M \times cell^M) \times (s \times cell)$ induced by $\simeq_0$ is defined as follows:

$(s^M, s \vdash cell^M \simeq cell)$ holds if one of the following hold:

- 1. $cell^M = \langle \text{PTR } l^M \rangle$
  2. $cell = \langle \text{PTR } l \rangle$
  3. $s^M(l^M -_A 1) = \langle \text{HEADER } h^M\ p^M\ m^M \rangle$
  4. $s(l -_A 1) = \langle \text{HEADER } h\ p\ m \rangle$

5. $h^M = h$
6. $p^M = p$
7. $m^M = m$
8. for all $0 \leq i < \mho m$, $(s^M, s \vdash (l^M +_A i) \simeq_0 (l +_A i))$

- $cell^M = \langle \text{FIXNUM } m \rangle = cell$
- $cell^M = imm = cell$
- $cell^M = b^* = cell$

Definition 21 Let

$\Sigma_{msbcm} = \langle t^M, n^M, c^M, v^M, a^M, u^M, k^M, s^M, r_1^M, r_2^M, r_3^M, r_4^M \rangle$, and
$\Sigma_{gsbcm} = \langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4 \rangle$

be MSBCM and GSBCM states respectively. The *state correspondence relation* $\cong \subset state^M \times state$ is defined as follows:

$$\Sigma_{msbcm} \cong \Sigma_{gsbcm}$$

if there exists a location correspondence $\simeq_0$ and a term correspondence $\simeq$ induced by $\simeq_0$ such that

- for all $l^M \in ldom(s^M)$ and $l \in ldom(s)$,
  $(s^M, s \vdash l^M \simeq_0 l) \Rightarrow (s^M, s \vdash s^M(l^M) \simeq s(l))$
- $(s^M, s \vdash t^M \simeq t)$
- $n^M = n$
- $(s^M, s \vdash v^M \simeq v)$
- $\#a^M = \#a$ and $(\forall 0 \leq i < \#a)(s^M, s \vdash a^M(i) \simeq a(i))$
- $(s^M, s \vdash u^M \simeq u)$
- $(s^M, s \vdash k^M \simeq k)$
- $(s^M, s \vdash r_2^M \simeq r_2)$
- $r_3^M = r_3$
- $r_4^M = r_4$

Lemma 22 $\cong$ *is a transitive relation, i.e., if* $\Sigma \cong \Sigma'$ *and* $\Sigma' \cong \Sigma''$ *then* $\Sigma \cong \Sigma''$.

Proof: Follows trivially, since the location and term correspondence relations $\simeq_0$ and $\simeq$ are easily seen to be transitive.

□

3.4 Establishing State Correspondence

Let $d_u \stackrel{\text{def}}{=} \langle \text{IMMEDIATE UNSPECIFIED} \rangle$. Then, the MSBCM and GSBCM *loaders*, L_{msbcm} and L_{gsbcm}, are partial functions defined as follows:

$$\begin{array}{rcl} L_{msbcm}(\langle s^{sbc}\ t^{sbc} \rangle) & = & L_{gsbcm}(\langle s^{sbc}\ t^{sbc} \rangle) \\ & \stackrel{\text{def}}{=} & \langle t, 0, c, d_u, \langle\rangle, \langle \text{IMMEDIATE EMPTY-ENV} \rangle, \\ & & \quad \langle \text{IMMEDIATE HALT} \rangle, s, d_u, d_u, 0, d_u \rangle \\ & & \text{if } s = load(s^{sbc}) \text{ and } t = relocate\,(t^{sbc}) \\ & & \text{and } \mathcal{S}(t, s) = \langle \text{TEMPLATE } \#f\ c{::}o^* \rangle. \end{array}$$

Lemma 23 *The* MSBCM *Loader and the* GSBCM *Loader map* SBC *programs to initial states that are related by the state correspondence relation. That is, if* $\mathcal{P}$ *is an* SBC *program, then*

$$L_{msbcm}(\mathcal{P}) \cong L_{gsbcm}(\mathcal{P})$$

Proof: Let $L_{msbcm}(\mathcal{P}) = \langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4 \rangle = L_{gsbcm}(\mathcal{P})$ (since the two loaders are equal by definition). Let the location correspondence $\simeq_0$ be defined as follows:

$$(s, s \vdash \langle 0, m \rangle \simeq_0 \langle 0, m \rangle) \text{ if } 0 \leq m < \#s(0) \text{ and } s(\langle 0, m \rangle) \text{ is not a header}$$

Let $\simeq$ be the term correspondence induced by $\simeq_0$.

Now, by definition of a term correspondence (Definition 20), we have that for all m, imm, and b^*,

$$\begin{array}{l} (s, s \vdash \langle \text{FIXNUM } m \rangle \simeq \langle \text{FIXNUM } m \rangle), \\ (s, s \vdash imm \simeq imm), \text{ and} \\ (s, s \vdash b^* \simeq b^*) \end{array}$$

Let $l \in ldom(s)$ be such that $\mathcal{S}(\langle \text{PTR } l \rangle, s)$ is defined. Then, $s(l -_A 1) = \langle \text{HEADER } h\ p\ m \rangle$ for some h, p, m. Since s is an MSBCM store, $s(l +_A i)$ is

not a header for all $0 \leq i < \mho m$. Thus, by definition of $\simeq_0$ above, we have that for all $0 \leq i < \mho m$, $(s^M, s \vdash (l +_A i) \simeq_0 (l +_A i))$. Then, by definition of a term correspondence (Definition 20), we have that

$$(s, s \vdash \langle \text{PTR } l \rangle \simeq \langle \text{PTR } l \rangle)$$

It then follows immediately from the definition of state correspondence (Definition 21) that $\simeq_0$ and $\simeq$ are witness to the state correspondence:

$$L_{msbcm}(\mathcal{P}) \cong L_{gsbcm}(\mathcal{P})$$

□

3.5 Preserving State Correspondence

3.5.1 Garbage Collection

We define a binary predicate on GSBCM states and show that GSBCM states Σ and *garbage-collect*(Σ) are related. We use this to show that if an MSBCM state Σ^M is related to a GSBCM state Σ by the state correspondence relation, then Σ^M is also related to *garbage-collect*(Σ).

Let *gc-header-size-in-cells* be a function defined as follows:

$$\begin{aligned} &\textit{gc-header-size-in-cells}(d, s) = \mho m \\ &\quad \text{if } d = \langle \text{HEADER } h\ p\ m \rangle \\ &\quad \text{or } d = \langle \text{PTR } l \rangle \text{ and } s(l -_A 1) = \langle \text{HEADER } h\ p\ m \rangle \end{aligned}$$

Let *gc-header* be a predicate defined as follows:

- For $i \in \{0, 2\}$, if $s(\langle i, 0 \rangle)$ is defined then *gc-header*$(\langle i, 0 \rangle, s)$ holds;
- For $i \in \{1, 3\}$, if $s(\langle i, -\#s_i + 1 \rangle)$ is defined then *gc-header*$(\langle i, -\#s_i + 1 \rangle, s)$ holds;
- For all l, if

  $$\begin{aligned} &\textit{gc-header}(l, s), \\ &\textit{gc-header-size-in-cells}(s(l), s), \text{ and} \\ &s(l +_A (\textit{gc-header-size-in-cells}(s(l), s) + 1)) \end{aligned}$$

  are defined, then

  $$\textit{gc-header}(l +_A (\textit{gc-header-size-in-cells}(s(l), s) +_A 1), s)$$

  holds.

Definition 24 *A* GC *store is a store that satisfies the following conditions:*

1. *If* gc-header(l, s) *then either*
   (a) $s(l)$ *is a header, or*
   (b) $s(l) = \langle \text{PTR } l' \rangle$, $s(l' -_A 1)$ *is a header, and* l *has the form* $\langle 0, m \rangle$ *or* $\langle 1, m \rangle$ *while* l' *has the form* $\langle 2, n \rangle$.
2. *If* gc-header(l, s) *then for all* $1 \leq x < \mho m$
   - *If* $h \in bhtag$, *then* $s(l +_A x) = b^{bpw}$ *for some* b^{bpw}.
   - *If* $h \in bhtag$, *then* $s(l +_A x) =$ vd *for some* vd.

   where either $s(l) = \langle \text{HEADER } h\ p\ m \rangle$ *or* $s(l) = \langle \text{PTR } l' \rangle$ *and* $s(l') = \langle \text{HEADER } h\ p\ m \rangle$.
3. *For all* l, l', *if* $s(l) = \langle \text{PTR } l' \rangle$ *then* gc-header$(l' -_A 1, s)$ *holds.*

Definition 25 *A* GC state *is a sequence of the form*

$$\langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4 \rangle$$

that satisfies the following state invariants:

1. s *is a* GC *store*
2. *Template*(t, s)
3. *Offset*(n, t, s)
4. *Value*(v, s)
5. *Environment*(u, s)
6. *Continuation*(k, s)
7. $(\forall 0 \leq i < \#a)$ *Value*$(a(i), s)$

Lemma 26 *Every* GSBCM *store is a* GC *store, and every* GSBCM *state is a* GC *state.*

Proof: Follows trivially from the definitions, since the invariants satisfied by GC stores are weaker than those satisfied by GSBCM stores.

□

Definition 27 A GC *location correspondence* $\simeq_0 \subseteq (s \times l) \times (s \times l)$ is defined to be a relation such that for all s, s', the set $\{(l, l') \mid (s^M, s \vdash l \simeq_0 l')\}$ is a 1-to-1 map from a subset of $ldom(s)$ to a subset of $ldom(s')$. Further, if $(s^M, s \vdash l \simeq_0 l')$ holds, then $s(l)$ and $s'(l')$ are not headers.

Definition 28 Let $\simeq_0^{gc} \subseteq (s \times l) \times (s \times l)$ be a GC location correspondence. A GC *term correspondence* $\simeq^{gc} \subseteq (s \times cell) \times (s \times cell)$ induced by $\simeq_0$ is defined as follows:

$(s, s' \vdash cell_1 \simeq^{gc} cell_2)$ holds if one of the following hold:

- 1. $cell_1 = \langle \text{PTR } l_1 \rangle$
  2. $cell_2 = \langle \text{PTR } l_2 \rangle$
  3. $(l_1 = l') \wedge (s(l' -_A 1) = \langle \text{HEADER } h\ p\ m \rangle)$ or $s(l_1 -_A 1) = \langle \text{PTR } l' \rangle \wedge (s(l' -_A 1) = \langle \text{HEADER } h\ p\ m \rangle)$
  4. $(l_2 = l') \wedge (s(l' -_A 1) = \langle \text{HEADER } h\ p\ m \rangle)$ or $s(l_2 -_A 1) = \langle \text{PTR } l' \rangle \wedge (s(l' -_A 1) = \langle \text{HEADER } h\ p\ m \rangle)$
  5. $h = h$
  6. $p = p$
  7. $m = m$
  8. for all $0 \leq i < \mho m$, $(s, s' \vdash (l' +_A i) \simeq_0^{gc} (l' +_A i))$
- $cell_1 = \langle \text{FIXNUM } m \rangle = cell_2$
- $cell_1 = imm = cell_2$
- $cell_1 = b^* = cell_2$

Definition 29 Let

$$\Sigma = \langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4 \rangle, \text{ and}$$
$$\Sigma' = \langle t', n', c', v', a', u', k', s', r_1', r_2', r_3', r_4' \rangle$$

be GSBCM states. The GC *state correspondence relation* $\cong \subset state \times state$ is defined as follows:

$$\Sigma \cong^{gc} \Sigma'$$

if there exists a GC location correspondence $\simeq_0^{gc}$ and a GC term correspondence $\simeq^{gc}$ induced by $\simeq_0^{gc}$ such that

- for all $l \in ldom(s)$ and $l' \in ldom(s')$,
  $(s, s' \vdash l \simeq_0^{gc} l') \Rightarrow (s, s' \vdash s(l) \simeq^{gc} s'(l'))$
- $(s, s' \vdash t \simeq^{gc} t')$
- $n = n'$
- $(s, s' \vdash v \simeq^{gc} v')$
- $\#a = \#a'$ and $(\forall 0 \leq i < \#a)(s, s' \vdash a(i) \simeq^{gc} a'(i))$
- $(s, s' \vdash u \simeq^{gc} u')$
- $(s, s' \vdash k \simeq^{gc} k')$
- $(s, s' \vdash r_2 \simeq^{gc} r_2')$
- $r_3 = r_3'$
- $r_4 = r_4'$

Let $\Sigma = \langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4 \rangle$ be a GSBCM state. Let $\langle \text{PTR } l \rangle$ be such that *gc-header*$(l -_A 1, s)$ holds. Then,

$$
\begin{array}{l}
\textit{gc-convert}(\langle \text{PTR } l \rangle, \Sigma) = \\
\quad \begin{array}{ll}
\langle \langle \text{PTR } l' \rangle \ s \ r_1 \rangle & \text{if } s(l -_A 1) = \langle \text{PTR } l' \rangle \\
\langle \langle \text{PTR } l' \rangle \ s'' \ r_1' \rangle & \text{if } s(l -_A 1) = \langle \text{HEADER } h \ p \ m \rangle \\
\quad \text{where } l' = \langle 2, \#s(2) + 1 \rangle & \\
\quad \text{and } s' = s +_2 \mathcal{S}(\langle \text{PTR } l \rangle, s) & \\
\quad \text{and } s'' = s'[l -_A 1 \mapsto \langle \text{PTR } l' \rangle] & \\
\quad \text{and } r_1' = \langle \text{PTR } l' \rangle &
\end{array}
\end{array}
$$

$$
\begin{array}{lcll}
\textit{gc-trace}(d, \Sigma) & = & \textit{gc-convert}(d, \Sigma) & \text{if } d \text{ is a pointer} \\
 & & d & \text{otherwise}
\end{array}
$$

Lemma 30 *Let* $\Sigma = \langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4 \rangle$ *be a* GSBCM *state. Let* gc-trace$(d, \Sigma) = \langle d' \ s' \ r_1' \rangle$. *Then there exists a* GC *location correspondence* $\simeq_0^{gc}$ *such that, if* $\simeq^{gc}$ *is the* GC *term correspondence induced by* $\simeq_0^{gc}$, *then*

1. s' *is a* GC *store.*
2. $(s', s \vdash d' \simeq^{gc} d)$
3. $\Sigma \cong^{gc} \langle t, n, c, v, a, u, k, s', r_1', r_2, r_3, r_4 \rangle$ *with* $\simeq_0^{gc}$ *and* $\simeq^{gc}$ *as witnesses.*

4. *If* $d' = \langle \text{PTR } l' \rangle$ *then* $l' = \langle 2, m' \rangle$ *and* $s'(l' -_A 1) = \langle \text{HEADER } h\ p\ m \rangle$ *for some* h, p, m.

Proof: The proof is by cases.
Case $s(l -_A 1) = \langle \text{PTR } l' \rangle$
Let $\simeq_0^{gc}$ be defined such that for all $l \in ldoms$, $(s, s \vdash l \simeq_0^{gc} l)$ if $\neg gc\text{-}header(l, s)$.
Let $\simeq^{gc}$ be the term correspondence induced by $\simeq_0$.

1. s is a GC store by assumption.

2. $s(l' -_A 1) = \langle \text{HEADER } h\ p\ m \rangle$ since $gc\text{-}header(l -_A 1, s)$ and $s(l -_A 1) = \langle \text{PTR } l' \rangle$. $s(l -_A 1) = \langle \text{PTR } l' \rangle$ by case condition, and $s(l' -_A 1) = \langle \text{HEADER } h\ p\ m \rangle$ (as above). Thus, by definition of term correspondence,
$$(s, s \vdash \langle \text{PTR } l \rangle \simeq^{gc} \langle \text{PTR } l' \rangle).$$

3. $\Sigma \cong^{gc} \Sigma$ holds trivially.

4. $s(l' -_A 1) = \langle \text{HEADER } h\ p\ m \rangle$ as in (2), and $l' = \langle 2, m' \rangle$ since s is a GC store.

Case $s(l -_A 1) = \langle \text{HEADER } h\ p\ m \rangle$
Let $\simeq_0^{gc}$ be defined such that

- for all $l'' \in ldom(s)$, if $l'' \neq l +_A i$ for $0 \leq i < \mho m$ and if $\neg gc\text{-}header(l'', s)$, then $(s, s' \vdash l'' \simeq_0^{gc} l'')$.
- $(s, s' \vdash l +_A i \simeq_0^{gc} l' +_A i)$ for all $0 \leq i < \mho m$.

This is a one-to-one map since $l' = \langle 2, \#s(2) + 1 \rangle$ by definition, and hence $l' +_A i \notin ldom(s)$ for $0 \leq i < \mho m$. Let $\simeq^{gc}$ be the GC term correspondence induced by $\simeq_0^{gc}$.

1. Since s is a GC store, so is $s +_2 \mathcal{S}(\langle \text{PTR } l \rangle, s)$, and hence so is $s'' = (s +_2 \mathcal{S}(\langle \text{PTR } l \rangle, s))[l -_A 1 \mapsto \langle \text{PTR } l' \rangle]$.

2. $s(l -_A 1) = \langle \text{HEADER } h\ p\ m \rangle$ by case condition. $s'(l' -_A 1) = \langle \text{HEADER } h\ p\ m \rangle$ by definitions of s' and l'. For all $0 \leq i < \mho m$, $(s, s' \vdash l +_A i \simeq_0^{gc} l' +_A i)$ holds by definition of $\simeq_0^{gc}$. Thus, by definition of a term correspondence,
$$(s, s \vdash \langle \text{PTR } l \rangle \simeq^{gc} \langle \text{PTR } l' \rangle).$$

3. The registers of the state are unchanged except for r_1, but the definition of GC state correspondence does not depend on the value of r_1. Hence the only condition to verify is:

$$\begin{array}{l}\text{for all } l_1 \in \mathit{ldom}(s) \text{ and } l_2 \in \mathit{ldom}(s'),\\ \quad (s, s' \vdash l_1 \simeq_0^{gc} l_2) \Rightarrow (s, s' \vdash s(l_1) \simeq^{gc} s'(l_2))\end{array}$$

   Let $(s, s' \vdash l_1 \simeq_0^{gc} l_2)$. If $l_1 \neq l +_A i$ for $0 \leq i < \mho m$, then $l_1 = l_2$ by definition of $\simeq_0^{gc}$. Further, $\neg \mathit{gc\text{-}header}(l_1, s)$ and so $l_1 \neq l -_A 1$. Thus, $s'(l_2) = s'(l_1) = s(l_1)$. The result then follows since $(s, s' \vdash s(l_1) \simeq^{gc} s(l_1))$ holds.

   Otherwise, $l_1 = l +_A i$ for some $0 \leq i < \mho m$. Then, $l_2 = l' +_A i$ by definition of $\simeq_0^{gc}$. Now $s'(l_2) = s'(l' +_A i) = s(l +_A i)$ by definition of s'. The result then follows since $(s, s' \vdash s(l +_A i) \simeq^{gc} s(l +_A i))$ holds.

4. $s'(l' -_A 1)$ is a header by definition of s' and $l' = \langle 2, m \rangle$ by definition.

□

Lemma 31 *Let* $\Sigma = \langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4 \rangle$ *be a* GSBCM *state. Let* $\text{gc-trace}(t, \Sigma) = \langle t'\ s'\ r_1' \rangle$. *Let* $\mathcal{S}(t', s') = \langle \text{TEMPLATE}\ \#f\ c' :: o^* \rangle$. *Then* $\Sigma \cong \langle t', n, c', v, a, u, k, s', r_1', r_2, r_3, r_4 \rangle$.

Proof: By Lemma 30, $\Sigma \cong \langle t, n, c, v, a, u, k, s', r_1', r_2, r_3, r_4 \rangle$ and $(s', s \vdash t' \simeq t)$. Thus, by definition of state correspondence,

$$\langle t, n, c, v, a, u, k, s', r_1', r_2, r_3, r_4 \rangle \cong \langle t', n, c', v, a, u, k, s', r_1', r_2, r_3, r_4 \rangle$$

The result then holds by transitivity of state correspondence (Lemma 22).

□

Results analogous to Lemma 31 can be proved for v,u,k, and r_2.

Lemma 32 *Let* $\Sigma = \langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4 \rangle$ *be a* GSBCM *state. Let* $\text{gc-trace}(a(i), \Sigma) = \langle d'\ s'\ r_1' \rangle$. *Let* $a' = a[i \mapsto d']$. *Then*

$$\Sigma \cong \langle t, n, c, v, a', u, k, s', r_1', r_2, r_3, r_4 \rangle.$$

Proof: By Lemma 30, $\Sigma \cong \langle t, n, c, v, a, u, k, s', r_1', r_2, r_3, r_4\rangle$ and $(s, s' \vdash a(i) \simeq^{gc} d')$. Thus, for all $0 \leq i < \#a$, $(s, s' \vdash a(i) \simeq^{gc} a'(i))$ Thus, by definition of state correspondence,

$$\langle t, n, c, v, a, u, k, s', r_1', r_2, r_3, r_4\rangle \cong \langle t, n, c, v, a', u, k, s', r_1', r_2, r_3, r_4\rangle.$$

The result then holds by transitivity of state correspondence (Lemma 22).

□

Lemma 33 *Let* $\Sigma = \langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4\rangle$ *be a* GSBCM *state. Let* gc-trace$(s(\langle 2, m\rangle), \Sigma) = \langle d'\ s'\ r_1'\rangle$. *Then* $\Sigma \cong \langle t, n, c, v, a, u, k, s'[\langle 2, m\rangle \mapsto d'], r_1', r_2, r_3, r_4\rangle$.

Proof: By Lemma 30, $\Sigma \cong \langle t, n, c, v, a, u, k, s', r_1', r_2, r_3, r_4\rangle$ and $(s', s \vdash d' \simeq s(\langle 2, m\rangle))$. Since $s(\langle 2, m\rangle) = s'(\langle 2, m\rangle)$, we have that $(s', s \vdash d' \simeq s'(\langle 2, m\rangle))$.

Let $\simeq_0$ be defined as follows:

$$(s', s'[\langle 2, m\rangle \mapsto d'] \vdash l \simeq_0 l) \text{ if } l \in \mathit{ldom}(s') \text{ and } \neg \mathit{gc\text{-}header}(l, s')$$

Let $\simeq$ be the term correspondence induced by $\simeq_0$. Then it follows from the definition of a state correspondence that $\simeq_0$ and $\simeq$ are witnesses to the state correspondence

$$\langle t, n, c, v, a, u, k, s', r_1', r_2, r_3, r_4\rangle \cong \langle t, n, c, v, a, u, k, s'[\langle 2, m\rangle \mapsto d'], r_1', r_2, r_3, r_4\rangle$$

The result then holds by transitivity of state correspondence (Lemma 22).

□

Lemma 34 *Let* $\Sigma = \langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4\rangle$ *be an* GSBCM *state. Let* $\Sigma' =$ `if` C `then` Σ `else` garbage-collect(Σ). *Then* $\Sigma \cong \Sigma'$.

Proof: Let Σ be a GSBCM state. Then Σ is also a GC state. Let

$$\begin{array}{l} \mathit{gc\text{-}trace}(t, \Sigma) = \langle t'\ s^1\ r_1^1 \rangle \\ \Sigma^1 = \langle t', n, c, v, a, u, k, s^1, r_1^1, r_2, r_3, r_4 \rangle \end{array}$$

$$\begin{array}{l} \mathit{gc\text{-}trace}(v, \Sigma^1) = \langle v'\ s^2\ r_1^2 \rangle \\ \Sigma^2 = \langle t', n, c, v', a, u, k, s^2, r_1^2, r_2, r_3, r_4 \rangle \end{array}$$

$$\begin{array}{l} \mathit{gc\text{-}trace}(u, \Sigma^2) = \langle u'\ s^3\ r_1^3 \rangle \\ \Sigma^3 = \langle t', n, c, v', a, u', k, s^3, r_1^3, r_2, r_3, r_4 \rangle \end{array}$$

$$\begin{array}{l} \mathit{gc\text{-}trace}(k, \Sigma^3) = \langle k'\ s^4\ r_1^4 \rangle \\ \Sigma^4 = \langle t', n, c, v', a, u', k', s^4, r_1^4, r_2, r_3, r_4 \rangle \end{array}$$

$$\begin{array}{l} \mathit{gc\text{-}trace}(r_2, \Sigma^4) = \langle r_2'\ s^5\ r_1^5 \rangle \\ \Sigma^5 = \langle t', n, c, v', a, u', k', s^5, r_1^5, r_2', r_3, r_4 \rangle \end{array}$$

$$\begin{array}{l} \mathit{gc\text{-}trace\text{-}stack}(a, \Sigma^5) = \langle a'\ s^6\ r_1^6 \rangle \\ \Sigma^6 = \langle t', n, c, v', a', u', k', s^6, r_1^6, r_2', r_3, r_4 \rangle \end{array}$$

$$\begin{array}{l} \mathit{gc\text{-}scan\text{-}heap}(\Sigma^6) = \langle s^7\ r_1^7 \rangle \\ \Sigma^7 = \langle t', n, c, v', a', u', k', s^7, r_1^7, r_2', r_3, r_4 \rangle \end{array}$$

By Lemmas 31 (and analogous results for other registers), 32, 33, and the transitivity of $\cong^{gc}$, we have that

$$\Sigma \cong^{gc} \Sigma^7$$

Further, by Lemma 30(4), we have that none of the registers, stack, or cells in $s^7(2)$ contain pointers of the form $\langle \text{PTR}\ \langle i, m \rangle \rangle$ for $i \in \{0, 1, 3\}$. Thus, if $s^8 = \langle \langle\rangle\ \langle\rangle\ s^7(2)\ \langle\rangle \rangle$ and $\Sigma^8 = \Sigma^7[s' = s^8]$, then

$$\Sigma \cong^{gc} \Sigma^8$$

Let $\Sigma^9 = \mathit{flip}(\Sigma^8)$. Then,

$$\Sigma \cong^{gc} \Sigma^9$$

Let $\Sigma^{10} =$ `(codevector-set!` Σ^9 `(template-stob-codevector (template-ref` Σ^9 `)` Σ^9 `)` Σ^9 `)`.

That is, $\Sigma^{10} = \Sigma^9[c' = c_1]$ where $\mathcal{S}(t^9, s^9) = \langle \text{TEMPLATE}\ \#f\ c_1 :: o^* \rangle$ for some o^*. Since s^{10} is a GC store and all locations l in Σ^{10} have the form $\langle 0, m \rangle$, it follows that s^{10} is a GSBCM store. Thus, Σ^{10} is a GSBCM state.

Now *garbage-collect*$(\Sigma) = \Sigma^{10}$ by definition. Thus, since $\Sigma \cong^{gc} \Sigma$ and $\Sigma \cong^{gc} \Sigma^{10}$, we have that

$$\Sigma \cong^{gc} \texttt{if } C \texttt{ then } \Sigma \texttt{ else } \textit{garbage-collect}(\Sigma)$$

□

Lemma 35 *Let Σ_{msbcm} be a* MSBCM*state, and let Σ_{gsbcm} and Σ'_{gsbcm} be* GSBCM*states. If $\Sigma_{msbcm} \cong \Sigma_{gsbcm}$ and $\Sigma_{gsbcm} \cong^{gc} \Sigma'_{gsbcm}$, then $\Sigma_{msbcm} \cong \Sigma'_{gsbcm}$*

Proof: Let $\simeq_0$ and $\simeq$ be witnesses to the state correspondence $\Sigma_{msbcm} \cong \Sigma_{gsbcm}$. Let $\simeq_0^{gc}$ and $\simeq^{gc}$ be witnesses to the GC state correspondence $\Sigma_{gsbcm} \cong^{gc} \Sigma'_{gsbcm}$.

Let $(s, s' \vdash \mathit{cell} \simeq^{gc} \mathit{cell}')$ where $\mathit{cell} = \langle \text{PTR } l \rangle$ and $\mathit{cell}' = \langle \text{PTR } l' \rangle$. Then, since Σ_{gsbcm} and Σ'_{gsbcm} are GSBCMstates, it follows from the definition of GSBCM states that $s(l -_A 1)$ and $s'(l' -_A 1)$ are headers. Thus the GC term correspondence $\simeq^{gc}$ is a term correspondence $\simeq$. Similary, the GC state correspondence $\Sigma_{gsbcm} \cong^{gc} \Sigma'_{gsbcm}$ is also a state state correspondence $\Sigma_{gsbcm} \cong \Sigma'_{gsbcm}$. The result then follows by transitivity of $\cong$.

□

3.5.2 Microcode Functions

Lemma 36 Let $\simeq_0$ be a location correspondence and $\simeq$ be the term correspondence induced by $\simeq_0$. Let

$$\Sigma^M = \langle t^M, n^M, c^M, v^M, a^M, u^M, k^M, s^M, r_1^M, r_2^M, r_3^M, r_4^M \rangle, \text{ and}$$
$$\Sigma = \langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4 \rangle$$

be such that $\Sigma^M \cong \Sigma$ with witnesses $\simeq_0$ and $\simeq$. Let $d^M = \langle \text{PTR } l^M \rangle$, $d = \langle \text{PTR } l \rangle$, and $(s^M, s \vdash d^M \simeq d)$. Then,

1. $(s^M, s \vdash \texttt{template-ref}(\Sigma^M) \simeq \texttt{template-ref}(\Sigma))$
2. $(s^M, s \vdash \texttt{codevector-ref}(\Sigma^M) \simeq \texttt{codevector-ref}(\Sigma))$
3. $\texttt{offset-ref}(\Sigma^M) = \texttt{offset-ref}(\Sigma)$

4. $(s^M, s \vdash \texttt{value-ref}(\Sigma^M) \simeq \texttt{value-ref}(\Sigma))$
5. $(s^M, s \vdash \texttt{env-ref}(\Sigma^M) \simeq \texttt{env-ref}(\Sigma))$
6. $(s^M, s \vdash \texttt{cont-ref}(\Sigma^M) \simeq \texttt{cont-ref}(\Sigma))$
7. $(s^M, s \vdash \texttt{spare2-ref}(\Sigma^M) \simeq \texttt{spare2-ref}(\Sigma))$
8. $(s^M, s \vdash \texttt{spare3-ref}(\Sigma^M) \simeq \texttt{spare3-ref}(\Sigma))$
9. $(s^M, s \vdash \texttt{stack-top}(\Sigma^M) \simeq \texttt{stack-top}(\Sigma))$
10. $\texttt{stack-empty?}(\Sigma^M) = \texttt{stack-empty?}(\Sigma)$
11. $\texttt{stack-length}(\Sigma^M) = \texttt{stack-length}(\Sigma)$
12. $(s^M, s \vdash \texttt{stob-desc-ref}(d^M, i, \Sigma^M) \simeq \texttt{stob-desc-ref}(d, i, \Sigma))$
13. $(s^M, s \vdash \texttt{stob-byte-ref}(d^M, i, \Sigma^M) \simeq \texttt{stob-byte-ref}(d, i, \Sigma))$
14. $\texttt{stob-tag}(d^M, \Sigma^M) = \texttt{stob-tag}(d, \Sigma)$
15. $\texttt{stob-mutable?}(d^M, \Sigma^M) = \texttt{stob-mutable?}(d, \Sigma)$
16. $\texttt{stob-size-in-bytes}(d^M, \Sigma^M) = \texttt{stob-size-in-bytes}(d, \Sigma)$
17. $\texttt{stob-size-in-cells}(d^M, \Sigma^M) = \texttt{stob-size-in-cells}(d, \Sigma)$

Proof:
Cases 1–8: By definition, $\texttt{template-ref}(\Sigma^M) = t^M$ and $\texttt{template-ref}(\Sigma) = t$. Since $\Sigma^M \cong \Sigma$, we have by definition that $(s^M, s \vdash t^M \simeq t)$ as desired. Cases 2–8 are proved similarly.

Cases 9–11: Since $\Sigma^M \cong \Sigma$, we have by definition that $\#a^M = \#a$ and $(s^M, s \vdash a^M(i) \simeq a(i))$ for all $0 \leq i < \#a$. Now, $\texttt{stack-top}(\Sigma^M) = a^M(0)$ and $\texttt{stack-top}(\Sigma) = a(0)$. Hence,

$(s^M, s \vdash \texttt{stack-top}(\Sigma^M) \simeq \texttt{stack-top}(\Sigma))$
$\texttt{stack-empty?}(\Sigma^M) = (\#a^M = 0) = (\#a = 0) = \texttt{stack-empty?}(\Sigma)$
$\texttt{stack-length}(\Sigma^M) = \#a^M = \#a = \texttt{stack-length}(\Sigma)$

Cases 12–16: By assumption, $d^M = \langle \textsc{ptr}\ l^M \rangle$ and $d = \langle \textsc{ptr}\ l \rangle$. Let $s^M(l^M -_A 1) = \langle \textsc{header}\ h^M\ p^M\ m^M \rangle$ and $s(l -_A 1) = \langle \textsc{header}\ h\ p\ m \rangle$. Since $\Sigma^M \cong \Sigma$ and $(s^M, s \vdash d^M \simeq d)$, we have by definition that $h^M = h$,

$p^M = p$, $m^M = m$, and $(s^M, s \vdash (l^M +_A i) \simeq_0 (l +_A i))$ for all $0 \leq i < \mho m$. Hence, by definition of state correspondence, $(s^M, s \vdash s^M(l^M +_A i) \simeq s(l +_A i))$ for all $0 \leq i < \mho m$.

Since

$$\texttt{stob-desc-ref}(d^M, i, \Sigma^M) = s^M(l^M +_A i), \text{ and}$$
$$\texttt{stob-desc-ref}(d, i, \Sigma) = s(l +_A i)$$

we have that

$$(s^M, s \vdash \texttt{stob-desc-ref}(d^M, i, \Sigma^M) \simeq \texttt{stob-desc-ref}(d, i, \Sigma)).$$

Similarly, $(s^M, s \vdash \texttt{stob-byte-ref}(d^M, i, \Sigma^M) \simeq \texttt{stob-byte-ref}(d, i, \Sigma))$. Finally,

$$\texttt{stob-tag}(d^M, \Sigma^M) = h^M = h = \texttt{stob-tag}(d, \Sigma)$$
$$\texttt{stob-mutable?}(d^M, \Sigma^M) = p^M = p = \texttt{stob-mutable?}(d, \Sigma)$$
$$\texttt{stob-size-in-bytes}(d^M, \Sigma^M) = m^M = m = \texttt{stob-size-in-bytes}(d, \Sigma)$$
$$\texttt{stob-size-in-cells}(d^M, \Sigma^M) = \mho m^M = \mho m$$
$$= \texttt{stob-size-in-cells}(d, \Sigma)$$

□

Lemma 37 Let $\simeq_0$ be a location correspondence and $\simeq$ be the term correspondence induced by $\simeq_0$. Let $l^M \in ldom(s^M)$, $l \in ldom(s)$, $(s^M, s \vdash l^M \simeq_0 l)$, and $(s^M, s \vdash cell^M \simeq cell)$ for some $l^M, l, cell^M, cell$ such that $s^M(l^M)$ and $s(l)$ are not headers. Let $s'^M = s^M[l^M \mapsto cell^M]$ and $s' = s[l \mapsto cell]$.

Let $\simeq_0'$ be a location correspondence such that $(s'^M, s' \vdash l^M \simeq_0' l)$ holds if and only if $(s^M, s \vdash l^M \simeq_0 l)$ holds. Note that this is a valid definition since $ldom(s'^M) = ldom(s^M)$ and $ldom(s') = ldom(s)$. Let $\simeq'$ be the term correspondence induced by $\simeq_0'$.

Then,

$$(\forall cell^M, cell)(s^M, s \vdash cell^M \simeq cell) \Rightarrow (s'^M, s' \vdash cell^M \simeq' cell)$$

Proof: Let $(s^M, s \vdash cell^M \simeq cell)$. We proceed by cases:

Case $cell^M = \langle \text{PTR } l'^M \rangle$ **and** $cell = \langle \text{PTR } l' \rangle$**:**

By definition of term correspondence, the following hold:

1. $s^M(l'^M -_A 1) = \langle \text{HEADER } h^M \; p^M \; m^M \rangle$

2. $s(l' -_A 1) = \langle \text{HEADER } h\ p\ m \rangle$
3. $h^M = h$
4. $p^M = p$
5. $m^M = m$
6. for all $0 \leq i < \mho m$, $(s^M, s \vdash (l'^M +_A i) \simeq_0 (l' +_A i))$

Now $s^M(l'^M -_A 1)$ is a header while $s^M(l^M)$ is not (by assumption) and hence $l'^M -_A 1 \neq l^M$. Similarly, $l' -_A 1 \neq l$. Thus,

1. $s'^M(l'^M -_A 1) = s^M(l'^M -_A 1) = \langle \text{HEADER } h^M\ p^M\ m^M \rangle$
2. $s'(l' -_A 1) = s(l' -_A 1) = \langle \text{HEADER } h\ p\ m \rangle$
3. $h^M = h$
4. $p^M = p$
5. $m^M = m$
6. By definition of $\simeq'_0$, for all l^M, l, $(s^M, s \vdash l^M \simeq_0 l)$ holds if and only if $(s'^M, s' \vdash l^M \simeq'_0 l)$ holds. We thus have that for all $0 \leq i < \mho m$, $(s'^M, s' \vdash (l'^M +_A i) \simeq'_0 (l' +_A i))$.

Thus, by definition of term correspondence, $(s'^M, s' \vdash cell^M \simeq' cell)$ holds.

Case $cell^M = \langle \text{FIXNUM } m \rangle$ **and** $cell = \langle \text{FIXNUM } m \rangle$**:**
By definition of term correspondence, $(s'^M, s' \vdash cell^M \simeq' cell)$ holds.

Case $cell^M = imm$ **and** $cell = imm$**:**
By definition of term correspondence, $(s'^M, s' \vdash cell^M \simeq' cell)$ holds.

Case $cell^M = b^*$ **and** $cell = b^*$**:**
By definition of term correspondence, $(s'^M, s' \vdash cell^M \simeq' cell)$ holds.

□

Lemma 38 Let $\simeq_0$ be a location correspondence and $\simeq$ be the term correspondence induced by $\simeq_0$. Let $l^M \notin ldom(s^M)$, $l \notin ldom(s)$, and $(s^M, s \vdash cell^M \simeq cell)$ for some $cell^M, cell$. Let $s'^M = s^M[l^M \mapsto cell^M]$ and $s' = s[l \mapsto cell]$.

Let $\simeq'_0$ be a location correspondence such that, for all l_1^M, l_1, $(s'^M, s' \vdash l_1^M \simeq'_0 l_1)$ holds if and only if either $(s^M, s \vdash l_1^M \simeq_0 l_1)$ holds or $l_1^M = l^M$ and

$l_1 = l$. Note that this is a valid definition since $ldom(s'^M) = ldom(s^M) \cup \{l^M\}$ and $ldom(s') = ldom(s) \cup \{l\}$. Let $\simeq'$ be the term correspondence induced by $\simeq'_0$.

Then,

$$(\forall cell^M, cell)(s^M, s \vdash cell^M \simeq cell) \Rightarrow (s'^M, s' \vdash cell^M \simeq' cell)$$

Proof: Let $(s^M, s \vdash cell^M \simeq cell)$. We proceed by cases:

Case $cell^M = \langle \text{PTR } l'^M \rangle$ **and** $cell = \langle \text{PTR } l' \rangle$**:**

By definition of term correspondence, the following hold:

1. $s^M(l'^M -_A 1) = \langle \text{HEADER } h^M\ p^M\ m^M \rangle$
2. $s(l' -_A 1) = \langle \text{HEADER } h\ p\ m \rangle$
3. $h^M = h$
4. $p^M = p$
5. $m^M = m$
6. for all $0 \leq i < \mho m$, $(s^M, s \vdash (l'^M +_A i) \simeq_0 (l' +_A i))$

Now $s^M(l'^M -_A 1)$ is defined while $s^M(l^M)$ is not (by assumption) and hence $l'^M -_A 1 \neq l^M$. Similarly, $l' -_A 1 \neq l$. Thus,

1. $s'^M(l'^M -_A 1) = s^M(l'^M -_A 1) = \langle \text{HEADER } h^M\ p^M\ m^M \rangle$
2. $s'(l' -_A 1) = s(l' -_A 1) = \langle \text{HEADER } h\ p\ m \rangle$
3. $h^M = h$
4. $p^M = p$
5. $m^M = m$
6. By definition of $\simeq'_0$, for all l_1^M, l_1, $(s^M, s \vdash l_1^M \simeq_0 l_1)$ holds if and only if either $(s'^M, s' \vdash l_1^M \simeq'_0 l_1)$ holds or $l_1^M = l^M$ and $l_1 = l$. We thus have that for all $0 \leq i < \mho m$, $(s'^M, s' \vdash (l'^M +_A i) \simeq'_0 (l' +_A i))$.

Thus, by definition of term correspondence, $(s'^M, s' \vdash cell^M \simeq' cell)$ holds.

Case $cell^M = \langle \text{FIXNUM } m \rangle$ **and** $cell = \langle \text{FIXNUM } m \rangle$**:**

By definition of term correspondence, $(s'^M, s' \vdash cell^M \simeq' cell)$ holds.

Case $cell^M = imm$ **and** $cell = imm$**:**

By definition of term correspondence, $(s'^M, s' \vdash cell^M \simeq' cell)$ holds.

Case $cell^M = b^*$ **and** $cell = b^*$**:**

By definition of term correspondence, $(s'^M, s' \vdash cell^M \simeq' cell)$ holds.

□

Lemma 39 Let $\simeq_0$ be a location correspondence and $\simeq$ be the term correspondence induced by $\simeq_0$. Let

$$\Sigma^M = \langle t^M, n^M, c^M, v^M, a^M, u^M, k^M, s^M, r_1^M, r_2^M, r_3^M, r_4^M \rangle, \text{ and}$$
$$\Sigma = \langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4 \rangle$$

be such that $\Sigma^M \cong \Sigma$ with witnesses $\simeq_0$ and $\simeq$. Let $d^M = \langle \text{PTR } l^M \rangle$, $d = \langle \text{PTR } l \rangle$, $(s^M, s \vdash d^M \simeq d)$, and $(s^M, s \vdash vd^M \simeq vd)$ for some l^M, l, vd^M, vd. Then,

1. `template-set`$(d^M, \Sigma^M) \cong$ `template-set`(d, Σ)
2. `codevector-set`$(d^M, \Sigma^M) \cong$ `codevector-set`(d, Σ)
3. `offset-set`$(i, \Sigma^M) \cong$ `offset-set`(i, Σ)
4. `offset-inc`$(i, \Sigma^M) \cong$ `offset-inc`(i, Σ)
5. `value-set`$(d^M, \Sigma^M) \cong$ `value-set`(d, Σ)
6. `env-set`$(d^M, \Sigma^M) \cong$ `env-set`(d, Σ)
7. `cont-set`$(d^M, \Sigma^M) \cong$ `cont-set`(d, Σ)
8. `spare1-set`$(d^M, \Sigma^M) \cong$ `spare1-set`(d, Σ)
9. `spare2-set`$(d^M, \Sigma^M) \cong$ `spare2-set`(d, Σ)
10. `spare3-set`$(i, \Sigma^M) \cong$ `spare3-set`(i, Σ)
11. `stack-push`$(d^M, \Sigma^M) \cong$ `stack-push`(d, Σ)
12. `stack-pop`$(\Sigma^M) \cong$ `stack-pop`(Σ)
13. `stack-clear`$(\Sigma^M) \cong$ `stack-clear`(Σ)
14. `stob-desc-set`$(d^M, i, vd^M, \Sigma^M) \cong$ `stob-desc-set`(d, i, vd, Σ)
15. `stob-byte-set`$(d^M, i, b^M, \Sigma^M) \cong$ `stob-byte-set`(d, i, b, Σ)

16. $\texttt{make-desc-stob}^M(h, p, m, \Sigma^M) \cong \texttt{make-desc-stob}(h, p, m, \Sigma)$

17. $\texttt{make-desc-stob-from-stack}^M(h, p, \Sigma^M) \cong$
$\texttt{make-desc-stob-from-stack}(h, p, \Sigma)$

18. $\texttt{make-byte-stob}^M(h, p, m, \Sigma^M) \cong \texttt{make-byte-stob}(h, p, m, \Sigma)$

Proof:
Cases 1–10: By definition, $\texttt{template-set}(d^M, \Sigma^M) = \Sigma^M[t'^M = d^M]$ and $\texttt{template-set}(d, \Sigma) = \Sigma[t' = d]$. Since $\Sigma^M \cong \Sigma$ and $(s^M, s \vdash d^M \simeq d)$, we have by definition that $\Sigma^M[t'^M = d^M] \cong \Sigma[t' = d]$ as desired. Cases 2–10 are proved similarly.

Cases 11–13: Since $\Sigma^M \cong \Sigma$, we have by definition that $\#a^M = \#a$ and $(s^M, s \vdash a^M(i) \simeq a(i))$ for all $0 \leq i < \#a$. Now, $\texttt{stack-push}(d^M, \Sigma^M) = \Sigma[a'^M = d^M :: a^M]$ and $\texttt{stack-push}(d, \Sigma) = \Sigma[a' = d :: a]$. Since $(s^M, s \vdash d^M \simeq d)$, we have that $\#a'^M = \#a'$ and $(s^M, s \vdash a'^M(i) \simeq a'(i))$ for all $0 \leq i < \#a'$. Thus $\Sigma[a'^M = d^M :: a^M] \cong \Sigma[a' = d :: a]$. Cases 12 and 13 are proved similarly.

Cases 14–15: By assumption, $d^M = \langle \text{PTR } l^M \rangle$ and $d = \langle \text{PTR } l \rangle$. Let $s^M(l^M -_A 1) = \langle \text{HEADER } h^M \ p^M \ m^M \rangle$ and $s(l -_A 1) = \langle \text{HEADER } h \ p \ m \rangle$. Now, $\Sigma'^M = \texttt{stob-desc-set}(d^M, i, vd^M, \Sigma^M) = \Sigma[s'^M = s^M[(l^M +_A i) \mapsto vd^M]]$ and $\Sigma' = \texttt{stob-desc-set}(d, i, vd, \Sigma) = \Sigma[s' = s[(l +_A i) \mapsto vd]]$.

Let $\simeq'_0$ be a location correspondence such that $(s'^M, s' \vdash l^M \simeq'_0 l)$ holds if and only if $(s^M, s \vdash l^M \simeq_0 l)$ holds. Note that this is a valid definition since $ldom(s'^M) = ldom(s^M)$ and $ldom(s') = ldom(s)$. Let $\simeq'$ be the term correspondence induced by $\simeq'_0$. We show that $\Sigma'^M \cong \Sigma'$ by choosing $\simeq'_0$ and $\simeq'$ as the witness location and term correspondences, and showing that:

- for all $l^M \in ldom(s^M)$ and $l \in ldom(s)$,
$(s'^M, s' \vdash l^M \simeq'_0 l) \Rightarrow (s'^M, s' \vdash s'^M(l^M) \simeq' s'(l))$

  Let $(s'^M, s' \vdash \overline{l^M} \simeq'_0 \overline{l})$.

  Case $\overline{l^M} = l^M +_A i$**:**
  Since $(s'^M, s' \vdash \overline{l^M} \simeq'_0 \overline{l})$ we have by definition of $\simeq'_0$ that $\overline{l} = l +_A i$. By definition, $s'^M(l^M +_A i) = vd^M$ and $s'(l +_A i) = vd$. Since $(s^M, s \vdash vd^M \simeq vd)$ by assumption, the result follows by Lemma 37.

  Case $\overline{l^M} \neq l^M +_A i$**:**
  Since $(s'^M, s' \vdash \overline{l^M} \simeq'_0 \overline{l})$, we have by definition of $\simeq'_0$ that $\overline{l} \neq$

$l +_A i$ and $(s^M, s \vdash \overline{l^M} \simeq_0 \bar{l})$. Since $\Sigma^M \cong \Sigma$, we have that $(s^M, s \vdash s^M(\overline{l^M}) \simeq s(\bar{l}))$. Hence, by Lemma 37, $(s'^M, s' \vdash s^M(\overline{l^M}) \simeq' s(\bar{l}))$. Now $s^M(\overline{l^M}) = s'^M(\overline{l^M})$ since $\overline{l^M} \neq l^M +_A i$, and $s(\bar{l}) = s'(\bar{l})$ since $\bar{l} \neq l +_A i$. Thus, $(s'^M, s' \vdash s'^M(\overline{l^M}) \simeq' s'(\bar{l}))$ as desired.

Otherwise:

There are no other cases since $\simeq_0'$ is a 1-to-1 map and $(s^M, s \vdash l^M +_A i \simeq_0 l +_A i)$ by assumption.

- $(s'^M, s' \vdash t^M \simeq' t)$
  Since $\Sigma^M \cong \Sigma$, we have that $(s^M, s \vdash t^M \simeq t)$. The result then follows by Lemma 37.

- $n^M = n$
  Follows from $\Sigma^M \cong \Sigma$.

- $(s'^M, s' \vdash v^M \simeq' v)$
  Follows from $\Sigma^M \cong \Sigma$ and Lemma 37.

- $\#a^M = \#a$ and $(\forall 0 \leq i < \#a)(s'^M, s' \vdash a^M(i) \simeq' a(i))$
  Follows from $\Sigma^M \cong \Sigma$ and Lemma 37.

- $(s'^M, s' \vdash u^M \simeq' u)$
  Follows from $\Sigma^M \cong \Sigma$ and Lemma 37.

- $(s'^M, s' \vdash k^M \simeq' k)$
  Follows from $\Sigma^M \cong \Sigma$ and Lemma 37.

- $(s'^M, s' \vdash r_2^M \simeq' r_2)$
  Follows from $\Sigma^M \cong \Sigma$ and Lemma 37.

- $r_3^M = r_3$
  Follows from $\Sigma^M \cong \Sigma$.

- $r_4^M = r_4$
  Follows from $\Sigma^M \cong \Sigma$.

Case 15 is proved similarly.

Cases 16–18: Now, $\Sigma'^M = \texttt{make-desc-stob}^M(h, p, m, \Sigma) = \Sigma^M[s'^M = s^M \oplus_0 \langle h\ p\ m\rangle][r_1'^M = \langle 0, \#s^M(0)\rangle]$ and $\Sigma' = \texttt{make-desc-stob}(h, p, m, \Sigma) = \Sigma[s' = s \oplus_0 \langle h\ p\ m\rangle][r_1' = \langle 0, \#s(0)\rangle]$.

Let $\simeq_0$ be a location correspondence and $\simeq$ be the term correspondence induced by $\simeq_0$

Let $\simeq_0'$ be a location correspondence such that, for all l^M, l, $(s'^M, s' \vdash l^M \simeq_0' l)$ holds if and only if either $(s^M, s \vdash l^M \simeq_0 l)$ holds or $l^M = \langle 0, \#s^M + i\rangle$ and $l = \langle 0, \#s + i\rangle$ hold for some $1 \leq i < m + 1$. Let $\simeq'$ be the term correspondence induced by $\simeq_0'$. We show that $\Sigma'^M \cong \Sigma'$ by choosing $\simeq_0'$ and $\simeq'$ as the witness location and term correspondences, and showing that:

- for all $l^M \in ldom(s^M)$ and $l \in ldom(s)$,
  $(s'^M, s' \vdash l^M \simeq_0' l) \Rightarrow (s'^M, s' \vdash s'^M(l^M) \simeq' s'(l))$

  Let $(s'^M, s' \vdash \overline{l^M} \simeq_0' \overline{l})$.

  Case $\overline{l^M} = \langle 0, \#s^M(0) + i\rangle$ **for some** $0 \leq i < m + 1$**:**
  Since $(s'^M, s' \vdash \overline{l^M} \simeq_0' \overline{l})$, we have by definition of $\simeq_0'$ that $i > 0$ and $\overline{l} = \langle 0, \#s(0) + i\rangle$. By definition, $s'^M(\overline{l^M}) = d_u^M$ and $s'(\overline{l}) = d_u$. The result follows since $(s'^M, s' \vdash d_u^M \simeq' d_u)$ by definition.

  Case $\overline{l^M} <_A \langle 0, \#s^M(0)\rangle$**:**
  Since $(s'^M, s' \vdash \overline{l^M} \simeq_0' \overline{l})$, we have by definition of $\simeq_0'$ that $\overline{l} <_A \langle 0, \#s(0)\rangle$ and $(s^M, s \vdash \overline{l^M} \simeq_0 \overline{l})$. Since $\Sigma^M \cong \Sigma$, we have that $(s^M, s \vdash s^M(\overline{l^M}) \simeq s(\overline{l}))$. Hence, by Lemma 38, $(s'^M, s' \vdash s^M(\overline{l^M}) \simeq' s(\overline{l}))$. Now $s^M(\overline{l^M}) = s'^M(\overline{l^M})$ and $s(\overline{l}) = s'(\overline{l})$. Thus $(s'^M, s' \vdash s'^M(\overline{l^M}) \simeq' s'(\overline{l}))$ as desired.

- $(s'^M, s' \vdash t^M \simeq' t)$
  Since $\Sigma^M \cong \Sigma$, we have that $(s^M, s \vdash t^M \simeq t)$. The result then follows by Lemma 38.

- $n^M = n$
  Follows from $\Sigma^M \cong \Sigma$.

- $(s'^M, s' \vdash v^M \simeq' v)$
  Follows from $\Sigma^M \cong \Sigma$ and Lemma 38.

- $\#a^M = \#a$ and $(\forall 0 \leq i < \#a)(s'^M, s' \vdash a^M(i) \simeq' a(i))$
  Follows from $\Sigma^M \cong \Sigma$ and Lemma 38.

- $(s'^M, s' \vdash u^M \simeq' u)$
  Follows from $\Sigma^M \cong \Sigma$ and Lemma 38.

- $(s'^M, s' \vdash k^M \simeq' k)$
  Follows from $\Sigma^M \cong \Sigma$ and Lemma 38.

- $(s'^M, s' \vdash r_2^M \simeq' r_2)$
  Follows from $\Sigma^M \cong \Sigma$ and Lemma 38.

- $r_3^M = r_3$
  Follows from $\Sigma^M \cong \Sigma$.

- $r_4^M = r_4$
  Follows from $\Sigma^M \cong \Sigma$.

Case 17 and 18 are proved similarly.

□

Lemma 40 *Let Σ^M and Σ be* MSBCM *and* GSBCM *states respectively. If $\Sigma^M \cong \Sigma$, then Σ^M is a halt state of the* MSBCM *if and only if Σ is a halt state of the* GSBCM.

Proof: Let $\Sigma^M = \langle t^M, n^M, c^M, v^M, a^M, u^M, k^M, s^M, r_1^M, r_2^M, r_3^M, r_4^M \rangle$ be an MSBCM halt state, let $\Sigma = \langle t, n, c, v, a, u, k, s, r_1, r_2, r_3, r_4 \rangle$ be a GSBCM state, and let $\Sigma^M \cong \Sigma$. We show that Σ is a GSBCM halt state.

Since Σ^M is a MSBCM halt state, we have that

- $\mathcal{S}(c^M, s^M) = \langle$CODEVECTOR $\#f\ b^{*M}\rangle$,

- $b^{*M}(n^M) =$ `return`, and

- $k^M = \langle$IMMEDIATE HALT$\rangle$.

Since $\Sigma^M \cong \Sigma$, we have that $(s^M, s \vdash c^M \simeq c)$ and $(s^M, s \vdash k^M \simeq k)$. Hence, we have by definition of a term correspondence that

- $\mathcal{S}(c, s) = \langle$CODEVECTOR $\#f\ b^*\rangle$,

- $b^*(n) =$ `return`, and

- $k = \langle$IMMEDIATE HALT$\rangle$.

and so Σ is a GSBCM halt state.

The converse holds similarly.

□

Lemma 41 *Let Σ_i^M and Σ_i be* MSBCM *and* GSBCM *initial states respectively. If $\Sigma_i^M \cong \Sigma_i$, then*

1. *$\mathcal{R}^*_{msbcm}\Sigma_i^M$ is defined if and only if $\mathcal{R}^*_{gsbcm}\Sigma_i$ is defined,*
2. *If $\mathcal{R}^*_{msbcm}\Sigma_i^M$ and $\mathcal{R}^*_{gsbcm}\Sigma_i$ are defined, then $\mathcal{R}^*_{msbcm}\Sigma_i^M \cong \mathcal{R}^*_{gsbcm}\Sigma_i$.*

Proof: Each action rule of the MSBCM and GSBCM is expressed as the composition of the microcode functions and auxiliary functions. Each of these functions preserve term and state correspondence and hence so do their composition. Thus each action rule preserves state correspondence and hence so does the union and transitive closure of the action rules.

□

3.6 Correspondence of Final Answers

The following lemma asserts that the MSBCM and GSBCM answer functions map corresponding states to equal natural numbers.

Lemma 42 *Let Σ_f^M and Σ_f be* MSBCM *and* GSBCM *halt states respectively. If $\Sigma_f^M \cong \Sigma_f$, then $A_{msbcm}(\Sigma_f^M) = A_{gsbcm}(\Sigma_f)$.*

Proof: Let v^M and v be the value registers of Σ_f^M and Σ_f respectively. Since $\Sigma_f^M \cong \Sigma_f$, we have by definition of state correspondence that v^M and v are related (i.e. $(s^M, s \vdash v^M \simeq v)$). By definition of term correspondence, if $v^M = \langle \text{FIXNUM } m \rangle$ then $v = \langle \text{FIXNUM } m \rangle$ and so $A_{msbcm}(\Sigma_f^M) = A_{gsbcm}(\Sigma_f) = m$. If v^M is not of the form $\langle \text{FIXNUM } m \rangle$ then neither is v, and so $A_{msbcm}(\Sigma_f^M)$ and $A_{gsbcm}(\Sigma_f)$ are both undefined.

□

3.7 Correspondence of Semantics

We can now prove the main theorem of this chapter.

Theorem 43 *If $\mathcal{P}$ is an* SBC *program, then its operational semantics as given by the* MSBCM *yields the same answer as its operational semantics as given by the* GSBCM*. That is,*

$$\mathcal{O}^{sbc}_{msbcm}[\![\mathcal{P}]\!] = \mathcal{O}^{sbc}_{gsbcm}[\![\mathcal{P}]\!]$$

Proof: Follows immediately from Lemmas 23, 41, and 42.

□

4 Finite SBC State Machine

In this chapter, we give Stored Byte Code (SBC) programs an (operational) semantics in terms of a deterministic state machine with "finite" concrete states. The state machine is called the *Finite Stored Byte Code Machine (*FSBCM*)*. We show that this operational semantics is equivalent to the operational semantics in terms of the GSBCM. This chapter concludes the implementation and verification of the VLISP byte code interpreter.

This chapter presents:

1. an operational semantics of SBC programs in terms of a state machine called the Finite Stored Byte Code Machine (FSBCM).
2. a correspondence relation that relates GSBCM states and FSBCM states;
3. a proof that the operational semantics of SBC as given by the FSBCM is faithful to the operational semantics of SBC as given by the GSBCM. That is, if the FSBCM operational semantics assigns a meaning to an SBC program, then the GSBCM operational semantics assigns the same meaning to the program.

4.1 FSBC State Machine (FSBCM)

4.1.1 Finite Stored Byte Code (FSBC)

The Finite Stored Byte Code (FSBC) provides the syntactic objects that form the states of the FSBCM. All syntactic objects of FSBC have a finite structure. They are either finite numbers or finite sequences, with bounds specified by the syntax.

The abstract syntax of Finite Stored Byte Code (FSBC) is defined below. Let $z = 2 * \texttt{heap-size-in-words}$.

$$\begin{array}{rcl} \mathit{program} & ::= & \mathit{term} \\ \mathit{term} & ::= & \langle \mathit{store}\ \mathit{nat} \rangle \\ \mathit{store} & ::= & \mathit{nat}^z \\ \mathit{loc} & ::= & \mathit{nat} \end{array}$$

We will use i,m,n, t, c, v, u, k, and r-like variables for numbers, and s-like variables for *(*FSBC*) stores*.

We define a family of partial functions from MSBC terms to FSBCM terms (i.e., natural numbers). Since all SBC terms are MSBC terms, these functions will also map SBC terms to natural numbers.

$$
\begin{array}{lcll}
F^{byte}(\texttt{call}) & = & 0 & \\
F^{byte}(\texttt{return}) & = & 1 & \\
F^{byte}(\texttt{make-cont}) & = & 2 & \\
F^{byte}(\texttt{literal}) & = & 3 & \\
F^{byte}(\texttt{closure}) & = & 4 & \\
F^{byte}(\texttt{global}) & = & 5 & \\
F^{byte}(\texttt{local}) & = & 6 & \\
F^{byte}(\texttt{set-global!}) & = & 7 & \\
F^{byte}(\texttt{set-local!}) & = & 8 & \\
F^{byte}(\texttt{push}) & = & 9 & \\
F^{byte}(\texttt{make-env}) & = & 10 & \\
F^{byte}(\texttt{make-rest-list}) & = & 11 & \\
F^{byte}(\texttt{unspecified}) & = & 12 & \\
F^{byte}(\texttt{jump}) & = & 13 & \\
F^{byte}(\texttt{jump-if-false}) & = & 14 & \\
F^{byte}(\texttt{check-args=}) & = & 15 & \\
F^{byte}(\texttt{check-args>=}) & = & 16 & \\
F^{byte}(\texttt{empty}) & = & 18 & \\
F^{byte}(r) & = & m & \text{for } 19 \leq m < \texttt{MAXBYTEVAL} \\
F^{byte}(m) & = & m & \text{if } 0 \leq m < \texttt{MAXBYTEVAL} \\
 & & & \\
F^{byteseq}(\langle\rangle) & = & 0 & \\
F^{byteseq}(b^* \frown \langle b \rangle) & = & \texttt{MAXBYTEVAL} * F^{byteseq}(b^*) + F^{byte}(b) &
\end{array}
$$

$$
\begin{array}{lcl}
F^{loc}(\langle 0, m\rangle) & = & \texttt{low-heap-base} + m \\
 & & \quad \text{if } (\texttt{low-heap-base} + m) < \texttt{low-heap-limit} \\
F^{loc}(\langle 1, m\rangle) & = & \texttt{high-heap-limit} + m - 1 \\
 & & \quad \text{if } \texttt{stack-base} \leq (\texttt{high-heap-limit} + m - 1) \\
F^{loc}(\langle 2, m\rangle) & = & \texttt{old-low-heap-base} + m \\
 & & \quad \text{if } (\texttt{old-low-heap-base} + m) < \texttt{old-low-heap-limit}
\end{array}
$$

$$
\begin{array}{lcll}
F^{bool}(\#f) & = & 0 & \\
F^{bool}(\#t) & = & 1 & \\
 & & & \\
F^{imm}(\text{FALSE}) & = & 64 * 0 + 0 & \\
F^{imm}(\text{TRUE}) & = & 64 * 0 + 1 & \\
F^{imm}(\text{CHAR}(m)) & = & 64 * m + 2 & \text{if } 0 \leq m < \texttt{MAXBYTEVAL} \\
F^{imm}(\text{UNSPECIFIED}) & = & 64 * 0 + 3 & \\
F^{imm}(\text{UNDEFINED}) & = & 64 * 0 + 4 & \\
F^{imm}(\text{NULL}) & = & 64 * 0 + 5 & \\
F^{imm}(\text{HALT}) & = & 64 * 0 + 6 & \\
F^{imm}(\text{EMPTY-ENV}) & = & 64 * 0 + 7 & \\
F^{imm}(\text{EOF}) & = & 64 * 0 + 8 & \\
 & & & \\
F^{fixnum}(m) & = & m & \text{if } \texttt{MINFIXNUM} \leq m < \texttt{MAXFIXNUM} \\
 & & & \\
F^{htag}(\text{PAIR}) & = & 0 & \\
F^{htag}(\text{SYMBOL}) & = & 1 & \\
F^{htag}(\text{VECTOR}) & = & 2 & \\
F^{htag}(\text{CLOSURE}) & = & 3 & \\
F^{htag}(\text{LOCATION}) & = & 4 & \\
F^{htag}(\text{PORT}) & = & 5 & \\
F^{htag}(\text{CONTINUATION}) & = & 6 & \\
F^{htag}(\text{TEMPLATE}) & = & 7 & \\
F^{htag}(\text{ENVIRONMENT}) & = & 8 & \\
F^{htag}(\text{STRING}) & = & 10 & \\
F^{htag}(\text{CODEVECTOR}) & = & 11 & \\
 & & & \\
F^{desc}(\langle \text{FIXNUM } m \rangle) & = & 4 * F^{fixnum}(m) + 0 & \\
F^{desc}(\langle \text{IMMEDIATE } imm \rangle) & = & 4 * F^{imm}(imm) + 1 & \\
F^{desc}(\langle \text{HEADER } h\ p\ m \rangle) & = & 256 * m + 128 * F^{bool}(p) + 4 * F^{htag}(h) + 2 & \\
F^{desc}(\langle \text{PTR } l \rangle) & = & 4 * F^{loc}(l) + 3 & \\
 & & & \\
F^{cell}(desc) & = & F^{desc}(desc) & \\
F^{cell}(b^*) & = & F^{byteseq}(b^*) &
\end{array}
$$

4.1.2 Stores

A *store* s is represented as a fixed-length sequence of natural numbers. *Segments* of the store are represented as triples of locations $\langle l_1, l_2, l_3 \rangle$ where $l_1 \leq l_2 \leq l_3 \leq \#s$. This denotes the map from the locations l_1, $l_1 + 1$, ..., $l_3 - 1$ to the cells $s(l_1)$, $s(l_1 + 1)$, ..., $s(l_3 - 1)$. We call a subset of the domain of a segment to be the *active portion* of the segment; we will only be interested in the value of a segment at locations within this subset. We distinguish between *positive segments* and *negative segments*. The *active* portion of a positive segment consists of the locations l_1, $l_1 + 1$, ..., $l_2 - 1$, while the active portion of a negative segment consists of the locations l_2, $l_2 + 1$, ..., $l_3 - 1$.

4.1.3 States

The *states* of the FSBCM are the sequences of the form

$$\langle t, n, c, v, g_a, g_{a'}, u, k, s, g_0, g_1, g_2, g_3, r_1, r_2, r_3, r_4 \rangle$$

where $g_a, g_{a'}, g_0, g_1, g_2, g_3$ are six segments of the store s such that:

1. $g_0 = \langle \texttt{low-heap-base}, \texttt{hp}, \texttt{low-heap-limit} \rangle$
2. $g_a = \langle \texttt{low-heap-limit}, \texttt{sp}, \texttt{stack-base} \rangle$
3. $g_1 = \langle \texttt{stack-base}, \texttt{stack-base}, \texttt{high-heap-limit} \rangle$
4. $g_2 = \langle \texttt{old-low-heap-base}, \texttt{old-hp}, \texttt{old-low-heap-limit} \rangle$
5. $g_{a'} = \langle \texttt{old-low-heap-limit}, \texttt{old-sp}, \texttt{old-stack-base} \rangle$
6. $g_3 = \langle \texttt{old-stack-base}, \texttt{old-stack-base}, \texttt{old-high-heap-limit} \rangle$
7. Either `old-high-heap-limit` = `low-heap-base` or `high-heap-limit` = `old-low-heap-base`.

The components of a state are called, in order, its *template*, *offset*, *codevector*, *value*, *argument stack*, *old argument stack*, *environment*, *continuation*, *store*, *segment0*, *segment1*, *segment2*, *segment3*, *spare1*, *spare2*, *spare3*, and *spare4*, and we may informally speak of them as being held in registers. Note that the definition implies that the six segments are disjoint, i.e., if a location l is in the domain of a segment, then it is not in the domain of any other segment.

4.1.4 Action Rules

In Section 2.2.8, we specified the action rules of the MSBCM as the composition of stored object manipulators, microcode functions, and other auxiliary functions. Stored object manipulators were in turn specified in terms of the microcode functions and auxiliary functions. In Section 3.1.4, the action rules of the GSBCM were given the same definition forms as in the MSBCM. Similarly, the action rules of the FSBCM are given those same definition forms. We denote the transitive closure of the union of all FSBCM action rules by $\mathcal{R}^*_{fsbcm}$.

Now, the extensions of the GSBCM action rules are not the same as the extensions of the corresponding MSBCM action rules since three GSBCM mutator functions differ from the corresponding MSBCM functions. Similarly, the definitions of the FSBCM microcode functions and auxiliary functions differ from those in the GSBCM; hence, the extensions of the action rules differ from the extensions of the corresponding GSBCM action rules. We present the implementation of the FSBCM global state, microcode functions, and auxiliary functions in the next section.

4.1.5 Implementation

Global State

The following functions define the global state of the FSBCM.

```
(define *template* unspecified)          ; t
(define *codevector* *hp*)               ; c
(define *offset* 0)                      ; n
(define *value* unspecified)             ; v
(define *env* unspecified)               ; u
(define *cont* unspecified)              ; k
(define *spare1* unspecified)            ; r1
(define *spare2* unspecified)            ; r2
(define *spare3* 0)                      ; r3
(define *gc-spare* unspecified)          ; r4
(define *symbol-table* unspecified)      ; symbol table

(define *low-heap-base* heap-low)        ; g0(0)
(define *hp* heap-low)                   ; g0(1)
(define *low-heap-limit* heap-mid)       ; g0(2) = ga(0)
(define *sp* heap-mid)                   ; ga(1)
(define *stack-base* heap-mid)           ; ga(2) = g1(0) = g1(1)
(define *high-heap-limit* heap-mid)      ; g1(2)
```

```
(define *old-low-heap-base* heap-mid)     ; g2(0)
(define *old-hp* heap-mid)                ; g2(1)
(define *old-low-heap-limit* heap-mid)    ; g2(2) = ga'(0)
(define *old-sp* heap-high)               ; ga'(1)
(define *old-stack-base* heap-high)       ; ga'(2) = g3(0) = g3(1)
(define *old-high-heap-limit* heap-high)  ; g3(2)
```

Microcode Functions

Observers

```
(define (template-ref) *template*)
(define (codevector-ref) *codevector*)
(define (offset-ref) *offset*)
(define (value-ref) *value*)
(define (env-ref) *env*)
(define (cont-ref) *cont*)
(define (spare1-ref) *spare1*)
(define (spare2-ref) *spare2*)
(define (spare3-ref) *spare3*)
(define (gc-spare-ref) *gc-spare*)
(define (symbol-table-ref) *symbol-table*)

(define (stack-top) (stack-ref *sp*))
(define (stack-nth n) (stack-ref (addr+ *sp* n)))
(define (stack-empty?) (addr= *stack-base* *sp*))
(define (stack-length) (addr- *stack-base* *sp*))

(define (stob-desc-ref a i)
  (heap-ref (heap-pointer->addr a) i))
(define (stob-binarray-ref a i)
  (heap-byte-ref (heap-pointer->addr a) i))

(define (stob-header a) (heap-ref (heap-pointer->addr a) -1))
(define (desc-stob? a) (desc-header-tag? (stob-type a)))
(define (binarray-stob? a) (binarray-header-tag? (stob-type a)))

(define (stob-type a) (header-tag (stob-header a)))
(define (stob-mutable? a hdr-tag)
  (= (header-tags-bin (stob-header a))
     (make-header-tags-bin hdr-tag \#t)))
(define (stob-size-in-bins a)
  (header-size-in-bins (stob-header a)))
(define (stob-size-in-cells a)
  (header-size-in-cells (stob-header a)))
```

Mutators

```
(define (template-set! v) (set! *template* v))
(define (codevector-set! v)
  (set! *codevector* (heap-pointer->addr v)))
(define (offset-set! v) (set! *offset* v))
(define (offset-inc! n) (set! *offset* (+ *offset* n)))
(define (value-set! v) (set! *value* v))
(define (env-set! v) (set! *env* v))
(define (cont-set! v) (set! *cont* v))
(define (spare1-set! v) (set! *spare1* v))
(define (spare2-set! v) (set! *spare2* v))
(define (spare3-set! v) (set! *spare3* v))
(define (gc-spare-set! v) (set! *gc-spare* v))
(define (symbol-table-set! v) (set! *symbol-table* v))

(define (stack-push! v)
  (assert! (addr< *low-heap-limit* *sp*)
           "stack-push!: argument stack overflow")
  (set! *sp* (addr+ *sp* -1))
  (stack-set! *sp* v))

(define (stack-pop!) (set! *sp* (addr+ *sp* 1)))
(define (stack-clear!) (set! *sp* *stack-base*))

(define (stack-restore! ret)
  (lambda (ptr)
    (stack-clear!)
    (let ((a (addr+ (heap-pointer->addr ptr) 4)))
      (do ((src (addr+ a (- (stob-size-in-cells ptr) 5))
                (addr+ src -1))
           (a a))
          ((addr< src a) (ret))
        (stack-push! (heap-ref src 0))
        ))))

(define (stob-desc-set! a i v)
  (heap-set! (heap-pointer->addr a) i v))
(define (stob-binarray-set! a i v)
  (heap-byte-set! (heap-pointer->addr a) i v))

(define (make-desc-stob! ret s m)
  (lambda (w) ((alloc! ret) ((make-desc-header s m) w) w)))
(define (make-binarray-stob! ret s m)
  (lambda (w)
    ((alloc! ret) ((make-binarray-header s m) w) (bins->cells w))))
```

```
(define (alloc-object-from-stack hdr)
  (spare1-set! (addr->pointer *sp*))
  (stack-push! hdr)
  (set! *stack-base* *sp*)
  (set! *low-heap-limit* (addr+ *stack-base* (- stack-size-in-words)))
  )
(define *ret-make-desc-stob-from-stack!* (lambda () 0))
(define (make-desc-stob-from-stack! ret s m)
  (lambda ()
    (let ((new-low-heap-limit
           (addr+ *sp* (- -1 stack-size-in-words)))
          (hdr ((make-desc-header s m) (stack-length)))
          )
      (if (addr< *hp* new-low-heap-limit)
          (begin
            (alloc-object-from-stack hdr)
            (ret))
          (begin
            (set! *ret-make-desc-stob-from-stack!* ret)
            (gc-spare-set! hdr)
            (garbage-collect! ret-make-desc-stob-from-stack!))
          ))))
(define (ret-make-desc-stob-from-stack!)
  (alloc-object-from-stack (gc-spare-ref))
  (*ret-make-desc-stob-from-stack!*))

(define (nh-alloc! hdr newhp)
  (heap-set! *hp* 0 hdr)
  (spare1-set! (addr->pointer (addr+ *hp* 1)))
  (set! *hp* newhp))

(define *alloc-ret* (lambda () 0))
(define (alloc! ret)
  (lambda (hdr size)
    (let ((newhp (addr+ *hp* (+ size 1))))
      (if (addr< newhp *low-heap-limit*)
          (begin
            (nh-alloc! hdr newhp)
            (ret))
          (begin
            (gc-spare-set! hdr)
            (set! *alloc-ret* ret)
            (garbage-collect! after-garbage-collect!)
            )))))
```

```
(define (after-garbage-collect!)
  (let ((newhp
         (addr+ *hp* (+ (header-size-in-cells (gc-spare-ref)) 1))))
    (assert! (addr< newhp *low-heap-limit*)
             "Garbage collector could not reclaim enough space.")
    (nh-alloc! (gc-spare-ref) newhp)
    (*alloc-ret*)))
```

4.1.6 SBC Operational Semantics

We first define a function *load* that maps SBC stores to FSBCM stores. An SBC store is represented as the first store segment within an FSBCM store, and all SBC pointers within the SBC store are mapped to corresponding FSBC pointers. Let s^{sbc} be an SBC store. Then Let s^{sbc} be an SBC store. Then

$$relocate(cell, base) \stackrel{\text{def}}{=} \begin{array}{ll} \langle \text{PTR } base + m \rangle & \text{if } cell = \langle \text{PTR } m \rangle \\ cell & \text{otherwise} \end{array}$$

$$load(s^{sbc}, base) \stackrel{\text{def}}{=} s$$
$$\text{where } s(base + i) = relocate(s^{sbc}(i), base) \text{ for all } 0 \leq i < \#s^{sbc}$$

The FSBCM *Loader* L_{fsbcm} is defined to be a function that maps SBC programs to FSBCM initial states such that:

$$\begin{array}{l} L_{fsbcm}(\langle s^{sbc}\ t^{sbc} \rangle) \stackrel{\text{def}}{=} \\ \quad \langle t, 0, c, d_u, g_a, g_{a'}, F^{desc}(\langle \text{IMMEDIATE EMPTY-ENV} \rangle), \\ \qquad F^{desc}(\langle \text{IMMEDIATE HALT} \rangle), s, g_0, g_1, g_2, g_3, d_u, d_u, 0, d_u \rangle \\ \quad \text{if } d_u = F^{desc}(\langle \text{IMMEDIATE UNSPECIFIED} \rangle) \\ \quad \text{and } size^s = \texttt{heap-size-in-words} \\ \quad \text{and } size^a = \texttt{stack-size-in-words} \\ \quad \text{and } g_0 = \langle 0, \#s^{sbc}, (size^s - size^a) \rangle \\ \quad \text{and } g_a = \langle (size^s - size^a), size^s, size^s \rangle \\ \quad \text{and } g_1 = \langle size^s, size^s, size^s \rangle \\ \quad \text{and } g_2 = \langle size^s, size^s, (2 * size^s - size^a) \rangle \\ \quad \text{and } g'_a = \langle (2 * size^s - size^a), 2 * size^s, 2 * size^s \rangle \\ \quad \text{and } g_3 = \langle 2 * size^s, 2 * size^s, 2 * size^s \rangle \\ \quad \text{and } s = load(s^{sbc}, g_0(0)) \\ \quad \text{and } t = relocate(t^{sbc}, g_0(0)) \\ \quad \text{and } \mathcal{S}(t, s) = \langle \text{TEMPLATE } \#f\ c::o^* \rangle \end{array}$$

The FSBCM *Answer* A_{fsbcm} is defined to be a partial function that maps FSBCM halt states Σ^{f}_{fsbcm} to natural numbers such that

$$A_{fsbcm}(\langle t, n, c, v, g_a, g_{a'}, u, k, s, g_0, g_1, g_2, g_3, r_1, r_2, r_3, r_4\rangle) \stackrel{\text{def}}{=} m$$
$$\text{if } v = \langle \text{FIXNUM } m\rangle$$

Definition 44 *Let $\mathcal{P}$ be an* SBC *program. Then $L_{fsbcm}(\mathcal{P})$ is an initial state of the* FSBCM. *If $\mathcal{R}^{*}_{fsbcm}(L_{fsbcm}(\mathcal{P}))$ is defined, then the meaning of program $\mathcal{P}$ as given by the* SBC *operational semantics is*

$$\mathcal{O}_{fsbcm}[\![\mathcal{P}]\!] \stackrel{\text{def}}{=} A_{fsbcm}(\mathcal{R}^{*}_{fsbcm}(L_{fsbcm}(\mathcal{P})))$$

Otherwise $\mathcal{O}_{fsbcm}[\![\mathcal{P}]\!]$ is undefined.

4.2 Correspondence Relation

Definition 45 *We define a* FSBCM state correspondence *relation*

$$\cong \subset \text{GSBCM}state \times \text{FSBCM}state$$

as follows:

Let

$$\Sigma_{gsbcm} = \langle t^G, n^G, c^G, v^G, a^G, u^G, k^G, s^G, r_1^G, r_2^G, r_3^G, r_4^G\rangle, \text{ and}$$
$$\Sigma = \langle t, n, c, v, g_a, g_{a'}, u, k, s, g_0, g_1, g_2, g_3, r_1, r_2, r_3, r_4\rangle$$

be a GSBCM *state and* FSBCM *state respectively.*

Then, $\Sigma_{gsbcm} \cong \Sigma$ holds if

- $F^{desc}(t^G) = t$
- $F^{offset}(n^G) = n$
- $F^{desc}(c^G) = c$
- $F^{desc}(v^G) = v$
- $(g_a(1) + i) < g_a(2)$ *for all* $0 \leq i < \#a^G$
- $F^{desc}(a^G(i)) = s(g_a(1) + i)$ *for all* $0 \leq i < \#a^G$
- $F^{desc}(u^G) = u$
- $F^{desc}(k^G) = k$

- *For all* $\langle i, m\rangle \in \mathrm{ldom}(s^G)$,

$$\begin{array}{ll} (g_i(0) + m) < g_i(1) & \textit{if } i \in \{0, 2\} \\ g_i(1) \leq (g_i(2) + m) & \textit{if } i \in \{1, 3\} \end{array}$$

- *For all* $\langle i, m\rangle \in \mathrm{ldom}(s^G)$,

$$F^{cell}(s^G(\langle i, m\rangle)) = \begin{array}{ll} s(g_i(0) + m) & \textit{if } i \in \{0, 2\} \\ s(g_i(2) + m) & \textit{if } i \in \{1, 3\} \end{array}$$

- $F^{desc}(r_1^G) = r_1$
- $F^{desc}(r_2^G) = r_2$
- $F^{desc}(r_3^G) = r_3$
- $F^{desc}(r_4^G) = r_4$

4.3 Establishing State Correspondence

Lemma 46 *The* GSBCM *Loader and the* FSBCM *Loader map* SBC *programs to initial states that are related by the state correspondence relation. That is, if* $\mathcal{P} = \langle s^{sbc}\ t^{sbc}\rangle$ *is an* SBC *program, then*

$$L_{gsbcm}(\mathcal{P}) \cong L_{fsbcm}(\mathcal{P})$$

Proof: All cases are immediate from the definitions of the loaders and state correspondence except for the following:

1. For all $\langle i, m\rangle \in ldom(s^G)$,

$$\begin{array}{ll} (g_i(0) + m) < g_i(1) & \text{if } i \in \{0, 2\} \\ g_i(1) \leq (g_i(2) + m) & \text{if } i \in \{1, 3\} \end{array}$$

   By definition of L_{gsbcm}, if $\langle i, m\rangle \in ldom(s^G)$ then $i = 0$ and $0 \leq m < \#s^{sbc}$. But by definition of L_{fsbcm}, $g_0(1) = g_0(0) + \#s^{sbc}$, and so the condition holds.

2. For all $\langle i, m\rangle \in ldom(s^G)$,

$$F^{cell}(s^G(\langle i, m\rangle)) = \begin{array}{ll} s(g_i(0) + m) & \text{if } i \in \{0, 2\} \\ s(g_i(2) + m) & \text{if } i \in \{1, 3\} \end{array}$$

By definition of L_{gsbcm}, if $\langle i, m\rangle \in ldom(s^G)$ then $i = 0$ and $0 \leq m < \#s^{sbc}$. If $s^{sbc}(m) = \langle \text{PTR } l\rangle$ then

$$s^G(\langle 0, m\rangle) = relocate^G(\langle \text{PTR } l\rangle) = \langle \text{PTR } \langle 0, l\rangle\rangle$$
$$s(g_0(0) + m) = relocate(\langle \text{PTR } l\rangle, g_0(0)) = \langle \text{PTR } g_0(0) + l\rangle$$

But $F^{cell}(\langle \text{PTR } \langle 0, l\rangle\rangle) = \langle \text{PTR } g_0(0) + l\rangle$ by definition.

If $s^{sbc}(m) \neq \langle \text{PTR } l\rangle$ then

$$s^G(\langle 0, m\rangle) = relocate^G(s^{sbc}(m)) = s^{sbc}(m)$$
$$s(g_0(0) + m) = relocate(s^{sbc}(m), g_0(0)) = F^{cell}(s^{sbc}(m))$$

The result thus holds trivially.

□

4.4 Preserving State Correspondence

Lemma 47 *Let Σ_i^G and Σ_i be* GSBCM *and* FSBCM *initial states respectively. If $\Sigma_i^G \cong \Sigma_i$, then*

1. *If $\mathcal{R}^*_{fsbcm}\Sigma_i$ is defined, then so is $\mathcal{R}^*_{gsbcm}\Sigma_i^G$.*
2. *If $\mathcal{R}^*_{gsbcm}\Sigma_i^G$ and $\mathcal{R}^*_{fsbcm}\Sigma_i$ are both defined, then $\mathcal{R}^*_{gsbcm}\Sigma_i^G \cong \mathcal{R}^*_{fsbcm}\Sigma_i$.*

Proof: Each action rule of the GSBCM and FSBCM is expressed as the composition of the microcode functions and auxiliary functions. Each of these FSBCM functions is less defined than the corresponding GSBCM function. Thus each action rule is less defined than the corresponding GSBCM action rule and hence so is the union and transitive closure of the action rules.

By definition, if an FSBCM microcode or auxiliary function is defined on an argument, it returns the same value as the corresponding GSBCM function. Thus each action rule has this property, and hence so does the union and transitive closure of the action rules.

□

4.5 Correspondence of Final Answers

The following lemma asserts that the GSBCM and FSBCM answer functions map corresponding states to equal natural numbers.

Lemma 48 *Let* Σ_f^G *and* Σ_f *be* GSBCM *and* FSBCM *halt states respectively. If* $\Sigma_f^G \cong \Sigma_f$, *then* $A_{gsbcm}(\Sigma_f^M) = A_{fsbcm}(\Sigma_f)$.

Proof: Let v^G and v be the value registers of Σ_f^G and Σ_f respectively. Since $\Sigma_f^G \cong \Sigma_f$, we have by definition of state correspondence that v^G and v are related (i.e. $F^{desc}(v^G) = v$). By definition of F^{desc}, if $v^G = \langle \text{FIXNUM } m \rangle$ then $v = \langle \text{FIXNUM } m \rangle$ and so $A_{gsbcm}(\Sigma_f^M) = A_{fsbcm}(\Sigma_f) = m$. If v^G is not of the form $\langle \text{FIXNUM } m \rangle$ then neither is v, and so $A_{gsbcm}(\Sigma_f^M)$ and $A_{fsbcm}(\Sigma_f)$ are both undefined.

□

4.6 Correspondence of Semantics

We can now prove the main theorem of this chapter.

Theorem 49 *If the* FSBCM *operational semantics assigns a meaning to an* SBC *program, then the* GSBCM *operational semantics assigns the same meaning to the program. That is, if*

$$\mathcal{O}^{sbc}_{fsbcm}[\![\mathcal{P}]\!]$$

is defined, then

$$\mathcal{O}^{sbc}_{fsbcm}[\![\mathcal{P}]\!] = \mathcal{O}^{sbc}_{gsbcm}[\![\mathcal{P}]\!]$$

Proof: Follows immediately from Lemmas 46, 47, and 48.

□

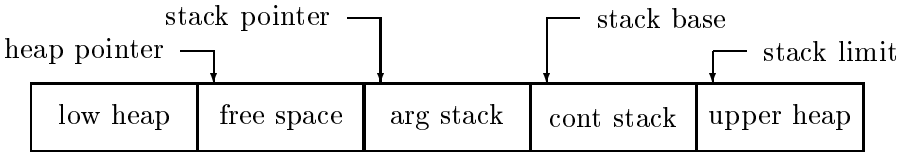

Figure 5: Memory Layout

A A New Storage Layout

The Vlisp VM currently uses two areas for heap allocation and a separate area for the argument stack. All stored objects are heap allocated. We think can verify a new organization for storage which greatly reduces the number of garbage collections.

The layout (Figure 5) uses just two equal sized areas, one of which is active at any time (except during garbage collection). All stack and heap allocation performed in the same area. All stored objects other than continuations are allocated in the low heap which grows upward. The stack grows downward and garbage collection in invoked when the stack and the low heap meet.

The new version of the MAKE-CONT instruction pushes registers on the stack, pushes a continuation header on the stack, adjusts the stack base and stack pointer to be at the newly allocated continuation, and finally updates the continuation register. The argument stack is not copied although we found this savings to be small.

The new version of RETURN does the old actions if the continuation is not in the continuation stack. Otherwise, the continuation stack is popped. This is where the real saving comes because we garbage collect many fewer continuations. When our compiler compiles itself, it now uses 90 GC's instead of 250.

Call-with-current-continuation is implemented by moving the stack limit to the current stack base. All the continuations on the stack now behave as if the were heap allocated.

The garbage collector copies the argument stack to the top of the new area, and traces the stack and the registers. After a GC, all continuations are in the low heap. One could try to preserve the continuation stack during GC, but it is too complicated for too little savings.

References

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

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

[3] Dino P. Oliva and Mitchell Wand. A verified compiler for pure PreScheme. Technical Report NU-CCS-92-5, Northeastern University College of Computer Science, February 1992.

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

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

[6] M. Wand and D. P. Oliva. Proving the correctness of storage representations. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 151–160, New York, 1992. ACM Press.