The VLISP Image Builder

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

The MITRE Corporation[1]
M92B096
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 image builder, which lays out a binary image for a program. It contains detailed proofs that the operational semantics of the resulting output matches the operational semantics of the input.

Contents

1 Introduction

This report presents and justifies the VLISP translation from linked byte code to stored byte code. A Linked Byte Code (LBC) program is translated to a sequence of simple (unstructured) terms. This sequence is called a "store" since it corresponds to the store of a state machine. The translated programs are called *Stored Byte Code* (SBC) programs. A straightforward translation is all that is needed to map SBC programs to binary images that can be loaded into the store of a physical computer and executed. The program that translates LBC programs to binary images is the *Image Builder*. However, in this report, we shall refer to the program that translates LBCprograms to SBC programs as the image builder, and shall defer discussion of the translation of SBC programs to binary images to the VLISP report on the byte-code interpreter [4].

This report addresses the following issues:

1. LBC terms are tree structured whereas SBC terms have a flat structure. The translation from LBC to SBC consists of flattening the LBC terms and representing them by linked structures within a "store" (a store is a sequence of values). For example, the LBC term ⟨CONTINUATION $t\ n\ a\ u\ k$⟩ is represented in SBC as a "pointer" (i.e., index) into a store. This pointer identifies six locations of the store whose contents correspond to the representations of the six components of the LBC term.

   Note that it is possible to create circular structures (e.g., circular lists) within a store (in the conventional way that circular linked lists are implemented). Thus, the correctness proof of this translation needs to deal with the fact that during execution of LBC and SBC programs, circular data structures may be created.

2. The memory of a physical computer is organized as a sequence of *words*, each word being a sequence of *bytes*. Some data (e.g. numbers) are represented by entire words, while others (e.g. characters and machine instruction opcodes) can be represented by bytes. The latter representation permits several characters (or opcodes) to be represented within a word, and hence results in smaller executable images. The image builder performs this storage optimization of packing several bytes into words. The correctness proof ensures that the optimization does not alter the operational semantics of programs.

This report presents:

1. the syntax and operational semantics of VLISP Stored Byte Code (SBC);
2. a correspondence relation that relates VLISP Stored Byte Code (SBC) programs and VLISP Linked Byte Code (LBC) programs;
3. a proof that the operational semantics of SBC is faithful to the operational semantics of LBC with respect to the above correspondence relation. That is, if an SBC program corresponds to an LBC program, then their respective operational semantics yield equal answers;
4. an *image builder* algorithm for translating LBC programs to SBC programs; and
5. a proof that the image builder translates LBC programs to corresponding SBC programs.

We do not reproduce the syntax and semantics of Linked Byte Code but refer the reader to the VLISP report that describes that language [1].

1.1 Notation

We will use the notation contained in Figure 1. Additional explanation of our notation can be found in the VLISP report [2].

$\langle \dots \rangle$	finite sequence formation, commas optional
$\#s$	length of sequence s
$\langle x \dots \rangle$	sequence s with $s(0) = x$
$\langle \dots x \rangle$	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 Stored Byte Code (SBC)

In this chapter, we present the syntax of the VLISP Stored Byte Code (SBC) and give it an (operational) semantics in a style very similar to the presentation of the VLISP Flattened Byte Code [3].

2.1 Syntax

The Stored Byte Code (SBC) provides an alternate (more "concrete") representation of Linked Byte Code (LBC) terms. LBC represents objects such as environments, closures, and continuations as nested lists. SBC flattens all LBC terms and represents them within a "store", where a store is a sequence of "cells" (which are defined below). An SBC term consists of a store together with an index into the store. For example, the LBC term $\langle$CONTINUATION t n a u $k\rangle$ is represented in SBC as an index ("pointer") into a store. This pointer identifies six cells ("locations") of the store that correspond to the representations of the six components of the LBC term.

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:

$$
\begin{array}{rcl}
\mathit{program} & ::= & \mathit{term} \\
\mathit{term} & ::= & \langle \mathit{store}\ \mathit{vdesc} \rangle \\
\mathit{store} & ::= & \mathit{cell}^* \\
\mathit{cell} & ::= & \mathit{desc} \mid \mathit{byte}^{\mathit{bpw}} \\
\mathit{desc} & ::= & \langle \textsc{header}\ \mathit{htag}\ \mathit{bool}\ \mathit{nat} \rangle \mid \mathit{vdesc} \\
\mathit{vdesc} & ::= & \langle \textsc{ptr}\ \mathit{loc} \rangle \mid \langle \textsc{fixnum}\ \mathit{nat} \rangle \mid \langle \textsc{imm}\ \mathit{imm} \rangle \\
\mathit{imm} & ::= & \textsc{false} \mid \textsc{true} \mid \langle \textsc{char}\ \mathit{nat} \rangle \mid \textsc{null} \mid \textsc{undefined} \\
\mathit{htag} & ::= & \mathit{bhtag} \mid \mathit{dhtag} \\
\mathit{bhtag} & ::= & \textsc{string} \mid \textsc{codevector}
\end{array}
$$

dhtag	::=	PAIR | SYMBOL | VECTOR | LOCATION | TEMPLATE
byte	::=	*nat* | `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>=` | *ident*
loc	::=	*nat*
obj	::=	*vdesc* | *byte*

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,

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

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

2.2 Semantics

In this section, we define the operational semantics of stored byte code as a deterministic state machine with concrete states. The state machine is called the *Stored Byte Code Machine (*SBCM*)*.

2.2.1 SBC State Machines (SBCM)

We define the syntax of SBC 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 Flattener report [3].

Augmented Stored Byte Code (ASBC)
The Augmented Stored Byte Code (ASBC) provides the syntactic objects that form the states of the SBCM. 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 SBCM and are also generated during the course of computation.

We distinguish between ASBC and SBC for two reasons. First, SBC is the representation language of the image builder's output. We shall identify a small collection of primitives that permit SBC programs to be constructed. The image builder shall use these primitives to manipulate SBC programs. By changing the specification and implementation of these primitives, we shall be able to change the representation language of the image builder's output.

Second, the VLISP byte code interpreter [4] is derived via a sequence of state machines, each of which is a refinement of the previous. Each state machine shall provide an alternate definition of the operational semantics of SBC programs. Thus SBC shall be a common sublanguage of each state machine, while the augmented languages of each machine shall vary.

Abstract Syntax
The new ASBC tokens are as follows:

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

The new ASBC syntactic categories are:

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

The abstract syntax of Augmented Stored Byte Code (ASBC) is defined below. This grammar redefines several syntactic categories of SBC. However, the extensions of these syntactic categories include the extensions of the SBC syntactic categories.

program	::=	*term*
term	::=	$\langle$*store vdesc*$\rangle$
store	::=	*cell**
cell	::=	*desc* | *byte*^{bpw}
desc	::=	$\langle$HEADER *htag bool nat*$\rangle$ | *vdesc*
vdesc	::=	$\langle$PTR *loc*$\rangle$ | $\langle$FIXNUM *nat*$\rangle$ | $\langle$IMM *imm*$\rangle$
imm	::=	FALSE | TRUE | $\langle$CHAR *nat*$\rangle$ | NULL | UNDEFINED
		UNSPECIFIED | EMPTY-ENV | HALT | EOF
htag	::=	*bhtag* | *dhtag*
bhtag	::=	STRING | CODEVECTOR
dhtag	::=	PAIR | SYMBOL | VECTOR | LOCATION | TEMPLATE |
		CLOSURE | PORT | CONTINUATION | ENVIRONMENT
byte	::=	*nat* | `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>=` | *ident*
loc	::=	*nat*
obj	::=	*vdesc* | *byte*
stack	::=	*vdesc**

We extend the naming conventions of SBC for variables. Thus, we will use 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 *(*ASBC*) programs*,
T	for *(*ASBC*) terms*,
s	for *(*ASBC*) stores*,
c	for *(*ASBC*) cells*,
d	for *(*ASBC*) descriptors*,
vd	for *(*ASBC*) value descriptors*,
imm	for *(*ASBC*) immediates*,
h	for *(*ASBC*) header tags*,
bh	for *(*ASBC*) byte header tags*,
dh	for *(*ASBC*) descriptor header tags*,
b	for *(*ASBC*) bytes*,
l	for *(*ASBC*) locations*,
o	for *(*ASBC*) stored object data*, and
a	for *(*ASBC*) stacks*

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

Stores

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

1. If $s(l) = \langle \textsc{header}\ h\ p\ m \rangle$ 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,
$$s(l)(1) \neq \textsc{header} \Rightarrow \\ (\exists l', h, p, m) \quad (s(l') = \langle \textsc{header}\ h\ p\ m \rangle) \wedge \\ (l' < l \leq l' + \mho m)$$
3. If $s(l) = \langle \textsc{ptr}\ l' \rangle$ for some l, l', then $s(l'-1) = \langle \textsc{header}\ h\ p\ m \rangle$ 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). Note that every SBC store is also an ASBC store.

Stored Objects

An ASBC store may 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*. 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 ASBC store s.

Let d be a pointer into ASBC 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*

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

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

Definition 3

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

The following lemma asserts that appending a stored object to an ASBC store s results in a new ASBC store s' such that $\langle \text{PTR } (\#s + 1)\rangle$ is a pointer to that stored object in s'.

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

Proof: Immediate from definitions.

□

Predicates

We define a family of predicates that characterize the types of stored objects in an ASBC store. These predicates are defined recursively and are presented in Figure 2.

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

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

States

The states of a stored byte code state machine (SBCM) are the sequences of the form

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

that satisfy the following (SBCM) state invariants:

1. s is an ASBC store
2. Template(t, s)
3. Codevector(c, s)
4. $\mathcal{S}(t, s) = \langle \text{TEMPLATE } \#f\ c::o^*\rangle$
5. Offset(n, t, s)
6. Value(v, s)

$$
\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) \\
 & & \qquad \text{or Location}(o^*(i),s) \\
 & & \qquad \text{or Template}(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{imm 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{imm 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{imm}\ imm\rangle \\
 & & \quad \text{or } \mathcal{S}(v,s) \text{ is defined}
\end{array}
$$

Figure 2: Predicates for Stored Objects

7. Environment(u, s)
8. Continuation(k, s)
9. $(\forall 0 \leq i < \#a)$ Value$(a(i), s) \vee$ Environment$(a(i), s)$

The components of a state are called, in order, its *template, offset, code-vector, value, argument stack, environment, continuation,* and *store*, and we may informally speak of them as being held in registers. The first invariant ensures that the store is an ASBC 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).

Initial and Halt States

The initial states $\langle t, n, c, v, a, u, k, s\rangle$ are the states such that

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

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

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

Actions

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.

In the remainder of this report, we shall only consider pure action rules. By abuse of notation, we shall consider $\mathcal{R}$ and $\mathcal{R}^*$ to be functions from states to states. We now present a few auxiliary functions, then present the pure action rules.

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 ASBC store s.

$$\begin{array}{l} \textit{env-lookup}(u, m, s) \stackrel{\text{def}}{=} l + m \\ \qquad \text{if } u = \langle \text{PTR } l \rangle \\ \qquad \text{and } \mathcal{S}(u, s) = \langle \text{ENVIRONMENT } \#t\ u' :: vd^* \rangle \end{array}$$

$$\begin{array}{l} \textit{env-frame}(u, 0, s) \stackrel{\text{def}}{=} u, \text{ and} \\ \textit{env-frame}(u, n+1, s) \stackrel{\text{def}}{=} \textit{env-frame}(u', n, s) \\ \qquad \textit{if } \mathcal{S}(u, s) = \langle \text{ENVIRONMENT } \#t\ u' :: vd^* \rangle \end{array}$$

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 ASBC store s. The tail of this linked list is the list pointed to by lp in ASBC store s. The function returns a pointer to the new linked list, together with the modified stack and ASBC store.

$$\begin{aligned}
&\textit{stack-to-list}(lp, 0, a, s) \stackrel{\text{def}}{=} \langle lp\ a\ s\rangle, \text{ and}\\
&\textit{stack-to-list}(lp, n+1, a, s) \stackrel{\text{def}}{=}\\
&\quad \textit{stack-to-list}(\langle \text{PTR } \#s+1\rangle, n, a \dagger 1, s \rhd \langle \text{PAIR } \#f\ \langle a(0)\ lp\rangle\rangle)
\end{aligned}$$

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

$$\begin{aligned}
&\textit{list-to-stack}(\langle \text{IMM NULL}\rangle, a, s) \stackrel{\text{def}}{=} a, \text{ and}\\
&\textit{list-to-stack}(p, a, s) \stackrel{\text{def}}{=} \textit{list-to-stack}(p', v :: a, s)\\
&\qquad\qquad\qquad\qquad \text{if } \mathcal{S}(p, s) = \langle \text{PAIR } p\ \langle v\ p'\rangle\rangle
\end{aligned}$$

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

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

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

Pure Rules

We adopt the presentation format for pure action rules that is described in the VLISP Flattener report [3]. We reproduce that description below with minor variations.

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), and a specification of the new values of some registers. Often the domain is specified by equations giving "the form" of certain registers, especially the code. In all specifications, the original values of the various registers are designated by conventional variables used exactly as in the above definition of a state: t, n, c, v, a, u, k, and s. Call these the original register variables. The new values of the registers are indicated by the same variables with primes attached: t', n', c', v', a', u', k', and s'. 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 following two auxiliary functions. Let $\mathcal{S}(c, s) = \langle \textsc{codevector}\ \#f\ b^* \rangle$ for some c, s, b^*. Then,

$$\begin{array}{rcl} \mathit{current\_inst}(n, c, s) & \stackrel{\text{def}}{=} & b^*(n) \\ \mathit{current\_inst\_param}(n, i, c, s) & \stackrel{\text{def}}{=} & b^*(n+i) \end{array}$$

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 $j + 8$ places

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

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

Rule 1: return
Domain conditions:
$\mathit{current\_inst}(n, c, s) = \texttt{return}$
$k = \textsc{halt}$
Changes:
Halt in final state Σ

Rule 2: return
Domain conditions:
$\mathit{current\_inst}(n, c, s) = \texttt{return}$
$\mathcal{S}(k, s) = \langle \textsc{continuation}\ \#f\ \langle t_1\ n_1\ u_1\ k_1 \rangle \frown a_1 \rangle$
$\mathcal{S}(t_1, s) = \langle \textsc{template}\ \#f\ c_1 :: o^* \rangle$
$n_1 = \langle \textsc{fixnum}\ m \rangle$
Changes:
$t' = t_1$
$c' = c_1$
$n' = m$
$a' = a_1$
$u' = u_1$
$k' = k_1$

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

Rule 4: jump-if-false $m_1\ m_2$
Domain conditions:
$current\_inst(n, c, s) = \texttt{jump-if-false}$
$current\_inst\_param(n, 1, c, s) = m_1$
$current\_inst\_param(n, 2, c, s) = m_2$
$v \neq \langle \textsc{imm false} \rangle$
Changes:
$n' = n + 3$

Rule 5: jump-if-false $m_1\ m_2$
Domain conditions:
$current\_inst(n, c, s) = \texttt{jump-if-false}$
$current\_inst\_param(n, 1, c, s) = m_1$
$current\_inst\_param(n, 2, c, s) = m_2$
$v = \langle \textsc{imm false} \rangle$
Changes:
$n' = n + 3 + (m_1 \otimes m_2)$

Rule 6: jump $m_1\ m_2$
Domain conditions:
$current\_inst(n, c, s) = \texttt{jump}$
$current\_inst\_param(n, 1, c, s) = m_1$
$current\_inst\_param(n, 2, c, s) = m_2$
Changes:
$n' = n + 3 + (m_1 \otimes m_2)$

Rule 7: make-cont $m_1\ m_2\ m_3$
Domain conditions:

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

Changes:

$n' = n + 4$
$a' = \langle\rangle$
$k' = \langle \textsc{ptr}\ \#s + 1 \rangle$
$s' = s \rhd \langle \textsc{continuation}\ \#f\ \langle t\ \langle \textsc{fixnum}\ m \rangle\ u\ k \rangle \frown a \rangle$

Rule 8: literal m
Domain conditions:

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

Changes:

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

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:

$n' = n + 2$
$v' = \langle \textsc{ptr}\ (\#s + 1) \rangle$
$s' = s \rhd \langle \textsc{closure}\ \#f\ \langle o_m\ u\ \langle \textsc{imm unspecified} \rangle \rangle \rangle$

Rule 10**: global** m
Domain conditions:

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

Changes:

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

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

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

Changes:

$n' = n + 2$
$s' = s[l \mapsto v]$
$v' = \langle \textsc{imm unspecified} \rangle$

Rule 12**: local** $m_1\ m_2$
Domain conditions:

$current\_inst(n, c, s) = \texttt{local}$
$current\_inst\_param(n, 1, c, s) = m_1$
$current\_inst\_param(n, 2, c, s) = m_2$
$l = \mathit{env\text{-}lookup}(\mathit{env\text{-}frame}(u, m_1, s), m_2, s)$
$v_1 = s(l)$
$v_1 \neq \langle \textsc{imm undefined} \rangle$

Changes:

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

Rule 13: set-local! m_1 m_2
Domain conditions:

$current\_inst(n, c, s) = \texttt{set-local!}$
$current\_inst\_param(n, 1, c, s) = m_1$
$current\_inst\_param(n, 2, c, s) = m_2$
$l = env\text{-}lookup(env\text{-}frame(u, m_1, s), m_2, s)$

Changes:

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

Rule 14: push
Domain conditions:

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

Changes:

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

Rule 15: make-env m
Domain conditions:

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

Changes:

$n' = n + 2$
$a' = \langle\rangle$
$u' = \langle \text{PTR } (\#s + 1) \rangle$
$s' = s \triangleright \langle \text{ENVIRONMENT } \#t\ u::a \rangle$

Rule 16**: make-rest-list** m
Domain conditions:
$current\_inst(n, c, s) =$ `make-rest-list`
$current\_inst\_param(n, 1, c, s) = m$
$\#a \geq m$
$\langle v_1\ a_1\ s_1 \rangle = stack\text{-}to\text{-}list(\langle \text{IMM NULL} \rangle, (\#a - m), a, s)$
Changes:
$n' = n + 2$
$v' = v_1$
$a' = a_1$
$s' = s_1$

Rule 17**: unspecified**
Domain conditions:
$current\_inst(n, c, s) =$ `unspecified`
Changes:
$n' = n + 1$
$v' = \langle \text{IMM UNSPECIFIED} \rangle$

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:
$n' = n + 2$

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:
$n' = n + 2$

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:

$n' = n + 1$

$k' = v$

2.2.2 Operational Semantics of SBC Programs

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

$$L_{sbcm}(\langle s\ t \rangle) \stackrel{\text{def}}{=} \langle t, 0, c, \langle \text{IMM UNSPECIFIED} \rangle, \langle \rangle, \langle \text{IMM EMPTY-ENV} \rangle, \langle \text{IMM HALT} \rangle, s \rangle \text{ if } \mathcal{S}(t, s) = \langle \text{TEMPLATE } \#f\ c :: o^* \rangle.$$

Note that SBCM initial states are ASBC terms, and hence the loader involves coercing SBC terms to ASBC terms. This coercion is trivial since every SBC term is also an ASBC term. The SBCM *Answer* A_{sbcm} is defined to be a partial function that maps SBCM halt states to natural numbers such that:

$$A_{sbcm}(\langle t, n, c, v, a, u, k, s \rangle) \stackrel{\text{def}}{=} m \quad \text{if } v = \langle \text{FIXNUM } m \rangle$$

Definition 5 *Let $\mathcal{P}$ be an* SBC *program. Then $L_{sbcm}(\mathcal{P})$ is an initial state of the* SBCM*. If $\mathcal{R}^*_{sbcm}(L_{sbcm}(\mathcal{P}))$ is defined, then the meaning of program $\mathcal{P}$ as given by the* SBC *operational semantics is*

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

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

3 Relating SBC and LBC

In this chapter, we define a correspondence relation between VLISP Stored Byte Code (SBC) programs and VLISP Linked Byte Code (LBC) programs. We prove that the operational semantics of VLISP Stored Byte Code (SBC) is faithful to the operational semantics of VLISP Linked Byte Code (LBC) with respect to this correspondence relation. That is, if an SBC program corresponds to an LBC program, then their respective operational semantics yield equal answers.

We adopt the concept and notation of storage relations [5] in defining program, term, and state correspondence relations. However, since we need to relate structures that may be circular, we modify their technique so that we can prove the correctness of storage representations for circular structures. We cannot merely take the greatest fixpoint of relations defined in the style of Oliva and Wand, since primitives like the Scheme standard procedure "`eq?`" distinguish between objects that have identical structure but are represented at different locations within the store.

3.1 Relating Syntax

Let $p = \langle n$ `constants`$::c^*$ `global-variables`$::n^* \rangle \frown t^*$ be an LBC program, where c^*, n^*, and t^* are LBC constant, global, and template tables respectively, and n is an index into t^* ($t^*(n)$ represents the "root" template of the program). We shall use an alternate syntax for LBC programs in this report and shall represent p by $\langle m^L\ \eta_c^L\ \eta_g^L\ \eta_t^L \rangle$, where $m^L = n$, $\eta_c^L = c^*$, $\eta_g^L = n^*$, and $\eta_t^L = t^*$. Let s^L be a sequence of length $\#s^L = \#\eta_g^L$.

Let $\langle s\ vd \rangle$ be an SBC program where s is an SBC store and vd is a pointer to a template object in the store s. vd represents the "root" template of the program.

We define a relation $\simeq^p$ (called a *program correspondence*) that relates LBC and SBC programs by relating constants, globals, and templates by another relation (called a *term correspondence*). $\simeq^p$ only relates programs that have the same semantics. Now the semantics of LBC considers the globals in η_g^L to be distinct from each other. Thus, the term correspondence relation needs to be one-to-one between the globals of related LBC and SBC programs. This is formalized by defining a 1-to-1 relation between a subset of $dom(s^L)$ and a subset of $dom(s)$. Each LBC global is associated with a unique location in $dom(s^L)$. Similarly, each SBC global is associated with a unique location in $dom(s)$. The term correspondence relation relates globals only if

their associated locations are related by the 1-to-1 location correspondence.

If $\simeq$ is a four-place relation, we write $(s^L, s \vdash x^L \simeq x)$ to mean that (s^L, x^L, s, x) is in the relation $\simeq$. We first define a relation between locations in s^L and locations in s. This relation relates locations that are associated with globals.

Definition 6 A *location correspondence* $\simeq_0 \subseteq (s^L \times l^L) \times (s \times l)$ is defined to be a relation such that for all s^L, s, the set $\{(l^L, l) \mid (s^L, s \vdash l^L \simeq_0 l)\}$ is a 1-to-1 relation between a subset of $dom(s^L)$ and a subset of $dom(s)$. Further, if $(s^L, s \vdash l^L \simeq_0 l)$ then $s(l-1) = \langle \text{HEADER LOCATION } \#t\ 2 \rangle$.

We extend the relation between locations to a relation between LBC and SBC terms.

Definition 7 Let $\simeq_0 \subseteq (s^L \times l^L) \times (s \times l)$ be a location correspondence relation. Let η_c^L, η_g^L, and η_t^L be LBC constant, global, and template tables respectively. A *term correspondence* $\simeq \subseteq (s^L \times (t^L + w^L + o^L + r^L + v^L)) \times (s \times vd)$ is defined as follows. The definition is by induction on the size of the second argument, i.e., the LBC term. Notationally, this is the first term after the turnstile.

1. $(s^L, s \vdash t^L \simeq t)$ if
   (a) $\mathcal{S}(t, s) = \langle \text{TEMPLATE } \#f\ c{::}o^* \rangle$
   (b) $t^L = \langle \texttt{template}\ w^L\ e^L \rangle$
   (c) $(s^L, s \vdash w^L \simeq c)$
   (d) $\#e^L = \#(c{::}o^*)$
   (e) $(\forall 0 \leq i < \#o^*)\ (s^L, s \vdash (e^L \dagger 1)(i) \simeq o^*(i))$
2. $(s^L, s \vdash w^L \simeq c)$ if
   (a) $\mathcal{S}(c, s) = \langle \text{CODEVECTOR } \#f\ b^* \rangle$
   (b) $w^L = b^*$
3. $(s^L, s \vdash o^L \simeq d)$ if
   - (a) $o^L = \langle \texttt{constant}\ i \rangle$
     (b) $\eta_c(i) = c^L$
     (c) $(s^L, s \vdash c^L \simeq d)$

- (a) $o^L = \langle \texttt{global-variable}\ i \rangle$
  (b) $r^L = \eta_c^L(\eta_g^L(i))$
  (c) $l^L = i$
  (d) $d = \langle \textsc{ptr}\ l \rangle$
  (e) $\mathcal{S}(d, s) = \langle \textsc{location}\ \#\ \langle v_1\ v_2 \rangle \rangle$
  (f) $(s^L, s \vdash l^L \simeq_0 l)$
  (g) $(s^L, s \vdash r^L \simeq v_2)$
- (a) $o^L = \langle \texttt{template}\ i \rangle$
  (b) $\eta_t(i) = t^L$
  (c) $(s^L, s \vdash t^L \simeq d)$

4. $(s^L, s \vdash r^L \simeq d)$ if
   (a) $\mathcal{S}(d, s) = \langle \textsc{symbol}\ \#f\ \langle d' \rangle \rangle$
   (b) $\mathcal{S}(d', s) = \langle \textsc{string}\ \#f\ b^* \rangle$
   (c) $\texttt{char}{\rightarrow}\texttt{int}^*(\texttt{symbol}{\rightarrow}\texttt{string}(r^L)) = b^*$
5. $(s^L, s \vdash v^L \simeq v)$ if
   - (a) $v^L = \langle \texttt{immutable-pair}\ m_1^L\ m_2^L \rangle$
     (b) $\mathcal{S}(v, s) = \langle \textsc{pair}\ \#f\ \langle v_1\ v_2 \rangle \rangle$
     (c) $(s^L, s \vdash \eta_c^L(m_1^L) \simeq v_1)$
     (d) $(s^L, s \vdash \eta_c^L(m_2^L) \simeq v_2)$
   - (a) $v^L = ch^{L*}$
     (b) $\mathcal{S}(v, s) = \langle \textsc{string}\ \#f\ b^* \rangle$
     (c) $\texttt{char}{\rightarrow}\texttt{int}^*(ch^{L*}) = b^*$
   - (a) $v^L = \texttt{immutable-vector}::m^{L*}$
     (b) $\mathcal{S}(v, s) = \langle \textsc{vector}\ \#f\ v'^* \rangle$
     (c) $\#m^{L*} = \#v'^*$
     (d) $(\forall 0 \leq i < \#m^{L*})\ (s^L, s \vdash \eta_c^L(m^{L*}(i)) \simeq v'^*(i))$
   - (a) $v^L = m$ (a number)
     (b) $v = \langle \textsc{fixnum}\ m \rangle$
   - (a) $v^L = \#f$
     (b) $v = \langle \textsc{imm false} \rangle$
   - (a) $v^L = \#t$

(b) $v = \langle \text{IMM TRUE} \rangle$

- (a) $v^L = ch$ (a character)
  (b) $v = \langle \text{IMM } \langle \text{CHAR } \texttt{char}\rightarrow\texttt{int}(ch) \rangle \rangle$
- (a) $v^L = \texttt{nil}$
  (b) $v = \langle \text{IMM NULL} \rangle$

We define an LBC program to be related to an SBC program if their root templates are related by a term correspondence:

Definition 8 *A* program correspondence relation $\simeq^p \subset$ (LBC programs $\times$ SBC programs) *is defined as follows:*

$$\langle m^L\ \eta_c^L\ \eta_g^L\ \eta_t^L \rangle \simeq^p \langle s\ vd \rangle$$

if there exists a location correspondence $\simeq_0$*, and a term correspondence* $\simeq$ *induced by* $\simeq_0$*, such that* $(s^L, s \vdash \eta_t^L(m^L) \simeq vd)$ *where* $s^L = \text{UNDEFINED}^*$ *and* $\#s^L = \#\eta_g^L$*.*

3.2 Relating Semantics

We prove that programs related by the program correspondence relation have the same semantics. We proceed by first extending the program correspondence relation to a correspondence relation between LBCM and SBCM states. We then show that the LBCM and SBCM Loaders map related programs to related states. We prove that the LBCM and SBCM action rules preserve the state correspondence relation, and that the final answer functions map related states to the same natural number.

3.2.1 State Correspondence Relation

In Section 3.1, we presented location, term, and program correspondence relations between LBC and SBC locations, terms, and programs respectively. We extend these to correspondence relations between ALBC and ASBC locations, terms, and states.

Definition 9 An *augmented location correspondence* $\simeq_0 \subseteq (s^L \times l^L) \times (s \times l)$ is defined to be a relation such that for all s^L, s, the set $\{(l^L, l) \mid (s^L, s \vdash l^L \simeq_0 l)\}$ is a 1-to-1 relation between a subset of $dom(s^L)$ and a subset of $dom(s)$; further, if $(s^L, s \vdash l^L \simeq_0 l)$ then there exists an $l' \in \text{dom}(s)$ such that $s(l'-1) = \langle \text{HEADER } h\ p\ m \rangle$ and one of the following hold:

- $h = \textsc{location}$ and $l = l'$
- $h = \textsc{pair}$ and $l' \leq l < l' + 2$
- $h = \textsc{string}$ and $l = l'$
- $h = \textsc{vector}$ and $l' \leq l < l' + \mho m$
- $h = \textsc{closure}$ and $l = l' + 2$
- $h = \textsc{environment}$ and $l' + 1 \leq l < l' + \mho m$

Definition 10 Let $\simeq_0 \subseteq (s^L \times l^L) \times (s \times l)$ be an augmented location correspondence relation. An *augmented term correspondence* $\simeq \subseteq (s^L \times (t^L + w^L + o^L + i^L + v^L + u^L + a^L + k^L)) \times (s \times vd)$ is defined as follows. The definition is by induction on the size of the second argument, i.e., the ALBC term. Notationally, this is the first term after the turnstile. Note that clauses 1–4 and the first eight subclauses of clause 5 are identical to those in the definition of a term correspondence.

1. $(s^L, s \vdash t^L \simeq t)$ if
   (a) $\mathcal{S}(t, s) = \langle \textsc{template}\ \#f\ c{::}o^* \rangle$
   (b) $t^L = \langle \texttt{template}\ w^L\ e^L \rangle$
   (c) $(s^L, s \vdash w^L \simeq c)$
   (d) $\#e^L = \#(c{::}o^*)$
   (e) $(\forall 0 \leq i < \#o^*)\ (s^L, s \vdash (e^L \dagger 1)(i) \simeq o^*(i))$
2. $(s^L, s \vdash w^L \simeq c)$ if
   (a) $\mathcal{S}(c, s) = \langle \textsc{codevector}\ \#f\ b^* \rangle$
   (b) $w^L = b^*$
3. $(s^L, s \vdash o^L \simeq d)$ if
   - (a) $o^L = \langle \texttt{constant}\ i \rangle$
     (b) $\eta_c(i) = c^L$
     (c) $(s^L, s \vdash c^L \simeq d)$
   - (a) $o^L = \langle \texttt{global-variable}\ i \rangle$
     (b) $r^L = \eta_c^L(\eta_g^L(i))$

(c) $l^L = i$
(d) $d = \langle \textsc{ptr}\ l \rangle$
(e) $\mathcal{S}(d, s) = \langle \textsc{location}\ \#\ \langle v_1\ v_2 \rangle \rangle$
(f) $(s^L, s \vdash l^L \simeq_0 l)$
(g) $(s^L, s \vdash r^L \simeq v_2)$

- (a) $o^L = \langle \texttt{template}\ i \rangle$
  (b) $\eta_t(i) = t^L$
  (c) $(s^L, s \vdash t^L \simeq d)$

4. $(s^L, s \vdash r^L \simeq d)$ if

   (a) $\mathcal{S}(d, s) = \langle \textsc{symbol}\ \#f\ \langle d' \rangle \rangle$
   (b) $\mathcal{S}(d', s) = \langle \textsc{string}\ \#f\ b^* \rangle$
   (c) $\texttt{char}{\rightarrow}\texttt{int}^*(\texttt{symbol}{\rightarrow}\texttt{string}(r^L)) = b^*$

5. $(s^L, s \vdash v^L \simeq v)$ if

   - (a) $v^L = \langle \texttt{immutable-pair}\ m_1^L\ m_2^L \rangle$
     (b) $\mathcal{S}(v, s) = \langle \textsc{pair}\ \#f\ \langle v_1\ v_2 \rangle \rangle$
     (c) $(s^L, s \vdash \eta_c^L(m_1^L) \simeq v_1)$
     (d) $(s^L, s \vdash \eta_c^L(m_2^L) \simeq v_2)$
   - (a) $v^L = ch^{L*}$
     (b) $\mathcal{S}(v, s) = \langle \textsc{string}\ \#f\ b^* \rangle$
     (c) $\texttt{char}{\rightarrow}\texttt{int}^*(ch^{L*}) = b^*$
   - (a) $v^L = \texttt{immutable-vector}::m^{L*}$
     (b) $\mathcal{S}(v, s) = \langle \textsc{vector}\ \#f\ v'^* \rangle$
     (c) $\#m^{L*} = \#v'^*$
     (d) $(\forall 0 \leq i < \#m^{L*})\ (s^L, s \vdash \eta_c^L(m^{L*}(i)) \simeq v'^*(i))$
   - (a) $v^L = m$ (a number)
     (b) $v = \langle \textsc{fixnum}\ m \rangle$
   - (a) $v^L = \#f$
     (b) $v = \langle \textsc{imm false} \rangle$
   - (a) $v^L = \#t$
     (b) $v = \langle \textsc{imm true} \rangle$
   - (a) $v^L = ch$ (a character)

(b) $v = \langle \text{IMM } \langle \text{CHAR } \texttt{char}{\rightarrow}\texttt{int}(ch) \rangle \rangle$

- (a) $v^L = \texttt{nil}$
  (b) $v = \langle \text{IMM NULL} \rangle$
- (a) $v^L = \langle \text{CLOSURE } t^L \ u^L \ l^L \rangle$
  (b) $v = \langle \text{PTR } l \rangle$
  (c) $\mathcal{S}(v, s) = \langle \text{CLOSURE } \#f \ \langle t \ u \ \langle \text{IMM UNSPECIFIED} \rangle \rangle \rangle$
  (d) $(s^L, s \vdash t^L \simeq t)$
  (e) $(s^L, s \vdash u^L \simeq u)$
  (f) $(s^L, s \vdash l^L \simeq_0 l + 2)$
- (a) $v^L = \langle \text{MUTABLE-PAIR } l_1^L \ l_2^L \rangle$
  (b) $v = \langle \text{PTR } l \rangle$
  (c) $\mathcal{S}(v, s) = \langle \text{PAIR } \#t \ \langle v_1 \ v_2 \rangle \rangle$
  (d) $(s^L, s \vdash l_1^L \simeq_0 l)$
  (e) $(s^L, s \vdash l_2^L \simeq_0 l + 1)$
- (a) $v^L = \langle \text{MUTABLE-STRING } l^{L*} \rangle$
  (b) $v = \langle \text{PTR } l \rangle$
  (c) $\mathcal{S}(v, s) = \langle \text{STRING } \#t \ b^* \rangle$
  (d) $\#l^{L*} = \#b^*$
  (e) $(\forall 0 \leq i < \#l^{L*}) \ \texttt{char}{\rightarrow}\texttt{int}(s^L(l^{L*}(i))) = b^*(i)$
  (f) $(s^L, s \vdash l^{L*}(0) \simeq_0 l)$
  (g) for all $1 \leq i < \#l^{L*}$, the location $l^{L*}(i)$ is not related to any location l' by $\simeq_0$.
- (a) $v^L = \langle \text{MUTABLE-VECTOR } l^{L*} \rangle$
  (b) $v = \langle \text{PTR } l \rangle$
  (c) $\mathcal{S}(v, s) = \langle \text{VECTOR } \#t \ v'^* \rangle$
  (d) $\#l^{L*} = \#v'^*$
  (e) $(\forall 0 \leq i < \#l^{L*}) \ (s^L, s \vdash l^{L*}(i) \simeq_0 l + i)$
- (a) $v^L = \text{UNSPECIFIED}$
  (b) $v = \langle \text{IMM UNSPECIFIED} \rangle$
- (a) $v^L = \text{UNDEFINED}$
  (b) $v = \langle \text{IMM UNDEFINED} \rangle$
- (a) $v^L = \text{EOF}$
  (b) $v = \langle \text{IMM EOF} \rangle$

6. $(u^L, s^L) \simeq (u, s)$ if
   - (a) $u^L = \text{EMPTY-ENV}$
     (b) $u = \langle \text{IMM EMPTY-ENV} \rangle$
   - (a) $u^L = \langle \text{ENV } u'^L \; l^{L*} \rangle$
     (b) $u = \langle \text{PTR } l \rangle$
     (c) $\mathcal{S}(u, s) = \langle \text{ENVIRONMENT } \#t \; u' :: v^* \rangle$
     (d) $(s^L, s \vdash u'^L \simeq u')$
     (e) $\#l^{L*} = \#v^*$
     (f) $(\forall 0 \leq i < \#l^{L*})\ (s^L, s \vdash l^{L*}(i) \simeq_0 l + i + 1)$
7. $(s^L, s \vdash k^L \simeq k)$ if
   - (a) $k^L = \text{HALT}$
     (b) $k = \langle \text{IMM HALT} \rangle$
   - (a) $k^L = \langle \text{CONT } t^L \; n^L \; a^L \; u^L \; k^L \rangle$
     (b) $\mathcal{S}(k, s) = \langle \text{CONTINUATION } \#f \; \langle t \; n \; u \; k \rangle \frown a \rangle$
     (c) $(s^L, s \vdash t^L \simeq t)$
     (d) $(s^L, s \vdash n^L \simeq n)$
     (e) $(s^L, s \vdash a^L \simeq a)$
     (f) $(s^L, s \vdash u^L \simeq u)$
     (g) $(s^L, s \vdash k^L \simeq k)$

Definition 11 The *state correspondence relation* $\cong \subset state^L \times state$ is defined as follows:

$\langle t^L, n^L, v^L, a^L, u^L, k^L, s^L \rangle \cong \langle t, n, c, v, a, u, k, s \rangle$ if there exists an augmented location correspondence $\simeq_0$, and an augmented term correspondence $\simeq$ induced by $\simeq_0$, such that

- for all $l^L \in dom(s^L)$ and $l \in dom(s)$,
  $((s^L, s \vdash l^L \simeq_0 l) \text{ and } s(l) \notin bhtag) \Rightarrow (s^L, s \vdash s^L(l^L) \simeq s(l))$
- $(s^L, s \vdash t^L \simeq t)$
- $n^L = n$
- $(s^L, s \vdash v^L \simeq v)$
- $\#a^L = \#a$ and $(\forall 0 \leq i < \#a)(s^L, s \vdash a^L(i) \simeq a(i))$
- $(s^L, s \vdash u^L \simeq u)$
- $(s^L, s \vdash k^L \simeq k)$

3.2.2 Establishing State Correspondence

The *linked byte code machine loader* L_{lbcm} is a partial function defined as follows:

$$\begin{aligned} &L_{lbcm}(\langle m^L\ \eta_c^L\ \eta_g^L\ \eta_t^L\rangle) \\ &\quad = \langle \eta_t^L(m^L), 0, \text{UNSPECIFIED}, \langle\rangle, \text{EMPTY-ENV}, \text{HALT}, s^L\rangle \\ &\quad \text{where } s^L = \text{UNDEFINED}^* \text{ and } \#s^L = \#\eta_g^L. \end{aligned}$$

The *stored byte code machine loader* L_{sbcm} is a partial function defined as follows:

$$\begin{aligned} L_{sbcm}(\langle s\ vd\rangle) \quad = \quad & \langle vd, 0, c, \langle \text{IMM UNSPECIFIED}\rangle, \langle\rangle, \\ & \qquad \langle \text{IMM EMPTY-ENV}\rangle, \langle \text{IMM HALT}\rangle, s\rangle \\ & \text{if } \mathcal{S}(vd, s) = \langle \text{TEMPLATE } \#f\ c{::}o^*\rangle. \end{aligned}$$

Lemma 12 *The* LBCM *and* SBCM *Loaders map related programs to related states. That is,*

$$p^L \simeq^p p \Rightarrow L_{lbcm}(p^L) \cong L_{sbcm}(p)$$

Proof: Let $p^L \simeq^p p$. From the definition of a program correspondence, there exists a location correspondence $\simeq_0$, and a term correspondence $\simeq$ induced by $\simeq_0$, such that $(s^L, s \vdash \eta_t^L(m^L) \simeq vd)$ where $s^L = \text{UNDEFINED}^*$ and $\#s^L = \#\eta_g^L$. We show that $\simeq_0$ and $\simeq$ are witness to the state correspondence $L_{lbcm}(p^L) \cong L_{sbcm}(p)$.

All conditions in the definition of a state correspondence hold trivially, except for the condition that related locations of the stores s^L and s should contain related values. Now, by definition of a location correspondence, an SBCM location is related to an LBCM location only if the former indexes the first data cell of a location object. But, by definition of an SBC program, such a data cell is always the value $\langle$IMM UNDEFINED$\rangle$. This is related to the value of the LBCM location, since $s^L = \text{UNDEFINED}^*$.

□

3.2.3 Preserving State Correspondence

We prove that each action rule preserves state correspondence. That is, if Σ^L and Σ are related LBCM and SBCM states respectively, then an LBCM action rule maps Σ^L to a state Σ'^L if and only if an SBCM action rule maps Σ to a state Σ'; furthermore, the states Σ'^L and Σ' are related.

The following two lemmas will be used extensively in the proof. The first lemma states that if related values are stored in related locations of stores in related states, then the resulting states are also related.

Lemma 13 Let $\simeq_0$ be an augmented location correspondence and $\simeq$ be the augmented term correspondence induced by $\simeq_0$. Let

$$\Sigma^L = \langle t^L, n^L, v^L, a^L, u^L, k^L, s^L \rangle \text{ and}$$
$$\Sigma = \langle t, n, c, v, a, u, k, s \rangle$$

be related states of the LBCM and SBCM respectively; that is, let $\Sigma^L \cong \Sigma$ with witnesses $\simeq_0$ and $\simeq$.

Let l^L, l be related locations $(s^L, s \vdash l^L \simeq_0 l)$ such that $s(l) \in \mathit{dhtag}$. Let e^L, e be related terms $(s^L, s \vdash e^L \simeq e)$. Let $s'^L = s^L[l^L \mapsto e^L]$ and $s' = s[l \mapsto e]$. Let $\simeq'_0$ be an augmented location correspondence defined such that $(s'^L, s' \vdash l^L \simeq'_0 l)$ if and only if $(s^L, s \vdash l^L \simeq_0 l)$. Let $\simeq'$ be the augmented term correspondence induced by $\simeq'_0$. Then,

1. terms related by $\simeq$ relative to s^L, s are also related by $\simeq'$ relative to s'^L, s'. That is,

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

2. If two locations are related by $\simeq'_0$, then their contents are related by $\simeq'$. That is,

$$\text{for all } l^L \in \mathit{dom}(s'^L) \text{ and } l \in \mathit{dom}(s')$$
$$((s'^L, s' \vdash l^L \simeq'_0 l) \text{ and } s(l) \notin \mathit{bhtag}) \Rightarrow (s'^L, s' \vdash s'^L(l^L) \simeq' s'(l))$$

3. $\langle t^L, n^L, v^L, a^L, u^L, k^L, s'^L \rangle \cong \langle t, n, c, v, a, u, k, s' \rangle$ is a state correspondence with witnesses $\simeq'_0$ and $\simeq'$.

Proof:

1. Since $(s^L, s \vdash l^L \simeq_0 l)$, we have from definition 9 that l is an address that lies within a stored object which represents a location, pair, string, vector, closure, or environment. Since $s(l) \in \mathit{dhtag}$ by assumption, l does not lie within a string object. By inspection, we see that the definition of an augmented term correspondence relation does not depend on the content of such a location.

Now, l^L is a location related to l by $\simeq_0$, and l does not lie within a string object. By inspection, we see that the definition of an augmented term correspondence relation does not depend on the content of such a location.

Thus, modifying the contents of l^L and l does not affect the augmented term correspondence relation $\simeq$, hence the result.

2. Let $l_1^L \in dom(s'^L)$, $l_1 \in dom(s')$, and $(s'^L, s' \vdash l_1^L \simeq_0 l_1)$. The proof is by cases:

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

   $$\begin{array}{ll}
   (s'^L, s' \vdash l_1^L \simeq'_0 l_1) & \\
   \quad \Rightarrow (s^L, s \vdash l_1^L \simeq_0 l_1) & \text{by definition of } \simeq_0 \\
   \quad \Rightarrow (s^L, s \vdash s^L(l_1^L) \simeq s(l_1)) & \text{since } \Sigma^L \cong \Sigma \text{ with witnesses} \\
   & \simeq_0 \text{ and } \simeq \\
   \quad \Rightarrow (s'^L, s' \vdash s^L(l_1^L) \simeq' s(l_1)) & \text{by part 1 of this lemma} \\
   \quad \Rightarrow (s'^L, s' \vdash s'^L(l_1^L) \simeq' s'(l_1)) & \text{by case condition and} \\
   & \text{definitions of } s'^L \text{ and } s'
   \end{array}$$

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

   $$\begin{array}{ll}
   (s^L, s \vdash e^L \simeq e) & \text{by assumption} \\
   \quad \Rightarrow (s'^L, s' \vdash e^L \simeq' e) & \text{by part 1 of this lemma} \\
   \quad \Rightarrow (s'^L, s' \vdash s'^L(l^L) \simeq' s'(l)) & \text{by definition of } s'^L \text{ and } s' \\
   \quad \Rightarrow (s'^L, s' \vdash s'^L(l_1^L) \simeq' s'(l_1)) & \text{by case condition}
   \end{array}$$

   There are no other cases by definition of $\simeq_0$ (since $\simeq_0$ is 1-to-1 on locations).

3. Follows trivially from Definition 11 and parts (1) and (2) of this lemma.

□

The following lemma states that if related values are stored in unused locations of stores in related states, then the resulting states are also related.

Lemma 14 Let $\simeq_0$ be an augmented location correspondence and $\simeq$ be the augmented term correspondence induced by $\simeq_0$. Let

$$\Sigma^L = \langle t^L, n^L, v^L, a^L, u^L, k^L, s^L \rangle \text{ and}$$
$$\Sigma = \langle t, n, c, v, a, u, k, s \rangle$$

be related states of the LBCM and SBCM respectively; that is, let $\Sigma^L \cong \Sigma$ with witnesses $\simeq_0$ and $\simeq$.

Let l^L, l be locations such that $l^L \notin dom(s^L)$ and $l \notin dom(s)$. Let e^L, e be related terms, i.e., let $(s^L, s \vdash e^L \simeq e)$. Let $s'^L = s^L[l^L \mapsto e^L]$ and $s' = s[l \mapsto e][l_1 \mapsto e_1] \ldots [l_k \mapsto e_k]$ where $l_i \notin dom(s[l \mapsto e])$ and e_i are terms for $1 \leq i \leq k$. Let $\simeq'_0$ be a new augmented location correspondence such that for all locations l_1^L, l_1, $(s'^L, s' \vdash l_1^L \simeq'_0 l_1)$ holds if either $(s^L, s \vdash l_1^L \simeq_0 l_1)$ holds or if $l_1^L = l^L$ and $l_1 = l$. That is, the relation $\{(l_1^L, l_1) \mid (s'^L, s' \vdash l_1^L \simeq'_0 l_1)\}$ extends the relation $\{(l_1^L, l_1) \mid (s^L, s \vdash l_1^L \simeq l_1)\}$ with the tuple (l^L, l). Let $\simeq'$ be the augmented term correspondence induced by $\simeq'_0$ (by Definition 10).

Then,

1. for all $x^L \in D^L, x \in D$, $(s^L, s \vdash x^L \simeq x) \Rightarrow (s'^L, s' \vdash x^L \simeq' x)$.
2. for all $l^L \in dom(s'^L)$ and $l \in dom(s')$
   $(s'^L, s' \vdash l^L \simeq'_0 l) \Rightarrow (s'^L, s' \vdash s'^L(l^L) \simeq' s'(l))$
3. $\langle t^L, n^L, v^L, a^L, u^L, k^L, s'^L \rangle \cong \langle t, n, c, v, a, u, k, s' \rangle$ with witnesses $\simeq_0$ and $\simeq$.

Proof:

1. $\simeq$ and $\simeq'$ are obtained as the transitive closures of the relations $\simeq_0$ and $\simeq'_0$ respectively, under the same set of closure rules. But, by definition of $\simeq'_0$, $(s^L, s \vdash l_1^L \simeq_0 l_1)$ implies $(s'^L, s' \vdash l_1^L \simeq'_0 l_1)$. Further, if $(s^L, s \vdash l_1^L \simeq_0 l_1)$ then $s^L(l_1^L) = s'^L(l_1^L)$ and $s(l_1) = s'(l_1)$. The result thus follows (by induction over the set of closure rules).

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

   Case $(l_1^L \neq l^L)$ **and** $(l_1 \neq l)$**:** Since $s'^L = s^L[l^L \mapsto e^L]$, we have that $l_1^L \in dom(s^L)$ and $s'^L(l_1^L) = s^L(l_1^L)$. Now, $\simeq_0$ relates all locations in $dom(s^L)$ to locations in $dom(s)$. Since $\simeq'_0$ extends $\simeq_0$ and is 1-to-1 on locations, $l_1 \in dom(s)$ and hence $s'(l_1) = s(l_1)$. Now,

   $$\begin{array}{ll} (s'^L, s' \vdash l_1^L \simeq'_0 l_1) & \\ \quad \Rightarrow (s^L, s \vdash l_1^L \simeq_0 l_1) & \text{by definition of } \simeq'_0 \\ \quad \Rightarrow (s^L, s \vdash s^L(l_1^L) \simeq s(l_1)) & \text{since } \Sigma^L \cong \Sigma \\ \quad \Rightarrow (s^L, s \vdash s'^L(l_1^L) \simeq s'(l_1)) & \text{since } s'^L(l_1^L) = s^L(l_1^L) \\ & \text{and } s'(l_1) = s(l_1) \\ \quad \Rightarrow (s'^L, s' \vdash s'^L(l_1^L) \simeq' s'(l_1)) & \text{by part 1 of this lemma} \end{array}$$

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

$$
\begin{array}{ll}
(s^L, s \vdash e^L \simeq e) & \\
\quad \Rightarrow (s'^L, s' \vdash e^L \simeq' e) & \text{by part 1 of this lemma} \\
\quad \Rightarrow (s'^L, s' \vdash s'^L(l^L) \simeq' s'(l)) & \text{by definition of } s'^L \text{ and } s' \\
\quad \Rightarrow (s'^L, s' \vdash s'^L(l_1^L) \simeq' s'(l_1)) & \text{by case condition}
\end{array}
$$

There are no other cases by definition of $\simeq_0'$.

3. Follows trivially from Definition 11 and parts (1) and (2) of this lemma.

□

Lemma 15 *Let* $\simeq_0$ *be an augmented location correspondence and* $\simeq$ *be the augmented term correspondence induced by* $\simeq_0$*. Let*

$$\Sigma^L = \langle t^L, n^L, v^L, a^L, u^L, k^L, s^L \rangle \text{ and}$$
$$\Sigma = \langle t, n, c, v, a, u, k, s \rangle$$

be related states of the LBCM *and* SBCM *respectively; that is, let* $\Sigma^L \cong \Sigma$ *with witnesses* $\simeq_0$ *and* $\simeq$*. Then,*

$$\mathcal{R}^L(\Sigma^L) \cong \mathcal{R}(\Sigma)$$

Proof: We prove this in the following subsections by considering each action rule in turn. We classify the action rules into three categories: rules that do not mutate the store, rules that mutate the store by altering its values at already defined locations, and rules that mutate the store by augmenting the defined domain of the store. In this report, we only present proofs for representative rules in each category. Proofs of all action rules are detailed in a separate manuscript, and closely follow the representative proofs presented here.

□

Correspondence of Auxiliary Functions

Lemma 16 Let $(s^L, s \vdash u^L \simeq u)$. Then,

$(s^L, s \vdash \textit{env-reference}(u^L, m_1, m_2) \simeq_0 \textit{env-lookup}(\textit{env-frame}(u, m_1, s), m_2, s))$

Proof: The proof is by induction on m_1.

Case $m_1 = 0$**:** By definition,

$$\begin{array}{l} \mathit{env\text{-}reference}(u^L, 0, m_2) \\ \quad = l^*(m_2 - 1) \\ \quad \text{if } u^L = \langle \textsc{env}\ u'^L\ l^* \rangle \text{ and } 0 \leq (m_2 - 1) < \#l^*. \end{array}$$

$$\begin{array}{l} \mathit{env\text{-}lookup}(\mathit{env\text{-}frame}(u, 0, s), m_2, s) \\ \quad = \mathit{env\text{-}lookup}(u, m_2, s) = l + m_2 \\ \quad \text{if } u = \langle \textsc{ptr}\ l \rangle,\ \mathcal{S}(u, s) = \langle \textsc{environment}\ \#t\ u' :: vd^* \rangle, \\ \quad \text{and } 1 \leq m_2 < \#(u' :: vd^*) \end{array}$$

Since $(s^L, s \vdash u^L \simeq u)$, we have by definition of $\simeq$ (Definition 10) that $(s^L, s \vdash l^*(m_2 - 1) \simeq_0 l + (m_2 - 1) + 1)$ as desired.

Case $m_1 > 0$**:** By definition,

$$\begin{array}{l} \mathit{env\text{-}reference}(u^L, m_1, m_2) \\ \quad = \mathit{env\text{-}reference}(u'^L, m_1 - 1, m_2) \\ \quad \text{if } u^L = \langle \textsc{env}\ u'^L\ l^* \rangle \end{array}$$

$$\begin{array}{l} \mathit{env\text{-}lookup}(\mathit{env\text{-}frame}(u, m_1, s), m_2, s) \\ \quad = \mathit{env\text{-}lookup}(\mathit{env\text{-}frame}(u', m_1 - 1, s), m_2, s) \\ \quad \text{if } \mathcal{S}(u, s) = \langle \textsc{environment}\ \#t\ u' :: vd^* \rangle \end{array}$$

Since $(s^L, s \vdash u^L \simeq u)$, we have by definition that $(s^L, s \vdash u'^L \simeq u')$. The result then follows by inductive hypothesis since $m_1 - 1 < m_1$.

□

Correspondence of Actions

Rule 1: call m

Changes:

$$\begin{array}{rcl} \Sigma'^L & = & \Sigma^L[t'^L = t_1^L][n'^L = 0][u'^L = u_1^L] \\ & & \text{where } v^L = \langle \textsc{closure}\ \#f\ \langle t_1^L\ u_1^L\ l_1^L \rangle \rangle \\ \Sigma' & = & \Sigma[t' = t_1][u' = u_1][c' = c_1][n' = 0] \\ & & \text{where } \mathcal{S}(v, s) = \langle \textsc{closure}\ \#f\ \langle t_1\ u_1\ d_1 \rangle \rangle \\ & & \text{and } \mathcal{S}(t_1, s) = \langle \textsc{template}\ \#f\ c_1 :: o^* \rangle \end{array}$$

Proof Obligations:

1. $(s^L, s \vdash t_1^L \simeq t_1)$ which follows by definition since $(s^L, s \vdash v^L \simeq v)$
2. $(s^L, s \vdash u_1^L \simeq u_1)$ which follows by definition since $(s^L, s \vdash v^L \simeq v)$
3. $0 = 0$

Rule 2: set-local! $m_1\ m_2$

Changes:

$$\begin{array}{rcl} \Sigma'^L & = & \Sigma^L[n'^L = n^L + 3][s'^L = s^L[l^L \mapsto v^L]][v'^L = \textsc{unspecified}] \\ & & \text{where } l^L = \textit{env-reference}(u^L, m_1, m_2) \\ \Sigma' & = & \Sigma[n' = n + 3][s' = s[l \mapsto v]][v' = \langle \textsc{imm unspecified} \rangle] \\ & & \text{where } l = \textit{env-lookup}(\textit{env-frame}(u, m_1, s), m_2, s) \end{array}$$

Proof Obligations:

By assumption, $\Sigma^L \cong \Sigma$. Let $\simeq_0$ and $\simeq$ be the location and term correspondences that are witness to the state correspondence $\Sigma^L \cong \Sigma$.

Let $s'^L = s^L[l^L \mapsto v^L]$ and $s' = s[l \mapsto v]$. Now, $(s^L, s \vdash v^L \simeq v)$ holds since $\Sigma^L \cong \Sigma$. $(s^L, s \vdash l^L \simeq_0 l)$ follows from Lemma 16. We can thus use Lemma 13 to prove the proof obligations. By Lemma 13(3),

$$\Sigma^L[s'^L = s^L[l^L \mapsto v^L]] \cong \Sigma[s' = s[l \mapsto v]]$$

with location correspondence $\simeq_0'$ and term correspondence $\simeq'$ as witnesses.

Then, $\Sigma'^L \cong \Sigma'$ holds with witnesses $\simeq_0'$ and $\simeq'$ since:

1. for all $l_1^L \in \textit{dom}(s'^L)$ and $l_1 \in \textit{dom}(s')$,
   $(s'^L, s' \vdash l_1^L \simeq_0' l_1) \Rightarrow (s'^L, s' \vdash s'^L(l_1^L) \simeq' s'(l_1))$
   This follows from Lemma 13.

2. $(s'^L, s' \vdash t^L \simeq' t)$ which follows from Lemma 13.

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

4. $(s'^L, s' \vdash v'^L \simeq' v')$ where we have that $v'^L = \text{UNSPECIFIED}$ and $v' = \langle \text{IMM UNSPECIFIED} \rangle$; this holds by definition of a term correspondence.

5. $\#a^L = \#a$ and $(\forall 0 \leq i < \#a)(s'^L, s' \vdash a^L(i) \simeq' a(i))$ which follows from Lemma 13.

6. $(s'^L, s' \vdash u^L \simeq' u)$ which follows from Lemma 13.

7. $(s'^L, s' \vdash k^L \simeq' k)$ which follows from Lemma 13.

Rule 3: closure m

Changes:

$$
\begin{array}{rcl}
\Sigma'^L & = & \Sigma^L[n'^L = n^L + 2] \\
 & & \quad [v'^L = \langle \text{CLOSURE } \eta_t(i)\ u^L\ \#s^L \rangle] \\
 & & \quad [s'^L = s^L \frown \langle \text{UNSPECIFIED} \rangle] \\
 & & \text{where } t^L = \langle \text{TEMPLATE } b^L\ e^L \rangle \\
 & & \text{and } e^L(m) = \langle \texttt{template}\ i \rangle \\
\Sigma' & = & \Sigma[s' = s \rhd \langle \text{CLOSURE } \#f\ \langle o_m\ u\ d^u \rangle \rangle] \\
 & & \quad [v' = \langle \text{PTR } \#s + 1 \rangle] \\
 & & \quad [n' = n + 2] \\
 & & \text{where } \mathcal{S}(t, s) = \langle \text{TEMPLATE } \#f\ c :: o^* \rangle \\
 & & \text{and } 1 \leq m \leq \#o^* \\
 & & \text{and } o_m = (c :: o^*)(m) \\
 & & \text{and } \mathcal{S}(o_m, s) = \langle \text{TEMPLATE } \#f\ c_1 :: o'^* \rangle \\
 & & \text{and } d^u = \langle \text{IMM UNSPECIFIED} \rangle
\end{array}
$$

Proof Obligations:

Let $\simeq_0$ and $\simeq$ be the location and augmented term correspondences that are witnesses to the state correspondence $\Sigma^L \cong \Sigma$. Let $\simeq'_0$ be a new location correspondence such that for all locations l_1^L, l_1, $(s'^L, s' \vdash l_1^L \simeq'_0 l_1)$ holds if either $(s^L, s \vdash l_1^L \simeq_0 l_1)$ holds, or if $l_1^L = \#s^L$ and $l_1 = \#s + 3$. Let $\simeq'$ be the augmented term correspondence induced by $\simeq_0$ (by Definition 10).

Let $e^L = \text{UNSPECIFIED}$ and $e = \langle \text{IMM UNSPECIFIED} \rangle$. Then, by definition, $(s^L, s \vdash e^L \simeq e)$. Let

$$\begin{array}{rcl} s'^L & = & s^L[l_1^L \mapsto e^L] \\ s' & = & s[\#s \mapsto \langle \text{HEADER CLOSURE } \#f\ 3 \rangle] \\ & & [\#s+1 \mapsto o_m] \\ & & [\#s+2 \mapsto u] \\ & & [l_1 \mapsto e] \end{array}$$

That is, let $s'^L = s^L \frown \langle e^L \rangle$ and $s' = s \rhd \langle \text{CLOSURE } \#f\ \langle o_m\ u\ e \rangle \rangle$.

We show that $\simeq'$ is witness to the state correspondence $\Sigma'^L \cong \Sigma'$. The above conditions permit us to use Lemma 14 in proving the obligations.

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

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

3. $n'^L = n'$ where $n'^L = n^L + 2$ and $n' = n + 2$; this holds since $n^L = n$ by the assumption that $\Sigma^L \cong \Sigma$.

4. $(s'^L, s' \vdash v'^L \simeq' v')$ where $v'^L = \langle \text{CLOSURE } \eta_t(i)\ u^L\ \#s^L \rangle$ and $v' = \langle \text{PTR } \#s+1 \rangle$.

   $\mathcal{S}(v', s') = \langle \text{CLOSURE } \#f\ \langle o_m\ u\ \langle \text{IMM UNSPECIFIED} \rangle \rangle \rangle$ by Lemma 4. Since $(s^L, s \vdash t^L \simeq t)$ by assumption, we have by definition of $\simeq$ that $(s^L, s \vdash \eta_t(i) \simeq o_m)$; hence, by Lemma 14, $(s'^L, s' \vdash \eta_t(i) \simeq' o_m)$. Since $(s^L, s \vdash u^L \simeq u)$ by assumption, we have by Lemma 14 that $(s'^L, s' \vdash u^L \simeq' u)$. By definition of $\simeq'_0$ we have that $(s'^L, s' \vdash \#s^L \simeq'_0 \#s+3)$. The result then holds by definition of $\simeq$.

5. $\#a^L = \#a$ and $(\forall 0 \leq i < \#a)(s'^L, s' \vdash a^L(i) \simeq' a(i))$ which follows from Lemma 14.

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

7. $(s'^L, s' \vdash k^L \simeq' k)$ which follows from Lemma 14.

Lemma 17 *Let Σ^L and Σ be* LBCM *and* SBCM *states respectively. If $\Sigma^L \cong \Sigma$, then Σ^L is a halt state of the* LBCM *if and only if Σ is a halt state of the* SBCM.

Proof: Since $\Sigma^L \cong \Sigma$, we have that $(s^L, s \vdash t^L \simeq t)$, $(s^L, s \vdash k^L \simeq k)$, and $n^L = n$. The result then follows immediately by definition of term correspondence and halt states.

□

Lemma 18 *Let Σ^L and Σ be* LBCM *and* SBCM *initial states respectively. If $\Sigma^L \cong \Sigma$, then*

$$\mathcal{R}^{*L}(\Sigma^L) \cong \mathcal{R}^*(\Sigma)$$

Proof: This follows immediately from Lemmas 15 and 17.

□

3.2.4 Correspondence of Final Answers

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

Lemma 19 *Let Σ_f^L and Σ_f be* LBCM *and* SBCM *halt states respectively. If $\Sigma_f^L \cong \Sigma_f$, then $A_{lbcm}(\Sigma_f^L) = A_{sbcm}(\Sigma_f)$.*

Proof: Let v^L and v be the value registers of Σ_f^L and Σ_f respectively. By definition of state correspondence, v^L and v are related (i.e. $(s^L, s \vdash v^L \simeq v)$). By definition of augmented term correspondence, if v^L is a number m, then $v = \langle \text{FIXNUM } m \rangle$ and so $A_{lbcm}(\Sigma_f^L) = A_{sbcm}(\Sigma_f) = m$. If v^L is not a number, then v is not of the form $\langle \text{FIXNUM } m \rangle$ and so $A_{lbcm}(\Sigma_f^L)$ and $A_{sbcm}(\Sigma_f)$ are both undefined.

□

3.2.5 Correspondence of Semantics

We can now prove the main theorem of this chapter.

Theorem 20 *If an* LBC *program p^L corresponds to an* SBC *program p, then their respective operational semantics yield equal answers. That is,*

$$p^L \simeq^p p \Rightarrow \mathcal{O}^{lbc}[\![p^L]\!] = \mathcal{O}^{sbc}[\![p]\!]$$

Proof: Follows immediately from Lemmas 12, 18, and 19.

□

4 The Image Builder Algorithm

In this chapter, we present the Scheme code that implements the image builder. The image builder translates LBC programs into SBC programs such that their respective operational semantics yield equal answers. The input of the image builder is an LBC program in the following format.

$$
\begin{array}{l}
(m^L \\
\quad (\texttt{constants}\ c^{L*}) \\
\quad (\texttt{globals}\ m^{L*}) \\
\quad \{(\texttt{template}\ (w^{L*}) \\
\qquad\qquad (\{(\texttt{constant}\ m^L) \mid (\texttt{global}\ m^L) \mid (\texttt{template}\ m^L)\}^*) \\
\qquad\qquad)\}^+ \\
\quad)
\end{array}
$$

where m^L ranges over numbers, c^L ranges over LBC constants, and w^L ranges over LBC byte codes (either numbers, byte code operations, or identifiers). The notation $\{\ldots\}^*$ denotes a sequence of zero or more terms drawn from within the braces, $\{\ldots\}^+$ denotes a sequence of one or more terms drawn from within the braces, and $\ldots \mid \ldots$ denotes a choice of terms.

The output of the image-builder is an SBC program in the format

$$\langle m_1\ m_2\ m_3\rangle \frown s \frown \langle m_4\ vd_1\ vd_2\rangle$$

where $m_1, m_2, m_3 = 0$ (these may eventually be used to convey information), s is an SBC store, m_4 is the length of s, vd_1 is a pointer to a template object in the store s, and vd_2 is a pointer to a vector object in the store s. vd_1 represents the root template of the program and vd_2 represents the symbol table of the program. The symbol table is a table of all symbol constants in the program. We do not discuss the symbol table any further in this report; it is only used by certain Scheme standard functions which have not been given a formal semantics and hence whose behavior is not being verified.

Primitive functions

The image builder *algorithm* is presented in terms of a collection of primitive functions. These functions have SBC syntactic classes as their ranges. The image builder constructs SBC terms by applying these functions to values in the appropriate domains; thus the image builder algorithm does not use the actual representation of SBC terms but manipulates them only through these primitive functions. The primitive functions are specified in Figure 3.

`bins-per-cell`	=	bpw
`integer->addr` m	=	m
`addr->pointer` m	=	$\langle$PTR $m\rangle$
`int->fixnum` m	=	$\langle$FIXNUM $m\rangle$
`false`	=	$\langle$IMM FALSE$\rangle$
`true`	=	$\langle$IMM TRUE$\rangle$
`char->immediate` ch	=	$\langle$IMM $\langle$CHAR $m\rangle\rangle$
		where $m =$ `char->integer` ch
`null`	=	$\langle$IMM NULL$\rangle$
`unassigned-marker`	=	$\langle$IMM UNDEFINED$\rangle$
`hdr/pair`	=	PAIR
`hdr/symbol`	=	SYMBOL
`hdr/string`	=	STRING
`hdr/vector`	=	VECTOR
`hdr/location`	=	LOCATION
`hdr/template`	=	TEMPLATE
`hdr/codevector`	=	CODEVECTOR
`make-header` $h\ p\ m$	=	$\langle$HEADER $h\ p\ m\rangle$
`header-size-in-bins` $\langle$HEADER $h\ p\ m\rangle$	=	m
`header-size-in-cells` $\langle$HEADER $h\ p\ m\rangle$	=	$\mho m$
`integer->binval` m	=	m
`opname->binval` r	=	r

Figure 3: Primitive functions.

The *implementation* of the image builder does not have SBC programs as its range; rather, it produces a binary image, i.e., a sequence of bits. This aspect of the algorithm is verified in the VLISP report on the virtual machine [4]. We define a function from SBC descriptors and cells to the natural numbers. The primitive functions are implemented as the composition of this function with the specifications of Figure 3.

Miscellaneous functions

```
(define (allign bins-lst)
  (let* ((bins-in-last-cell
          (remainder (length bins-lst) bins-per-cell))
         (extra-bins-needed
          (remainder (- bins-per-cell bins-in-last-cell)
                     bins-per-cell))
         )
    (do ((i 0 (+ i 1))
         (extra-bins-lst '() (cons 0 extra-bins-lst))
         )
        ((= i extra-bins-needed) (append bins-lst extra-bins-lst))
      )))
(define (threaded-foreach f lst acc)
  (if (null? lst) acc
      (threaded-foreach f (cdr lst) (f (car lst) acc))))
```

Translating Constants

```
(define (translate-constants const-list image)
  (threaded-foreach translate-constant const-list image))

(define (translate-constant c image)
  (image-constant-add!
   (cond ((number? c) (PAIR (int->fixnum c) image))
         ((boolean? c) (PAIR (if c true false) image))
         ((char? c) (PAIR (char->immediate c) image))
         ((null? c) (PAIR null image))
         ((symbol? c) (build-symbol c image))
         ((string? c) (build-immutable-string c image))
         ((and (pair? c) (eq? (car c) 'pair))
          (build-immutable-pair c image))
         ((and (pair? c) (eq? (car c) 'vector))
          (build-immutable-vector c image))
         (#t (compiler-error 'image-builder 'translate-constant
```

```
                            "Invalid constant" c))
        )))

(define (build-symbol sym image)
  (let* ((symstr (symbol->string sym))
         (p1 (build-immutable-string symstr image))
         (str-ptr (PAIR-fst p1))
         (image1 (PAIR-snd p1))
         (p2 (image-store-add-stob image1
              (list (make-header hdr/symbol #f 1) str-ptr)))
         (symbol-ptr (PAIR-fst p2))
         (image2 (PAIR-snd p2))
         )
    (PAIR symbol-ptr
          (image-symbol-table-add image2 symstr symbol-ptr))
    ))

(define (build-immutable-string str image)
  (let ((charint-lst
         (append (map (lambda (x)
                        (integer->binval (char->integer x)))
                      (string->list str))
                 (list 0)))
        )
    (image-store-add-stob image
     (cons (make-header hdr/string #f (length charint-lst))
           (allign charint-lst))
     )))

(define (build-immutable-pair c image)
  (image-store-add-stob image
   (list (make-header hdr/pair #f 2)
         (image-constant-lookup image (cadr c))
         (image-constant-lookup image (caddr c))
         )
   ))

(define (build-immutable-vector c image)
  (let ((vlist (map (lambda (e)
                      (image-constant-lookup image e))
                    (cdr c))))
    (image-store-add-stob image
     (cons (make-header hdr/vector #f (length vlist)) vlist)
     )))
```

Translating Globals

```
(define (translate-globals globals-list image)
  (threaded-foreach translate-global globals-list image))

(define (translate-global lbc-global-name-index image)
  (let ((sbc-global-name
         (image-constant-lookup image lbc-global-name-index)))
    (image-global-add!
     (image-store-add-stob image
      (list (make-header hdr/location #t 2)
            unassigned-marker
            sbc-global-name)
     ))))
```

Translating Templates

```
(define (translate-templates template-lst image)
  (threaded-foreach translate-template template-lst image))

(define (translate-template lbc-templ image)
  (let ((lbc-code-list (cadr lbc-templ))
        (lbc-templ-contents (cdr (caddr lbc-templ)))
        ;Drop dummy first element
        )
    (image-template-add!
     (build-template lbc-code-list lbc-templ-contents image)
     ;codevector will be added as real first element of template
     )))

(define (translate-template-contents lbc-templ-contents image)
  (map (lambda (x)
         (cond ((eq? (car x) 'constant)
                (image-constant-lookup image (cadr x)))
               ((eq? (car x) 'global-variable)
                (image-global-lookup image (cadr x)))
               ((eq? (car x) 'template)
                (image-template-lookup image (cadr x)))
               ))
       lbc-templ-contents))

(define (build-template lbc-code-list lbc-templ-contents image)
  (let ((sbc-templ-contents
         (translate-template-contents lbc-templ-contents image))
        (cv-pair (build-codevector lbc-code-list image))
```

```
       )
    (image-store-add-stob
     (PAIR-snd cv-pair)
     (cons (make-header hdr/template #f
                        (+ 1 (length sbc-templ-contents)))
           (cons (PAIR-fst cv-pair) sbc-templ-contents))
     )))

(define (translate-codelist lbc-code-list)
  (map (lambda (x)
         (integer->binval (if (symbol? x) (opname->opcode x) x)))
       lbc-code-list))

(define (build-codevector lbc-code-list image)
  (let ((sbc-code-list (translate-codelist lbc-code-list)))
    (image-store-add-stob image
     (cons (make-header hdr/codevector #f (length sbc-code-list))
           (align sbc-code-list))
     )))
```

Translating the symbol table

```
(define (build-symbol-table image)
  (do ((i 0 (+ i 1))
       (image image
              (let ((p (build-mutable-list
                        (image-symbol-table-list image i) image)))
                (image-symbol-table-set! (PAIR-snd p) i
                                         (PAIR-fst p))
                ))
       )
      ((= i 256) (build-mutable-vector
                  (vector->list (image-symbol-table image))
                  image))
    ))

(define (build-mutable-list lst-of-sbc-values image)
  (let loop ((lst-of-sbc-values lst-of-sbc-values)
             (sbc-list null)
             (image image))
    (if (null? lst-of-sbc-values) (cons sbc-list image)
        (let ((p (build-immutable-pair-for-st
                  (car lst-of-sbc-values) sbc-list image)))
          (loop (cdr lst-of-sbc-values) (PAIR-fst p) (PAIR-snd p))
          ))))
```

```
(define (build-immutable-pair-for-st sbc-v1 sbc-v2 image)
  (image-store-add-stob image
   (list (make-header hdr/pair #f 2) sbc-v1 sbc-v2)
   ))

(define (build-mutable-vector sbc-lst image)
  (image-store-add-stob image
   (cons (make-header hdr/vector #t (length sbc-lst)) sbc-lst)
   ))
```

Translating LBC programs to SBC programs

```
(define (build-image lc output-port)
  (let* ((lbc-root-template (car lc))
         (lbc-constants-list (cdr (cadr lc)))
         (lbc-globals-list (cdr (caddr lc)))
         (lbc-templates-list (cdddr lc))
         (image (null-image (length lbc-constants-list)
                            (length lbc-globals-list)
                            (length lbc-templates-list)
                           output-port))
         (image-with-consts
          (translate-constants lbc-constants-list image))
         (image-with-globals
          (translate-globals lbc-globals-list image-with-consts))
         (image-with-templates
          (translate-templates lbc-templates-list
                               image-with-globals))
         (pair-with-symbol-table
          (build-symbol-table image-with-templates))
         (symbol-table-ptr (PAIR-fst pair-with-symbol-table))
         (final-image (PAIR-snd pair-with-symbol-table))
         )
    (write-cell (image-store-size final-image) output-port)
    (write-cell
     (image-template-lookup final-image lbc-root-template)
     output-port)
    (write-cell symbol-table-ptr output-port)
    ))

(define (write-cell cell out-port)
  (write-bin-lst (cell->bin-list cell) out-port))
(define (cell->bin-list cell)
  (let ((pos (>= cell 0)))
```

```
    (do ((w cell (quotient w MAXBINVAL))
         (i 0 (+ i 1))
         (lst '()
              (let ((v (remainder w MAXBINVAL)))
                (cons (if pos v (if (= i 0)
                                    (+ v MAXBINVAL)
                                    (+ v MAXBINVAL -1)))
                      lst)))
         )
        ((= i bins-per-cell)
         (if (> w 0)
             (compiler-error
              "number is larger than machine word" w)
             lst))
      )))
(define (write-bin-lst lst out-port)
  (for-each (lambda (bin)
              (write-char (integer->char bin) out-port))
            lst))
(define (write-stob-list lst out-port)
  (for-each (lambda (stob) (write-stob stob out-port)) lst))
(define (write-stob stob out-port)
  (if (desc-header-tag? (header-tag (car stob)))
      (for-each (lambda (cell) (write-cell cell out-port)) stob)
      (begin
        (write-cell (car stob) out-port)
        (write-bin-lst (cdr stob) out-port))
      ))
```

Image data type

An *image* is a tuple with four components:

Constant table: a list of constants, with each constant represented as a descriptor which may be a pointer into the store (see below);

Global table: a list of globals, with each global represented as a pointer into the store;

Template table: a list of templates, with each template represented as a pointer into the store;

Store: a list of bytes that comprise the actual code image.

In addition, the implementation builds a *symbol table*, i.e., a table of all symbols represented in the constant table.

Each component is represented as a vector together with a counter that contains the length of the list being represented. Thus, an image is represented as a vector with nine components:

1. constant-table: a vector of constants, with each constant represented as a descriptor which may be a pointer into the store (see below);
2. constant-table-index: number of constants in constant table;
3. global-table: a vector of globals, with each global represented as a pointer into the store;
4. global-table-index: number of globals in global table;
5. template-table: a vector of templates, with each template represented as a pointer into the store;
6. template-table-index: number of templates in template table;
7. store: a list of bytes (integers), that comprise the actual code image;
8. store-index: length of store;
9. symbol-table: the symbol table represented as a hash table.

```
(define (PAIR a b)                (cons a b))
(define (PAIR-fst m)              (car m))
(define (PAIR-snd m)              (cdr m))

(define (null-image cn gn tn output-port)
  (write-cell 0 output-port)
  (write-cell 0 output-port)
  (write-cell 0 output-port)
  (list->vector
   (list (make-vector cn 0) 0 (make-vector gn 0) 0
         (make-vector tn 0) 0 output-port 0 (make-vector 256 '())
         )))
(define image/constant-table 0)
(define image/constant-table-index 1)
(define image/global-table 2)
(define image/global-table-index 3)
(define image/template-table 4)
```

```
(define image/template-table-index 5)
(define image/store 6)
(define image/store-index 7)
(define image/symbol-table 8)

(define (image-lookup image vector-kind n)
    (vector-ref (vector-ref image vector-kind) (- n 1)))
(define (image-add! m vector-kind index-kind)
  (let ((v (PAIR-fst m))
        (image (PAIR-snd m)))
    (vector-set! (vector-ref image vector-kind)
                 (vector-ref image index-kind)
                 v)
    (vector-set! image index-kind
                 (+ 1 (vector-ref image index-kind)))
    image
    ))

(define (image-constant-lookup image n)
  (image-lookup image image/constant-table n))
(define (image-constant-add! p)
  (image-add! p image/constant-table image/constant-table-index))

(define (image-global-lookup image n)
  (image-lookup image image/global-table n))
(define (image-global-add! p)
  (image-add! p image/global-table image/global-table-index))

(define (image-template-lookup image n)
  (image-lookup image image/template-table n))
(define (image-template-add! p)
  (image-add! p image/template-table image/template-table-index))
```

The store is represented by an output port; stobs are added to the store by writing them out.

```
(define (image-store-size image)
  (vector-ref image image/store-index))
(define (image-store-append-stob! image stob)
  (write-stob stob (vector-ref image image/store))
  )
(define (image-store-add-stob image stob)
  (let* ((out-port (vector-ref image image/store))
         (old-hp (vector-ref image image/store-index))
```

```
         (len (+ 1 (header-size-in-cells (car stob))))
         )
    (image-store-append-stob! image stob)
    (vector-set! image image/store-index (+ old-hp len))
    (PAIR (addr->pointer (integer->addr (+ old-hp 1))) image)
    ))
(define (hash s)
  (let ((len (string-length s))
        (hash-prime 251))
    (let loop ((i 0)
               (acc 0))
      (if (= i len) acc
          (loop (+ i 1) (remainder
                         (+ (* acc 256)
                            (char->integer (string-ref s i)))
                         hash-prime))
          ))))
(define (image-symbol-table image)
  (vector-ref image image/symbol-table))
(define (image-symbol-table-list image n)
  (vector-ref (image-symbol-table image) n))
(define (image-symbol-table-set! image index val)
  (vector-set! (image-symbol-table image) index val)
  image)
(define (image-symbol-table-add image symstr symbol-ptr)
  (let ((index (hash symstr)))
    (image-symbol-table-set! image index
     (cons symbol-ptr (image-symbol-table-list image index)) )
    ))
```

5 Correctness of the Image Builder

We prove that the image builder translates LBC programs to related SBC programs. That is, if p^L is an LBC program then $p = \mathit{image\text{-}builder}(p^L)$ is an SBC program and $p^L \simeq^p p$. Theorem 20 permits us to conclude that the image builder preserves the operational semantics of programs, as desired.

Let $\langle m^L\ \eta_c^L\ \eta_g^L\ \eta_t^L \rangle$ be an LBC program where η_c^L, η_g^L, and η_t^L are LBC constant, global, and template tables respectively, and m^L is an index into η_t^L (m^L represents the "root" template).

The image builder proceeds by translating the LBC constant, global, and template tables. It maintains an intermediate data structure (called an image) of the form $\langle \eta_c\ \eta_g\ \eta_t\ s \rangle$. η_c, η_g, and η_t are SBC constant, global, and template tables respectively and represent the translations of the respective LBC tables. s is an SBC store and contains the SBC representations of the translated constant, global, and template terms.

5.1 Translating Constants

The function *translate-constants* translates LBC constants into SBC constants. After translation, entries in the LBC and SBC constant tables are related via a term correspondence relation. *translate-constants* is expressed in terms of an auxiliary function that translates a single LBC constant to an SBC constant.

Lemma 21 *Let $\simeq_0$ be a location correspondence and $\simeq$ be the term correspondence induced by $\simeq_0$. Let η_c^L be an* LBC *constant table. Let $\langle s\ \eta_c \rangle$ be an* SBC *constant table such that $\#\eta_c < \#\eta_c^L$ and $(\langle\rangle, s \vdash \eta_c^L(i) \simeq \eta_c(i))$ for all $0 \leq i < \#\eta_c$. Then, if*

$$\text{translate-constant}(\eta_c^L, \langle \eta_c\ \eta_g\ \eta_t\ s \rangle) = \langle \eta_c'\ \eta_g\ \eta_t\ s' \rangle$$

then $\eta_c' = \eta_c \frown \langle vd \rangle$ for some vd, and $(\langle\rangle, s' \vdash \eta_c^L(i) \simeq \eta_c'(i))$ for all $0 \leq i < \#\eta_c'$.

Proof: Let $j = \#\eta_c$.

Case $\eta_c^L(j) = m$

Then, $\eta_c'(j) = \langle \text{FIXNUM}\ m \rangle$ and $s' = s$. By definition of a term correspondence, $(\langle\rangle, s \vdash m \simeq \langle \text{FIXNUM}\ m \rangle)$.

Case $\eta_c^L(j) = \#f$

Then, $\eta_c'(j) = \langle \text{IMM FALSE} \rangle$ and $s' = s$. By definition of a term correspondence, $(\langle\rangle, s \vdash \#f \simeq \langle \text{IMM FALSE} \rangle)$.

Case $\eta_c^L(j) = \#t$

Then, $\eta_c'(j) = \langle \text{IMM TRUE} \rangle$ and $s' = s$. By definition of a term correspondence, $(\langle\rangle, s \vdash \#t \simeq \langle \text{IMM TRUE} \rangle)$.

Case $\eta_c^L(j) = ch$

Then, $\eta_c'(j) = \langle \text{IMM} \langle \text{CHAR } char{\rightarrow}int(ch) \rangle \rangle$ and $s' = s$. By definition of a term correspondence, $(\langle\rangle, s \vdash ch \simeq (\langle \text{IMM} \langle \text{CHAR } char{\rightarrow}int(ch) \rangle \rangle))$.

Case $\eta_c^L(j) = \texttt{nil}$

Then, $\eta_c'(j) = \langle \text{IMM NULL} \rangle$ and $s' = s$. By definition of a term correspondence, $(\langle\rangle, s \vdash \texttt{nil} \simeq \langle \text{IMM NULL} \rangle)$.

Case $\eta_c^L(j) = ch^{L*}$

Then, $\eta_c'(j) = \langle \text{PTR } \#s{+}1 \rangle$ and $s' = s \triangleright \langle \text{STRING } \#f \ char{\rightarrow}int^*(ch^{L*}) \rangle$. By definition of a term correspondence,

$$(\langle\rangle, s' \vdash ch^{L*} \simeq \langle \text{PTR } \#s + 1 \rangle)$$

Case $\eta_c^L(j) = \langle \texttt{immutable-pair}\ m_1^L\ m_2^L \rangle$

Then, $\eta_c'(j) = \langle \text{PTR } \#s + 1 \rangle$ and $s' = s \triangleright \langle \text{PAIR } \#f \ \langle \eta_c(m_1^L)\ \eta_c(m_2^L) \rangle \rangle$. Now, by definition of LBC, $m_1^L < j$ and $m_2^L < j$. Thus, by assumption, $(\langle\rangle, s \vdash \eta_c^L(m_i^L) \simeq \eta_c(m_i^L))$. Then, by definition of a term correspondence,

$$(\langle\rangle, s' \vdash \langle \texttt{immutable-pair}\ m_1^L\ m_2^L \rangle \simeq \langle \text{PTR } \#s + 1 \rangle)$$

Case $\eta_c^L(j) = \texttt{immutable-vector}{::}m^{L*}$

Then, $\eta_c'(j) = \langle \text{PTR } \#s + 1 \rangle$ and $s' = s \triangleright \langle \text{VECTOR } \#f \ \eta_c^*(m^{L*}) \rangle$. Now, by definition of LBC, $m^{L*}(i) < j$ for $0 \leq i < \#m^{L*}$. Thus, by assumption, for $0 \leq i < \#m^{L*}$, $(\langle\rangle, s' \vdash \eta_c^L(m^{L*}(i)) \simeq \eta_c^*(m^{L*})(i))$. Then, by definition of a term correspondence,

$$(\langle\rangle, s' \vdash (\texttt{immutable-vector}{::}m^{L*}) \simeq \langle \text{PTR } \#s + 1 \rangle)$$

Case $\eta_c^L(j) = r^L$

Let $b^* = char{\rightarrow}int^*(symbol{\rightarrow}string(r^L))$. Let $s_a = s \triangleright \langle \text{STRING } \#f \ b^* \rangle$ and $s' = s_a \triangleright \langle \text{SYMBOL } \#f \ \langle \langle \text{PTR } \#s{+}1 \rangle \rangle \rangle$. Then, $\eta_c'(j) = \langle \text{PTR } \#s_a +$

$1\rangle$. Now, $\mathcal{S}(\langle \textsc{ptr}\ \#s_a + 1\rangle, s') = \langle \textsc{symbol}\ \#f\ \langle\langle \textsc{ptr}\ \#s + 1\rangle\rangle\rangle$ and $\mathcal{S}(\langle \textsc{ptr}\ \#s + 1\rangle, s') = \langle \textsc{string}\ \#f\ b^*\rangle$. Then, by definition of a term correspondence, $(\langle\rangle, s' \vdash r^L \simeq \langle \textsc{ptr}\ \#s_a + 1\rangle)$.

□

5.2 Translating Globals

The function *translate-globals* translates LBC globals into SBC globals. After translation, entries in the LBC and SBC global tables are related via a term correspondence relation. *translate-globals* is expressed in terms of an auxiliary function that translates a single LBC global to an SBC global.

Lemma 22 *Let $\simeq_0$ be a location correspondence and $\simeq$ be the term correspondence induced by $\simeq_0$. Let s^L be an* LBC *store, and η_c^L and η_g^L be* LBC *constant and global tables respectively. Let s be an* SBC *store, and $\langle s\ \eta_c\rangle$ and $\langle s\ \eta_g\rangle$ be* SBC *constant and global tables respectively. Let*

$$\#s^L = \#\eta_g s^L = \textsc{unspecified}^*$$
$$\#\eta_c = \#\eta_c^L$$
$$(s^L, s \vdash \eta_c^L(i) \simeq \eta_c(i))\ \textit{for all}\ 0 \le i < \#\eta_c$$

$$\#\eta_g < \#\eta_g^L$$
$$(s^L, s \vdash \langle \texttt{global-variable}\ i\rangle \simeq \eta_g(i))\ \textit{for all}\ 0 \le i < \#\eta_g$$

Then, if

$$\text{translate-global}(\eta_c^L, \eta_g^L, \langle \eta_c\ \eta_g\ \eta_t\ s\rangle) = \langle \eta_c\ \eta_g'\ \eta_t\ s'\rangle$$

then there exists a location correspondence $\simeq_0'$ such that, if $\simeq'$ is the term correspondence induced by $\simeq_0'$ and $s'^L = s^L \frown \langle \textsc{unspecified}\rangle$, then

$$\eta_g' = \eta_g \frown \langle vd\rangle\ \textit{for some}\ vd,\ \textit{and}$$
$$(s'^L, s' \vdash \langle \texttt{global-variable}\ i\rangle \simeq' \eta_g'(i))\ \textit{for all}\ 0 \le i < \#\eta_g'$$

Proof: By definition,

$$\eta_g' = \eta_g \frown \langle \textsc{ptr}\ \#s + 1\rangle$$
$$s' = s \rhd \langle \textsc{location}\ \#t\ \langle\langle \textsc{undefined}\ \eta_c(\eta_g^L(\#\eta_g))\rangle\rangle\rangle$$

Let $\simeq_0'$ be a new location correspondence such that for all locations l^L, l, $(s'^L, s' \vdash l^L \simeq_0' l)$ holds if either $(s^L, s \vdash l^L \simeq_0 l)$ holds or if $l^L = \#s^L$ and $l = (\#s + 1)$. That is, the relation $\{(l^L, l) \mid (s'^L, s' \vdash l^L \simeq_0' l)\}$ extends the relation $\{(l^L, l) \mid (s^L, s \vdash l^L \simeq l)\}$ with the tuple $(\#s^L, (\#s + 1))$. Let $\simeq'$ be the term correspondence induced by $\simeq_0'$.

We now show that

$$(s'^L, s' \vdash \langle \texttt{global-variable}\ i \rangle \simeq' \eta_g'(i)) \text{ for all } 0 \leq i < \#\eta_g'$$

Since $\simeq_0'$ strictly extends $\simeq_0$, it follows that $\simeq'$ strictly extends $\simeq$. The result then follows by assumption for $0 \leq i < \#\eta_g$. Now let $i = \#\eta_g = (\#\eta_g' - 1)$. Then,

$$\begin{aligned}(s'^L, s' \vdash \langle \texttt{global-variable}\ i \rangle &\simeq' \eta_g'(i)) \\ &= (s'^L, s' \vdash \langle \texttt{global-variable}\ \#\eta_g \rangle \simeq' \langle \textsc{ptr}\ \#s + 1 \rangle)\end{aligned}$$

The proof obligations then are:

- $\mathcal{S}(\langle \textsc{ptr}\ \#s + 1 \rangle, s') = \langle \textsc{location}\ \#t\ \langle v_1\ v_2 \rangle \rangle$
  This holds by Lemma 4, with $v_1 = \textsc{undefined}$ and $v_2 = \eta_c(\eta_g^L(\#\eta_g))$.

- $(s'^L, s' \vdash \#\eta_g \simeq_0' \#s + 1)$
  This holds by definition of $\simeq_0'$, since $\#s^L = \#\eta_g$ by assumption.

- $(s'^L, s' \vdash \eta_c^L(\eta_g^L(\#\eta_g)) \simeq' \eta_c(\eta_g^L(\#\eta_g)))$
  Now, since $\#\eta_g < \#\eta_g^L$ by assumption, we have by definition of an LBC program that $\eta_g^L(\#\eta_g) < \#\eta_c^L$. Since $\#\eta_c^L = \#\eta_c$, we also have that $\eta_g^L(\#\eta_g) < \#\eta_c$. The result then holds by the assumption that

  $$(s^L, s \vdash \eta_c^L(i) \simeq \eta_c(i)) \text{ for all } 0 \leq i < \#\eta_c$$

□

5.3 Translating Templates

The function *translate-templates* translates LBC templates into SBC templates. After translation, entries in the LBC and SBC template tables are related via a term correspondence relation. *translate-templates* is expressed in terms of an auxiliary function *translate-template* that translates a single LBC template to an SBC template.

Lemma 23 *Let $\simeq_0$ be a location correspondence and $\simeq$ be the term correspondence induced by $\simeq_0$. Let s^L be an* LBC *store, and η_c^L, η_g^L, and η_t^L be* LBC *constant, global, and template tables respectively. Let s be an* SBC *store, and $\langle s\ \eta_c\rangle$, $\langle s\ \eta_g\rangle$, and $\langle s\ \eta_t\rangle$ be* SBC *constant, global, and template tables respectively. Let*

$$\begin{array}{l}
\#s^L = \#\eta_g^L s^L = \textsc{unspecified}^* \\
\#\eta_c = \#\eta_c^L \\
(s^L, s \vdash \eta_c^L(i) \simeq \eta_c(i)) \textit{ for all } 0 \le i < \#\eta_c \\
\\
\#\eta_g = \#\eta_g^L \\
(s^L, s \vdash \langle \texttt{global-variable}\ i\rangle \simeq \eta_g(i)) \textit{ for all } 0 \le i < \#\eta_g \\
\\
\#\eta_t < \#\eta_t^L \\
(s^L, s \vdash \eta_t^L(i) \simeq \eta_t(i)) \textit{ for all } 0 \le i < \#\eta_t
\end{array}$$

Then, if

$$\text{translate-template}(\eta_c^L, \eta_g^L, \eta_t^L, \langle \eta_c\ \eta_g\ \eta_t\ s\rangle) = \langle \eta_c\ \eta_g\ \eta_t'\ s'\rangle$$

then

$$\begin{array}{l}
\eta_t' = \eta_t \frown \langle vd\rangle \textit{ for some } vd, \textit{ and} \\
(s^L, s' \vdash \eta_t^L(i) \simeq \eta_t'(i)) \textit{ for all } 0 \le i < \#\eta_t'
\end{array}$$

Proof: Let $\eta_t^L(\#\eta_t) = \langle \texttt{template}\ w^L\ e^L\rangle$. Then, by definition,

$$\begin{array}{l}
s'' = s \rhd \langle \textsc{codevector}\ \#f\ w^L\rangle \\
c = \langle \textsc{ptr}\ \#s + 1\rangle \\
e'^L = e^L \dagger 1 \\
o^*(i) = \textit{translate-template-content}(e'^L(i)) \text{ for all } 0 \le i < \#e'^L \\
\#o^* = \#e'^L \\
s' = s'' \rhd \langle \textsc{template}\ \#f\ c{::}o^*\rangle \\
\eta_t' = \eta_t \frown \langle\langle \textsc{ptr}\ \#s'' + 1\rangle\rangle
\end{array}$$

Now, for $0 \le i < \#\eta_t$, the result follows immediately by assumption. Let $i = \#\eta_t = (\#\eta_t' - 1)$. Then,

$$\begin{array}{l}
(s^L, s' \vdash \eta_t^L(i) \simeq' \eta_t'(i)) \\
\quad = (s^L, s' \vdash \eta_t^L(\#\eta_t) \simeq' \eta_t'(\#\eta_t' - 1)) \\
\quad = (s^L, s' \vdash \langle \texttt{template}\ w^L\ e^L\rangle \simeq' \langle \textsc{ptr}\ \#s'' + 1\rangle)
\end{array}$$

The proof obligations then are:

1. $\mathcal{S}(\langle \textsc{ptr}\ \#s'' + 1\rangle, s') = \langle \textsc{template}\ \#f\ c{::}o^*\rangle$
   This holds by Lemma 4, with c and o^* given as above.

2. $(s^L, s \vdash w^L \simeq c)$
   This holds by definition since $\mathcal{S}(c, s) = \langle \textsc{codevector}\ \#f\ w^L\rangle$ by Lemma 4.

3. $\#e^L = \#(c{::}o^*)$
   This holds since $\#o^* = \#e'^L$ and $e'^L = e^L \dagger 1$.

4. $(\forall 0 \leq i < \#o^*)\ (s^L, s \vdash e'^L(i) \simeq o^*(i))$
   o^* is defined in terms of the function *translate-template-content*. We proceed by cases:

   Case $e'^L(i) = \langle \texttt{constant}\ j\rangle$**:**
   By definition of an LBC program, $j < \#eta_c^L = \#eta_c$. Then, $o^*(i) = \textit{translate-template-content}(\langle \texttt{constant}\ j\rangle) = \eta_c(j)$. Now $(s^L, s \vdash \langle \texttt{constant}\ j\rangle \simeq \eta_c(j))$ holds since $(s^L, s \vdash \eta_c^L(j) \simeq \eta_c(j))$ holds by assumption.

   Case $e'^L(i) = \langle \texttt{global-variable}\ j\rangle$**:**
   By definition of an LBC program, $j < \#eta_g^L = \#eta_g$. Then, $o^*(i) = \textit{translate-template-content}(\langle \texttt{global-variable}\ j\rangle) = \eta_g(j)$. But $(s^L, s \vdash \langle \texttt{global-variable}\ j\rangle \simeq \eta_g(j))$ holds by assumption.

   Case $e'^L(i) = \langle \texttt{template}\ j\rangle$**:**
   By definition of an LBC program, $j < \#eta_t < \#eta_t^L$. Then, $o^*(i) = \textit{translate-template-content}(\langle \texttt{template}\ j\rangle) = \eta_t(j)$. Now $(s^L, s \vdash \langle \texttt{template}\ j\rangle \simeq \eta_t(j))$ holds since $(s^L, s \vdash \eta_t^L(j) \simeq \eta_t(j))$ holds by assumption.

□

5.4 Translating Programs

The image builder is defined to be the composition of the above functions.

Definition 24 Let $\mathcal{P}^L = \langle m^L\ \eta_c^L\ \eta_g^L\ \eta_t^L\rangle$ be an LBC program. Let

$$\begin{array}{lcl} a_0 & = & \langle\langle\rangle\ \langle\rangle\ \langle\rangle\ \langle\rangle\rangle \\ a_1 & = & \textit{translate-constants}\,(\eta_c^L, a_0) \\ a_2 & = & \textit{translate-globals}\,(\eta_c^L, \eta_g^L, a_1) \\ a_3 & = & \textit{translate-templates}\,(\eta_c^L, \eta_g^L, \eta_t^L, a_2) \end{array}$$

Then, if $a_3 = \langle \eta_c\ \eta_g\ \eta_t\ s \rangle$, then

$$\mathit{build\text{-}image}(\mathcal{P}^L) = \langle s\ \eta_t(m^L) \rangle$$

Theorem 25 Let $\mathcal{P}^L$ be an LBC program. Then $\mathcal{P} = \mathit{build\text{-}image}(\mathcal{P}^L)$ is an SBC program and $\mathcal{P}^L \simeq^p \mathcal{P}$ holds.

Proof: Let $a_3 = \langle \eta_c\ \eta_g\ \eta_t\ s \rangle$ be defined as in Definition 24 above. Then, by Lemmas 21, 22, and 23, there exists a location correspondence $\simeq_0$ and a term correspondence $\simeq$ induced by $\simeq_0$ such that

$$\begin{array}{l} \#\eta_c = \#\eta_c^L, \#\eta_g = \#\eta_g^L, \text{ and } \#\eta_t = \#\eta_t^L, \\ (s^L, s \vdash \eta_c^L(i) \simeq \eta_c(i)) \text{ for all } 0 \leq i < \#\eta_c, \\ (s^L, s \vdash \eta_g^L(i) \simeq \eta_g(i)) \text{ for all } 0 \leq i < \#\eta_g, \text{ and} \\ (s^L, s \vdash \eta_t^L(i) \simeq \eta_t(i)) \text{ for all } 0 \leq i < \#\eta_t \end{array}$$

In particular,

$$(s^L, s \vdash \eta_t^L(m^L) \simeq \eta_t(m^L))$$

and so $\mathcal{P}^L \simeq^p \mathcal{P}$ by definition.

□

References

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

[2] 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.

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

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

[5] 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.