The VLISP Byte-Code Compiler

J. D. Guttman J. D. Ramsdell L. G. Monk
W. M. Farmer V. Swarup

The MITRE Corporation[1]
M92B092
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 provides a detailed presentation of the byte-code compiler and its proof of correctness.

Contents

List of Tables

1 Formal Syntax

In this section we present the abstract formal syntax for Scheme and for the byte code language (BBC) to which we will compile it. The semantics of these languages will follow in the next section.

The style of our grammars is somewhat different from that found in the Scheme Standard [2]. Some of these differences are purely incidental. The others are intended to make explicit our conception of the formal content conveyed by the Backus-Naur form clauses. Although these clauses are sometimes used to build up homogeneous strings of symbols from a given alphabet, we shall instead use them to build up more structured, tree-like objects. The latter seems more in tune with our interest in an *abstract* syntax or "derivation trees" [4].

With this convention, a BNF form such as $p ::= \langle \texttt{lambda}\ i^*\ e \rangle$ allows a p to be a sequence s of length three, whose second element may itself be a sequence of any length. It does *not* mean a sequence of any length greater than or equal to two, where all but the first and last elements are is. We would write the latter as $p ::= \langle \texttt{lambda} \rangle^\frown i^{*\frown} \langle e \rangle$.

For the sake of definiteness, we mention that we identify natural numbers with finite von Neumann ordinals, so that each number is the set of all smaller ones. We take a finite sequence to be a function with a natural number as its domain. Thus, an object $s \in \mathrm{X}^*$ is a function $s : n \to \mathrm{X}$ for some $n \in \omega$, and an object $s \in \mathrm{X}^+$ is a function $s : n \to \mathrm{X}$ for some non-zero $n \in \omega$. Moreover, $s(m)$ is the mth element of the (zero-based) sequence s, if the length of s is greater than m, and is undefined otherwise.

Thus, we regard every non-terminal value in the language defined by a BNF as being a nested finite sequence. Hence, we may define the *rank* of any value in a language:

Definition 1 (Rank)
The rank of a token (terminal) in a grammar is 0. The rank of a non-terminal finite sequence s is $1 + \sup\{rank(e) : e \in \mathrm{ran}(s)\}$.

We will also use the notation contained in Table 1. In the last three items, which concern type coercions, x and y should be thought of as expressions with an explicit syntactic type. Many concepts and most of the rest of the notation used here are described in [5].

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

Table 1: Some Notation

1.1 Scheme Syntax

In this section, we present a grammar for Scheme comparable to the one given in the Scheme Standard [2]. The syntactic objects are built from the following tokens:

- Scheme identifiers i;
- Scheme constants c;
- The tokens: `quote`, `begin`, `lambda`, `dotted_lambda`, `set!`, and `if`.

The BNF syntax is presented in Table 2.

Sequenced forms $\langle$`begin`$\rangle^\frown e^+$ are not normally part of the underlying abstract syntax of Scheme. We have decided to include them, and to restrict primitive `lambda`-expressions to those containing a single body-form. We find this trade-off slightly more natural. For instance, Scheme source code of the form:

```
(if test
    (begin form_1 ... form_n)
    else_clause)
```

must be translated into the form:

e, E	::=	$i \mid c_{sq} \mid \langle \texttt{quote}\ c \rangle \mid e^+ \mid \langle \texttt{begin} \rangle^\frown e^+$
	$\mid$	$\langle \texttt{lambda}\ i^*\ e \rangle \mid \langle \texttt{dotted\_lambda}\ i^+\ e \rangle$
	$\mid$	$\langle \texttt{if}\ e_1\ e_2\ e_3 \rangle \mid \langle \texttt{if}\ e_1\ e_2 \rangle \mid \langle \texttt{set!}\ i\ e \rangle$
c, K	::=	$c_{pr} \mid$ strings $\mid$ lists, dotted lists, and vectors of c
c_{pr}	::=	numbers, booleans, characters, symbols and nil
c_{sq}	::=	numbers, booleans, characters and strings
i, I	::=	identifiers (variables)
Γ	::=	e (commands)

Table 2: Scheme Abstract Syntax

```
(if test
    ((lambda () form_1 ... form_n))
    else_clause)
```

to be represented in the official abstract syntax. This simply makes the compilation task a little more complicated, as we would need to eliminate the call to the lambda-form.

We have also syntactically assimilated Scheme procedures of the form:

```
(lambda var body)
```

to those of the form:

```
(lambda ( . var) body)
```

This is their treatment in the denotational semantics anyway.

Within the category of constants c, we distinguish *primitive* constants from compound constants. Primitive constants are those that are not represented in the semantics by allocating locations in the store, while compound constants do require storage.

1.2 The Basic Byte Code Syntax

This section presents the syntax of the VLISP Basic Byte Code. Expressions of the Basic Byte Code language (BBC) are nested lists constructed according to the BNF grammar given in Table 3 from the following tokens:

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

Table 3: Grammar for the Basic Byte Code

- natural numbers, Scheme identifiers, Scheme constants;
- "key-words": `lap`, `call`, `return`, `make-cont`, `literal`, `closure`, `global`, `local`, `set-global!`, `set-local!`, `push`, `make-env`, `make-rest-list`, `unspecified`, `unless-false`, `checkargs=`, and `checkargs>=`.

We will use n-like variables for natural numbers, i-like variables for identifiers, and c-like variables for constants. Similarly for the classes defined by the grammar, with

z for (BBC) *neutral instructions*,

m for (BBC) (*machine*) *instructions*,

b for (BBC) *closed instruction lists* (conjecturally < Eng. *block*),

y for (BBC) *open instruction lists*,

w for (BBC) *(general) instruction lists*, and

t for (BBC) *templates*.

We will prove later that the result of the byte-code compiler is always a closed instruction list b. In the flattener algorithm, we treat open and closed instruction lists differently. Instructions of the form $\langle i\rangle$ are used

only for "primop" identifiers (like `%%cons`) associated with a small set of denotationally specified Scheme primitives.

1.3 The Tabular Byte Code Syntax

The Tabular Byte Code (or TBC) provides crude tables (just sequences of entries) that allow the code in templates to refer to constants, global variables (identifiers), and other templates indirectly by indexing. Otherwise it is almost exactly the same as the BBC.

The tokens of the TBC are the same as those of the BBC, with the addition of `constant`, and `global-variable`, and with `lap` being replaced by `template`. We adapt and extend the variable conventions of BBC. Thus z stands here for a (TBC) neutral instruction (or the class of neutral instructions), and so on for all of the TBC syntactic classes, including two new classes: d is used for (TBC) *table entries* (or *template literals*), and e for (TBC) *template tables*. The defining productions (given in Table 4) are very similar to the ones given for BBC, but note that:

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

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

Table 4: Grammar for the Tabular Byte Code

2 Denotational Semantics

2.1 Scheme Denotational Semantics

This section is a slightly modified version of an appendix to the proposed IEEE Scheme standard. As such, it is primarily the work of William Clinger and Jonathan Rees. It provides a formal denotational semantics for the primitive expressions of Scheme and selected built-in procedures. The version here differs from the standard version in three ways.

- We have reformulated some definitions using different notation or conventions. These changes are largely cosmetic.
- The domains have been made somewhat more concrete. In particular, `L` has been identified with `N`, and `S` has been identified with `E`*. Note that this entails that, at the current level of modeling, memory is conceived as unbounded.
- We have removed tests from the semantics to check whether a new storage location can be allocated in `S`. The official Scheme semantics uses conditionals that raise an "out of memory" error if there is no unallocated location. However, if memory is conceived as unbounded, this situation will not arise. Moreover, it does not seem that all situations in which a real Scheme interpreter can run out of memory are represented in the official semantics. Thus, we have chosen to represent all memory exhaustion errors uniformly at a much lower level in the formal specification.
- The constraints on $\mathcal{K}$ given in the section on Semantics of Constants has been added. It was needed in our work on the faithfulness of an operational semantics for the BBC to its denotational semantics.

We wish to rely as much as possible on standard usage for the theory of denotational semantics, but some clarifications and non-standard extensions are in order to keep our more complex semantical manipulations from becoming obscure or suspect.

Denotational domains are always taken to be pointed cpo's, with ordering and minimum element indicated as usual by $\sqsubseteq$ and $\bot$ respectively, possibly with disambiguating subscripts. Four domain constructors will be of particular importance:

1. Given two domains A and B, the "arrow domain" $A \to B$ has as elements all of the continuous functions from A to B. No new bottom is added: $\perp_{A\to B} = (\lambda x.\ \perp_B)$. Note that with this approach D is never equal to, say, $D \to D$, although the two may be isomorphic by a notationally suppressed isomorphism.

2. Given a sequence of denotational domains (of any ordinal length greater than one), their disjoint union (or disjoint sum[1]) is constructed in some standard way; a new bottom is added. For definiteness, we will take the non-bottom elements of the disjoint union of $\langle D_i \mid i < \beta\rangle$ to be the set of pairs $\langle i,\ x_i\rangle$ such that $x_i \in D_i$. In this case there are associated injections and projections, as well as typing predicates, as mentioned in the table of notation above. Note that if D_i is a direct summand of D, then $\perp_D : D_i$ is false.

3. Given a domain D, D^* is the domain of finite sequences from D. A new bottom is added, for which length is undefined, and which is different from the sequence of length 0. All other elements have finite lengths and can be applied to numbers less than their lengths. Sequences of defined, but different lengths are always incompatible in the ordering. Two sequences of the same length are ordered pointwise. Note that the domain version of D^* differs from the pure set of finite sequences from D, for which we use the same notation.

4. Cartesian products, usually of length two, but allowably of any ordinal length greater than one, have elements which are all sequences of the same length. Component extraction is by application to indices. No new bottom is added, as the sequence of bottoms of the components suffices.

We will have no need for a genuine direct sum construction, as opposed to a direct product; they are essentially different only for infinite sequences of arguments.

Call these four constructors *basic*, and a domain D produced by a basic constructor *decomposable*. Each basic constructor can be thought of as producing a domain from a sequence of *arguments*. We assume that the details are arranged so that these contructions are reversible, that is, that a decomposable domain has a unique basic constructor and argument sequence.

[1] This is not to be confused with a genuine direct sum nor with an ordinary union of domains which happen to be pairwise disjoint.

Thus we can speak of five disjoint classes of domains: arrow domains, disjoint sums, finite sequence domains, direct (or Cartesian) products, and indecomposable domains. Say that an indecomposable domain has an empty argument list.

An operation of functional lifting[2] will be useful to simplify certain fixed point constructions. Suppose that the arguments of D are the sequence of D_i for i less than β, and that for each i less than β, $f_i : D_i \to D_i$. (Here, and usually with denotational domains, $\to$ implies continuity. Also, association of application is usually to the left.) We define the *lifting* of these functionals to a function f on D. It should be easy to verify that $f : D \to D$. Assume x is an arbitrary non-bottom element of D, then:

1. If D is $D_0 \to D_1$ and $y \in D_0$, then $fxy = f_1(x(f_0(y)))$.
2. If D is a disjoint sum and $x = \langle i,\ x_i \rangle$, then $fx = \langle i,\ f_i(x_i) \rangle$. Also, f is strict, i.e., $f\bot_D = \bot_D$.
3. If $D = D_0^*$ and $x = \langle x_k : D_0 \mid k < n \rangle$, then $fx = \langle f_0(x_k) \mid k < n \rangle$; f is strict.
4. If D is a Cartesian product and $x = \langle x_i : D_i \mid i < \beta \rangle$, then $fx = \langle f_i(x_i) \mid i < \beta \rangle$.

We turn now to a few more detailed comments on the Scheme semantics.

The reason that expression continuations take sequences of values instead of single values is to simplify the formal treatment of procedure calls and to make it easy to add multiple return values.

The order of evaluation within a call is unspecified in the official Scheme semantics. In the compiler algorithm, however, we will select a single value for the permutations *permute* and *unpermute* applied to the arguments in a call before and after they are evaluated. The correctness proof will make use of this value.

We will also use Ide, Con, Exp, and Com to refer to the syntactic classes of identifiers i, constants c, expressions e, and commands (identical with expressions e), respectively.

The semantics of constants, given by a function $\mathcal{K}$, will not be completely defined. Rather, we will give constraints on this function. In essence, we consider its actual value to be a parameter to the semantics of Scheme.

[2]Not to be confused with what is sometimes called the lifting of a domain by adding a new bottom.

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

Table 5: Domains for the Semantics of Scheme

2.1.1 Domain equations

The domains used in the denotational semantics of Scheme are presented in Table 5.

Note that exactly one domain equation is actually not an equation. It implicitly introduces an isomorphism between E (which we assume to be indecomposable) and a disjoint sum, say $\psi_E : E \cong E_\Sigma$. This justifies an extension of the notations for projection, injection, and typing introduced above for disjoint sums. Thus, if D is an argument (component) of E_Σ, $e \in E$, and $d \in D$, then

$$e \,|\, D = \psi_E(e) \,|\, D;$$

$$(d \text{ in } E) = \psi_E^- 1(d \text{ in } E_\Sigma); \text{ and}$$

$e : D$ iff $\psi_E(e) : D$.

2.1.2 Semantic functions

$\mathcal{K}_0 : c_{pr} \to \mathtt{E}$
$\mathcal{K} : \mathrm{Con} \to \mathtt{E}$
$\mathcal{E} : \mathrm{Exp} \to \mathtt{U} \to \mathtt{K} \to \mathtt{C}$
$\mathcal{E}^* : \mathrm{Exp}^* \to \mathtt{U} \to \mathtt{K} \to \mathtt{C}$
$\mathcal{C} : \mathrm{Com}^* \to \mathtt{U} \to \mathtt{C} \to \mathtt{C}$

Semantics of Constants We first define $\mathcal{K}_0$. Suppose that c is a number, character, or symbol. Then $\mathcal{K}_0[\![c]\!] = c$. In addition:

$\mathcal{K}_0[\![\textsc{nil}]\!] = \mathit{null}$ in $\mathtt{E}$

$\mathcal{K}_0[\![\#\textsc{f}]\!] = \mathit{false}$ in $\mathtt{E}$

$\mathcal{K}_0[\![\#\textsc{t}]\!] = \mathit{true}$ in $\mathtt{E}$

We require that $\mathcal{K}_0 \sqsubseteq \mathcal{K}$ and several other conditions. First,

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

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

3. If c is a pair (and thus also if it is list or dotted list), and $\mathcal{K}[\![c]\!]$ is a non-bottom value ϵ, then $\epsilon : \mathtt{E}_\mathrm{p}$.

Second, if c is a string or vector, and $\mathcal{K}[\![c]\!]$ is a non-bottom value ϵ, then $(\epsilon \mid \mathtt{D})\ 1 = \mathit{immutable}$, where $\mathtt{D}$ is either $\mathtt{E}_\mathrm{s}$or $\mathtt{E}_\mathrm{v}$. Similarly, if c is a pair, and $\mathcal{K}[\![c]\!]$ is a non-bottom value ϵ, then $(\epsilon \mid \mathtt{E}_\mathrm{p})\ 2 = \mathit{immutable}$.

Finally, if c_0 is a vector or pair, and $\mathcal{K}[\![c_0]\!]$ is a non-bottom value ϵ, and c_1 is a subexpression of c_0, then $\mathcal{K}[\![c_1]\!]$ is also non-bottom.

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

Semantics of Expressions

$$\mathcal{E}[\![\mathrm{K}]\!] = \lambda\rho\kappa\,.\,\mathit{send}\,(\mathcal{K}[\![\mathrm{K}]\!])\,\kappa$$

$$\begin{aligned}\mathcal{E}[\![\mathrm{I}]\!] = \lambda\rho\kappa\,.\,&\mathit{hold}\,(\mathit{lookup}\,\rho\,\mathrm{I})\\ &(\mathit{single}(\lambda\epsilon\,.\,\epsilon = \mathit{empty}\text{ in }\mathtt{E}\rightarrow\\ &\qquad\mathit{wrong}\text{ ``undefined variable''},\\ &\quad\mathit{send}\,\epsilon\,\kappa))\end{aligned}$$

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

$$\begin{aligned}&\mathcal{E}[\![\langle\mathtt{lambda}\ \mathrm{I}^*\ \mathrm{E}_0\rangle]\!] =\\ &\quad\lambda\rho\kappa\,.\,\lambda\sigma\,.\\ &\qquad\mathit{send}\,(\langle\mathit{new}\,\sigma,\\ &\qquad\quad\lambda\epsilon^*\kappa'\,.\,\#\epsilon^* = \#\mathrm{I}^*\rightarrow\\ &\qquad\qquad\mathit{tievals}(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\rho'\kappa')\\ &\qquad\qquad\qquad(\mathit{extend}\,\rho\,\mathrm{I}^*\,\alpha^*))\\ &\qquad\qquad\qquad\epsilon^*,\\ &\qquad\qquad\mathit{wrong}\text{ ``wrong number of arguments''}\rangle\\ &\qquad\quad\text{in }\mathtt{E})\\ &\qquad\kappa\\ &\qquad(\mathit{update}\,(\mathit{new}\,\sigma)\,(\mathit{unspecified}\text{ in }\mathtt{E})\,\sigma)\end{aligned}$$

$$\begin{aligned}&\mathcal{E}[\![\langle\mathtt{dotted\_lambda}\ \mathrm{I}^*{}^\frown\langle\mathrm{I}\rangle\ \mathrm{E}_0\rangle]\!] =\\ &\quad\lambda\rho\kappa\,.\,\lambda\sigma\,.\\ &\qquad\mathit{send}\,(\langle\mathit{new}\,\sigma,\\ &\qquad\quad\lambda\epsilon^*\kappa'\,.\,\#\epsilon^* \geq \#\mathrm{I}^*\rightarrow\\ &\qquad\qquad\mathit{tievalsrest}\\ &\qquad\qquad\quad(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\rho'\kappa')\\ &\qquad\qquad\qquad\quad(\mathit{extend}\,\rho\,(\mathrm{I}^*{}^\frown\langle\mathrm{I}\rangle)\,\alpha^*))\\ &\qquad\qquad\quad\epsilon^*\\ &\qquad\qquad\quad(\#\mathrm{I}^*),\\ &\qquad\qquad\mathit{wrong}\text{ ``too few arguments''}\rangle\text{ in }\mathtt{E})\\ &\qquad\kappa\\ &\qquad(\mathit{update}\,(\mathit{new}\,\sigma)\,(\mathit{unspecified}\text{ in }\mathtt{E})\,\sigma)\end{aligned}$$

$$\mathcal{E}[\![\langle \texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2\rangle]\!] =$$
$$\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\,(\mathit{single}\,(\lambda\epsilon\,.\,\mathit{truish}\,\epsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa,$$
$$\qquad\qquad\qquad\qquad\qquad\qquad \mathcal{E}[\![\mathrm{E}_2]\!]\rho\kappa))$$

$$\mathcal{E}[\![\langle \texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1\rangle]\!] =$$
$$\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\,(\mathit{single}\,(\lambda\epsilon\,.\,\mathit{truish}\,\epsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa,$$
$$\qquad\qquad\qquad\qquad\qquad\qquad \mathit{send}\,(\mathit{unspecified}\ \text{in}\ \texttt{E})\,\kappa))$$

Here and elsewhere, any expressed value other than *empty* could have been used in place of *unspecified*.

$$\mathcal{E}[\![\langle \texttt{set!}\ \mathrm{I}\ \mathrm{E}\rangle]\!] =$$
$$\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}]\!]\,\rho\,(\mathit{single}(\lambda\epsilon\,.\,\mathit{assign}\,(\mathit{lookup}\,\rho\,\mathrm{I})$$
$$\qquad\qquad\qquad\qquad\qquad\qquad \epsilon$$
$$\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{send}\,(\mathit{unspecified}\ \text{in}\ \texttt{E})\,\kappa)))$$

$$\mathcal{E}[\![\texttt{begin} :: \Gamma^* {}^\frown \langle \mathrm{E}_0\rangle]\!] =$$
$$\quad \lambda\rho\kappa\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho(\mathcal{E}[\![\mathrm{E}_0]\!]\rho\kappa)$$

Semantics of Argument Lists and Command Lists

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

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

$$\mathcal{C}[\![\langle\rangle]\!] = \lambda\rho\theta\,.\,\theta$$

$$\mathcal{C}[\![\Gamma_0 :: \Gamma^*]\!] = \lambda\rho\theta\,.\,\mathcal{E}[\![\Gamma_0]\!]\,\rho\,(\lambda\epsilon^*\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho\theta)$$

2.1.3 Auxiliary functions

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

$$\mathit{extend} : \texttt{U} \to \mathrm{Ide}^* \to \texttt{L}^* \to \texttt{U}$$
$$\mathit{extend} =$$
$$\quad \lambda\rho\mathrm{I}^*\alpha^*\,.\,\#\mathrm{I}^* = 0 \to \rho,$$
$$\qquad\qquad \mathit{extend}\,(\rho[(\alpha^*\,0)/(\mathrm{I}^*\,0)])\,(\mathrm{I}^* \dagger 1)\,(\alpha^* \dagger 1)$$

$$\mathit{wrong} : \texttt{X} \to \texttt{C} \qquad \text{[implementation-dependent]}$$

$send : \mathtt{E} \to \mathtt{K} \to \mathtt{C}$
$send = \lambda\epsilon\kappa \,.\, \kappa\langle\epsilon\rangle$

$single : (\mathtt{E} \to \mathtt{C}) \to \mathtt{K}$
$single =$
$\quad \lambda\psi\epsilon^* \,.\, \#\epsilon^* = 1 \to \psi(\epsilon^*\, 0),$
$\qquad\qquad wrong$ "wrong number of return values"

The store is a sequence of expressed values indexed by the natural numbers less than the length of the store. The function *new* returns the smallest index corresponding to a location not in the store.

$new : \mathtt{S} \to \mathtt{L}$
$new = \lambda\sigma \,.\, \#\sigma$

$hold : \mathtt{L} \to \mathtt{K} \to \mathtt{C}$
$hold = \lambda\alpha\kappa\sigma \,.\, \alpha < \#\sigma \to send\,(\sigma\alpha)\kappa\sigma,\ empty$ in $\mathtt{E}$

$assign : \mathtt{L} \to \mathtt{E} \to \mathtt{C} \to \mathtt{C}$
$assign = \lambda\alpha\epsilon\theta\sigma \,.\, \theta(update\ \alpha\epsilon\sigma)$

$update : \mathtt{L} \to \mathtt{E} \to \mathtt{S} \to \mathtt{S}$

The value of $update\alpha\epsilon\sigma$ is the finite sequence σ' of length $\#\sigma' = \max(\alpha + 1, \#\sigma)$ such that, for any $\alpha' < \#\sigma'$,

$\alpha' = \alpha \to \epsilon,$
$\quad \alpha' < \#\sigma \to \sigma\ \alpha, empty$ in $\mathtt{E}$

$tievals : (\mathtt{L}^* \to \mathtt{C}) \to \mathtt{E}^* \to \mathtt{C}$
$tievals =$
$\quad \lambda\xi\epsilon^*\sigma \,.\, \#\epsilon^* = 0 \to \xi\langle\,\rangle\sigma,$
$\qquad tievals\,(\lambda\alpha^* \,.\, \xi(\langle new\ \sigma\,\rangle^\frown \alpha^*))$
$\qquad\qquad (\epsilon^* \dagger 1)$
$\qquad\qquad (update\ (new\ \sigma)(\epsilon^*\, 0)\sigma)$

$tievalsrest : (\mathtt{L}^* \to \mathtt{C}) \to \mathtt{E}^* \to \mathtt{N} \to \mathtt{C}$
$tievalsrest =$
$\quad \lambda\xi\epsilon^*\nu \,.\, list\,(\epsilon^* \dagger \nu)$
$\qquad\qquad (single(\lambda\epsilon \,.\, tievals\ \xi\ ((\epsilon^* \ddagger \nu)^\frown \langle\epsilon\rangle))))$

$truish : \mathtt{E} \to \mathtt{T}$
$truish = \lambda\epsilon\ .\ (\epsilon = false \text{ in } \mathtt{E}) \to false, true$

The auxiliary *perm*, used to define *permute* and *unpermute*, is not defined in the semantics. It is a function that, given a number ν, returns a permutation of the natural numbers less than ν. In the Vlisp implementation, we take $perm\nu$, for any $\nu > 0$, to be the permutation that given an argument $\nu_0 < \nu - 1$, returns $\nu_0 + 1$, and which returns 0 when applied to $\nu - 1$.

$permute : \mathrm{Exp}^* \to \mathrm{Exp}^*$
$permute = \lambda e^*\ .\ \lambda\nu\ .\ e^*((perm\ \#e^*)\ \nu)$

$unpermute : \mathtt{E}^* \to \mathtt{E}^*$
$unpermute = \lambda\epsilon^*\ .\ \lambda\nu\ .\ \epsilon^*((perm\ \#\epsilon^*)^{-1}\ \nu)$

$applicate : \mathtt{E} \to \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$
$applicate =$
$\quad \lambda\epsilon\epsilon^*\kappa\ .\ \epsilon : \mathtt{F} \to ((\epsilon \mid \mathtt{F})\ 1)\epsilon^*\kappa, wrong$ "bad procedure"

$twoarg : (\mathtt{E} \to \mathtt{E} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C})$
$twoarg =$
$\quad \lambda\zeta\epsilon^*\kappa\ .\ \#\epsilon^* = 2 \to \zeta\,(\epsilon^*\ 0)(\epsilon^*\ 1)\kappa,$
$\qquad\qquad wrong$ "wrong number of arguments"

$list : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$
$list =$
$\quad \lambda\epsilon^*\kappa\ .\ \#\epsilon^* = 0 \to send\,(null \text{ in } \mathtt{E})\ \kappa,$
$\qquad\qquad list\,(\epsilon^* \dagger 1)(single(\lambda\epsilon\ .\ cons\langle(\epsilon^*\ 0), \epsilon\rangle\kappa))$

$cons : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$
$cons =$
$\quad twoarg\,(\lambda\epsilon_1\epsilon_2\kappa\sigma\ .\ (\lambda\sigma'\ .\ send\,(\langle new\ \sigma, new\ \sigma', mutable\rangle \text{ in } \mathtt{E})$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \kappa$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (update(new\ \sigma')\epsilon_2\sigma')$
$\qquad\qquad\qquad\qquad (update(new\ \sigma)\epsilon_1\sigma)))$

2.2 Byte Code Denotational Semantics

2.2.1 Domain equations

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

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

2.2.2 Semantic functions

$\mathcal{B} : b \cup \{\langle\rangle\} \to \mathtt{U} \to \mathtt{P}$
$\mathcal{Z} : z \to \mathtt{U} \to \mathtt{P} \to \mathtt{P}$
$\mathcal{Y} : y \cup \{\langle\rangle\} \to \mathtt{U} \to \mathtt{P} \to \mathtt{P}$ (The variable y' will range over $y \cup \{\langle\rangle\}$)
$\mathcal{T} : t \to \mathtt{U} \to \mathtt{P}$

Semantic Clauses for Core Instructions

$\mathcal{B}[\![\langle\rangle]\!] = \lambda\rho\epsilon\epsilon^*\rho_R\psi\sigma \,.\, \epsilon : \mathtt{R} \to \epsilon \mid \mathtt{R} \text{ in } \mathtt{A}, \bot$

$\mathcal{B}[\![\langle\langle \mathtt{return} \rangle\rangle]\!] = \lambda\rho \,.\, \mathit{return}$

$\mathcal{B}[\![\langle\langle \mathtt{call}\ n \rangle\rangle]\!] = \lambda\rho \,.\, \mathit{call}\, n$

$\mathcal{B}[\![\langle\langle \mathtt{unless\text{-}false}\ b_1\ b_2 \rangle\rangle]\!] = \lambda\rho \,.\, \mathit{if\_truish}(\mathcal{B}[\![b_1]\!]\rho)(\mathcal{B}[\![b_2]\!]\rho)$

$\mathcal{B}[\![\langle \mathtt{make\text{-}cont}\ b_1\ n \rangle :: b_2]\!] = \lambda\rho \,.\, \mathit{make\_cont}\,(\mathcal{B}[\![b_1]\!]\rho)n(\mathcal{B}[\![b_2]\!]\rho)$

$\mathcal{B}[\![z :: b]\!] = \lambda\rho \,.\, \mathcal{Z}[\![z]\!]\rho\,(\mathcal{B}[\![b]\!]\rho)$

$\mathcal{Z}[\![\langle \mathtt{unless\text{-}false}\ y_1\ y_2 \rangle]\!] = \lambda\rho\pi \,.\, \mathit{if\_truish}(\mathcal{Y}[\![y_1]\!]\rho\pi)(\mathcal{Y}[\![y_2]\!]\rho\pi)$

$\mathcal{Z}[\![\langle \mathtt{literal}\ c \rangle]\!] = \lambda\rho \,.\, \mathit{literal}\,(\mathcal{K}[\![c]\!])$

$\mathcal{Z}[\![\langle \mathtt{closure}\ t \rangle]\!] = \lambda\rho \,.\, \mathit{closure}\,(\mathcal{T}[\![t]\!]\rho)$

$\mathcal{Z}[\![\langle \mathtt{global}\ i \rangle]\!] = \lambda\rho \,.\, \mathit{global}\,(\mathit{lookup}\,\rho\,i)$

$\mathcal{Z}[\![\langle \mathtt{local}\ n_0\ n_1 \rangle]\!] = \lambda\rho \,.\, \mathit{local}\, n_0\ n_1$

$\mathcal{Z}[\![\langle \mathtt{set\text{-}global!}\ i \rangle]\!] = \lambda\rho \,.\, \mathit{set\_global}\,(\mathit{lookup}\,\rho\,i)$

$\mathcal{Z}[\![\langle \texttt{set-local!}\ n_0\ n_1\rangle]\!] = \lambda\rho\,.\,\mathit{set\_local}\,n_0\,n_1$

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

$\mathcal{Z}[\![\langle \texttt{make-env}\ n\rangle]\!] = \lambda\rho\,.\,\mathit{make\_env}\,n$

$\mathcal{Z}[\![\langle \texttt{make-rest-list}\ n\rangle]\!] = \lambda\rho\,.\,\mathit{make\_rest\_list}\,n$

$\mathcal{Z}[\![\langle \texttt{unspecified}\rangle]\!] = \lambda\rho\,.\,\mathit{literal}\,\mathit{unspecified}$ in $\texttt{E}$

$\mathcal{Z}[\![\langle \texttt{checkargs=}\ n\rangle]\!] = \lambda\rho\,.\,\mathit{check\_args\_eq}\,n$

$\mathcal{Z}[\![\langle \texttt{checkargs>=}\ n\rangle]\!] = \lambda\rho\,.\,\mathit{check\_args\_ge}\,n$

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

$\mathcal{Y}[\![z :: y']\!] = \lambda\rho\pi\,.\,\mathcal{Z}[\![z]\!]\rho\,(\mathcal{Y}[\![y']\!]\rho\pi)$

$\mathcal{Y}[\![\langle\rangle]\!] = \lambda\rho\pi\,.\,\pi$

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

Semantic Clauses for Data Manipulation Primitives The clauses above do not specify the semantics for instructions z of the form $\langle i\rangle$. These instructions are considered to be data manipulation primitives. Since the standard data manipulation procedures of Scheme are not formally specified in the semantics, we have not thought it worthwhile to define them here.

However, in the formal semantics of the tabular byte code TBC below, we have specified several of these primitives. We have done so to demonstrate that it is straightforward to do so for a wide variety of primitives. Moreover, the proof that the operational semantics is faithful to the denotational semantics takes account of these primtives also.

2.2.3 Auxiliary functions

$\mathit{extend}_R : \texttt{U}_R \to \texttt{L}^* \to \texttt{U}_R$
$\mathit{extend}_R =$
$\quad \lambda\rho_R\alpha^*\,.\,\lambda\nu_1\nu_2\,.\,\nu_1 = 0 \to \mathit{rev}(\alpha^*)\,(\nu_2 - 1),\ \rho_R(\nu_1 - 1)\nu_2$

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

$$
\begin{array}{l}
\mathit{return} : \mathtt{P} \\
\mathit{return} = \lambda\epsilon\epsilon^*\rho_R\psi\,.\,\psi\epsilon \\
\\
\mathit{make\_cont} : \mathtt{P} \to \mathtt{N} \to \mathtt{P} \to \mathtt{P} \\
\mathit{make\_cont} = \\
\quad \lambda\pi'\nu\pi\,.\,\lambda\epsilon\epsilon^*\rho_R\psi\,.\,\#\epsilon^* = \nu \to \pi\epsilon\langle\rangle\rho_R(\lambda\epsilon\,.\,\pi'\epsilon\epsilon^*\rho_R\psi), \\
\qquad\qquad\qquad\qquad \mathit{wrong}\ \text{“bad stack”} \\
\\
\mathit{literal} : \mathtt{E} \to \mathtt{P} \to \mathtt{P} \\
\mathit{literal} = \lambda\epsilon'\pi\,.\,\lambda\epsilon\epsilon^*\rho_R\psi\,.\,\pi\epsilon'\epsilon^*\rho_R\psi \\
\\
\mathit{closure} : \mathtt{P} \to \mathtt{P} \to \mathtt{P} \\
\mathit{closure} = \\
\quad \lambda\pi'\pi\,.\,\lambda\epsilon\epsilon^*\rho_R\psi\sigma\,. \\
\qquad \pi(\mathit{fix}(\lambda\epsilon\,.\,\langle \mathit{new}\,\sigma,\ \lambda\epsilon^*\kappa\,.\,\pi'\epsilon\epsilon^*\rho_R(\lambda\epsilon\,.\,\kappa\langle\epsilon\rangle)\rangle\ \text{in}\ \mathtt{E})) \\
\qquad\ \ \epsilon^*\rho_R\psi(\mathit{update}(\mathit{new}\,\sigma)\ (\mathit{unspecified}\ \text{in}\ \mathtt{E})\,\sigma) \\
\\
\mathit{global} : \mathtt{L} \to \mathtt{P} \to \mathtt{P} \\
\mathit{global} = \\
\quad \lambda\alpha\pi\,.\,\lambda\epsilon\epsilon^*\rho_R\psi\,. \\
\qquad \mathit{hold}\,\alpha \\
\qquad\qquad (\mathit{single}\ (\lambda\epsilon\,.\,\epsilon \neq \mathit{empty}\ \text{in}\ \mathtt{E} \to \pi\epsilon\epsilon^*\rho_R\psi, \\
\qquad\qquad\qquad\qquad \mathit{wrong}\ \text{“undefined variable”})) \\
\\
\mathit{local} : \mathtt{N} \to \mathtt{N} \to \mathtt{P} \to \mathtt{P} \\
\mathit{local} = \\
\quad \lambda\nu_1\nu_2\pi\,.\,\lambda\epsilon\epsilon^*\rho_R\psi\,. \\
\qquad \mathit{hold}\,(\rho_R\nu_1\nu_2) \\
\qquad\qquad (\mathit{single}\ (\lambda\epsilon\,.\,\epsilon \neq \mathit{empty}\ \text{in}\ \mathtt{E} \to \pi\epsilon\epsilon^*\rho_R\psi, \\
\qquad\qquad\qquad\qquad \mathit{wrong}\ \text{“undefined variable”})) \\
\\
\mathit{set\_global} : \mathtt{L} \to \mathtt{P} \to \mathtt{P} \\
\mathit{set\_global} = \\
\quad \lambda\alpha\pi\,.\,\lambda\epsilon\epsilon^*\rho_R\psi\,. \\
\qquad \mathit{assign}\,\alpha\,\epsilon\,(\pi\,(\mathit{unspecified}\ \text{in}\ \mathtt{E})\,\epsilon^*\rho_R\psi) \\
\\
\mathit{set\_local} : \mathtt{N} \to \mathtt{N} \to \mathtt{P} \to \mathtt{P} \\
\mathit{set\_local} = \\
\quad \lambda\nu_1\nu_2\pi\,.\,\lambda\epsilon\epsilon^*\rho_R\psi\,. \\
\qquad \mathit{assign}\,(\rho_R\nu_1\nu_2)\,\epsilon\,(\pi\,(\mathit{unspecified}\ \text{in}\ \mathtt{E})\,\epsilon^*\rho_R\psi)
\end{array}
$$

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

$$
\begin{array}{l}
make\_env : \mathtt{N} \to \mathtt{P} \to \mathtt{P} \\
make\_env = \\
\quad \lambda\nu\pi \,.\, \lambda\epsilon\epsilon^*\rho_R\psi \,.\, \#\epsilon^* = \nu \to tievals\ (\lambda\alpha^* \,.\, \pi\epsilon\langle\rangle(extend_R\ \rho_R\alpha^*)\psi)\ \epsilon^*, \\
\qquad\qquad\qquad\qquad wrong\ \text{“bad stack”}
\end{array}
$$

$$
\begin{array}{l}
make\_rest\_list : \mathtt{N} \to \mathtt{P} \to \mathtt{P} \\
make\_rest\_list = \\
\quad \lambda\nu\pi \,.\, \lambda\epsilon\epsilon^*\rho_R\psi \,. \\
\qquad \#\epsilon^* \geq \nu \to \\
\qquad\quad list\,(\epsilon^* \dagger \nu) \\
\qquad\qquad\quad (single\ \lambda\epsilon \,.\, \pi\epsilon((\epsilon^* \ddagger \nu))\rho_R\psi), \\
\qquad wrong\ \text{“bad stack”}
\end{array}
$$

$$
\begin{array}{l}
if\_truish : \mathtt{P} \to \mathtt{P} \to \mathtt{P} \\
if\_truish = \\
\quad \lambda\pi_1\pi_2 \,.\, \lambda\epsilon\epsilon^*\rho_R\psi \,.\, truish\ \epsilon \to \pi_1\epsilon\epsilon^*\rho_R\psi, \pi_2\epsilon\epsilon^*\rho_R\psi
\end{array}
$$

$$
\begin{array}{l}
check\_args\_eq : \mathtt{N} \to \mathtt{P} \to \mathtt{P} \\
check\_args\_eq = \\
\quad \lambda\nu\pi \,.\, \lambda\epsilon\epsilon^*\rho_R\psi \,. \\
\qquad \#\epsilon^* = \nu \to \pi\epsilon\epsilon^*\rho_R\psi, wrong\ \text{“bad arg count”}
\end{array}
$$

$$
\begin{array}{l}
check\_args\_ge : \mathtt{N} \to \mathtt{P} \to \mathtt{P} \\
check\_args\_ge = \\
\quad \lambda\nu\pi \,.\, \lambda\epsilon\epsilon^*\rho_R\psi \,. \\
\qquad \#\epsilon^* \geq \nu \to \pi\epsilon\epsilon^*\rho_R\psi, wrong\ \text{“bad arg count”}
\end{array}
$$

2.3 Tabular Byte Code Denotational Semantics

In this section, we specify denotational semantics for the Tabular Byte Code TBC.

2.3.1 Semantic functions

Types of the Functions The semantics of the TBC make use of four main semantic functions:

$\mathcal{B}_\tau : b \cup \{\langle\rangle\} \to e \to \mathtt{U} \to \mathtt{P}$
$\mathcal{Z}_\tau : z \to e \to \mathtt{U} \to \mathtt{P} \to \mathtt{P}$
$\mathcal{Y}_\tau : y \cup \{\langle\rangle\} \to \mathtt{U} \to \mathtt{P} \to \mathtt{P}$ (The variable y' ranges over $y \cup \{\langle\rangle\}$)
$\mathcal{T}_\tau : t \to \mathtt{U} \to \mathtt{P}$

Semantic Clauses for Core Instructions In several of these clauses, we refer to a value in the template table. In the instructions:

`literal`, `global`, and `set-global!`,

we expect the relevant template table entry to be of one of the forms:

$\langle$`constant` $c\rangle$ or $\langle$`global-variable` $i\rangle$.

Naturally, it is the value c or i that interests us. Since the tag is the zeroth item in the sequence, we can extract the value by applying the sequence to the argument 1. Hence, these clauses use the expression $(e(n))(1)$ to extract the value from the template table.

$\mathcal{B}_\tau[\![\langle\rangle]\!] = \lambda e\rho\epsilon\epsilon^*\rho_R\psi\sigma\,.\,\epsilon : \mathtt{R} \to \epsilon \mid \mathtt{R} \text{ in } \mathtt{A}, \bot$

$\mathcal{B}_\tau[\![\langle\langle$`return`$\rangle\rangle]\!] == \lambda e\rho\,.\,\mathit{return}$

$\mathcal{B}_\tau[\![\langle\langle$`call` $n\rangle\rangle]\!] = \lambda e\rho\,.\,\mathit{call}\,n$

$\mathcal{B}_\tau[\![\langle\langle$`unless-false` $b_1\ b_2\rangle\rangle]\!] = \lambda e\rho\,.\,\mathit{if\_truish}(\mathcal{B}_\tau[\![b_1]\!]e\rho)(\mathcal{B}_\tau[\![b_2]\!]e\rho)$

$\mathcal{B}_\tau[\![\langle$`make-cont` $b_1\ n\rangle :: b_2]\!] = \lambda e\rho\,.\,\mathit{make\_cont}\,(\mathcal{B}_\tau[\![b_1]\!]e\rho)n(\mathcal{B}_\tau[\![b_2]\!]e\rho)$

$\mathcal{B}_\tau[\![z :: b]\!] = \lambda e\rho\,.\,\mathcal{Z}_\tau[\![z]\!]e\rho\,(\mathcal{B}_\tau[\![b]\!]e\rho)$

$\mathcal{Z}_\tau[\![\langle$`unless-false` $y_1\ y_2\rangle]\!] = \lambda e\rho\pi\,.\,\mathit{if\_truish}(\mathcal{Y}_\tau[\![y_1]\!]e\rho\pi)(\mathcal{Y}_\tau[\![y_2]\!]e\rho\pi)$

$\mathcal{Z}_\tau[\![\langle$`literal` $n\rangle]\!] = \lambda e\rho\,.\,\mathit{literal}\,(\mathcal{K}[\![(e(n))(1)]\!])$

$\mathcal{Z}_\tau[\![\langle$`closure` $n\rangle]\!] = \lambda e\rho\,.\,\mathit{closure}\,(\mathcal{T}_\tau[\![e(n)]\!]\rho)$

$\mathcal{Z}_\tau[\![\langle$`global` $n\rangle]\!] = \lambda e\rho\,.\,\mathit{global}\,(\mathit{lookup}\,\rho\,(e(n))(1))$

$\mathcal{Z}_\tau[\![\langle$`local` $n_0\ n_1\rangle]\!] = \lambda e\rho\,.\,\mathit{local}\,n_0\,n_1$

$\mathcal{Z}_\tau[\![\langle$`set-global!` $n\rangle]\!] = \lambda e\rho\,.\,\mathit{set\_global}\,(\mathit{lookup}\,\rho\,(e(n))(1))$

$\mathcal{Z}_\tau[\![\langle \texttt{set-local!}\ n_0\ n_1\rangle]\!] = \lambda e\rho\,.\,\mathit{set\_local}\,n_0\,n_1$

$\mathcal{Z}_\tau[\![\langle \texttt{push}\rangle]\!] = \lambda e\rho\,.\,\mathit{push}$

$\mathcal{Z}_\tau[\![\langle \texttt{make-env}\ n\rangle]\!] = \lambda e\rho\,.\,\mathit{make\_env}\,n$

$\mathcal{Z}_\tau[\![\langle \texttt{make-rest-list}\ n\rangle]\!] = \lambda e\rho\,.\,\mathit{make\_rest\_list}\,n$

$\mathcal{Z}_\tau[\![\langle \texttt{unspecified}\rangle]\!] = \lambda e\rho\,.\,\mathit{literal}\,\mathit{unspecified}$ in $\mathtt{E}$

$\mathcal{Z}_\tau[\![\langle \texttt{checkargs=}\ n\rangle]\!] = \lambda e\rho\,.\,\mathit{check\_args\_eq}\,n$

$\mathcal{Z}_\tau[\![\langle \texttt{checkargs>=}\ n\rangle]\!] = \lambda e\rho\,.\,\mathit{check\_args\_ge}\,n$

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

$\mathcal{Y}_\tau[\![z :: y']\!] = \lambda e\rho\pi\,.\,\mathcal{Z}_\tau[\![z]\!]e\rho\,(\mathcal{Y}_\tau[\![y']\!]e\rho\pi)$

$\mathcal{Y}_\tau[\![\langle\rangle]\!] = \lambda e\rho\pi\,.\,\pi$

$\mathcal{T}_\tau[\![\langle \texttt{template}\ b\ e\rangle]\!] = \mathcal{B}_\tau[\![b]\!]e$

Semantic Clauses for Primitives We will specify the denotational semantics for the most important data manipulation primitives. These are:

> `%%cwcc`, `%%apply`, `%%eqv`, `%%set-car!`,
> `%%car`, `%%add`, and `%%cons`.

Some other primitives, such as the vector and string primitives, can be easily handled in the same style. However, others, such as the I/O primitives `read-char` and `write-char`, are not so easily handled. This is because the official Scheme semantics, from which we have derived our byte code semantics, has not specified how I/O is to be handled. We consider I/O to be easier and more natural to express in the operational framework. In the operational context, the sequence of atomic actions makes it easier to specify the succession of I/O events.

$$\begin{array}{l}\mathcal{Z}_\tau[\![\langle \texttt{\%\%add}\rangle]\!] = \\ \quad \lambda e\rho\pi\,.\,\lambda\epsilon\epsilon^*\rho_R\psi\,. \\ \quad\ (\lambda\epsilon^*\,. \\ \quad\quad \pi(\mathit{sum\_vals}\,\epsilon^*)\langle\rangle\rho_R\psi) \\ \quad\quad (\mathit{rev}\,\epsilon^*)\end{array}$$

$$
\begin{array}{l}
\mathcal{Z}_\tau[\![\langle \texttt{\%\%eqv} \rangle]\!] = \\
\quad \lambda e \rho \pi \,.\, \lambda \epsilon \epsilon^* \rho_R \psi \,. \\
\quad\quad (\lambda \epsilon^* \,. \\
\quad\quad\quad \#\epsilon^* \geq 2 \rightarrow \\
\quad\quad\quad\quad \pi((\epsilon^* \, 0 = \epsilon^* \, 1 \rightarrow \mathit{true}, \mathit{false}) \text{ in } \texttt{E}) \\
\quad\quad\quad\quad\quad \langle\rangle \rho_R \psi, \\
\quad\quad\quad\quad \mathit{wrong} \text{ “bad arg count”}) \\
\quad\quad (\mathit{rev}\, \epsilon^*)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{Z}_\tau[\![\langle \texttt{\%\%car} \rangle]\!] = \\
\quad \lambda e \rho \pi \,.\, \lambda \epsilon \epsilon^* \rho_R \psi \sigma \,. \\
\quad\quad (\lambda \epsilon^* \,. \\
\quad\quad\quad \#\epsilon^* \geq 1 \rightarrow \\
\quad\quad\quad\quad \epsilon^* \, 0 : \texttt{E}_\text{p} \rightarrow \\
\quad\quad\quad\quad\quad \pi(\sigma((\epsilon^* \, 0 \mid \texttt{E}_\text{p}) 0)) \\
\quad\quad\quad\quad\quad\quad \langle\rangle \rho_R \psi \sigma, \\
\quad\quad\quad\quad\quad \mathit{wrong} \text{ “attempt to take car of non-pair”} \sigma, \\
\quad\quad\quad\quad \mathit{wrong} \text{ “bad arg count”} \sigma) \\
\quad\quad (\mathit{rev}\, \epsilon^*)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{Z}_\tau[\![\langle \texttt{\%\%cdr} \rangle]\!] = \\
\quad \lambda e \rho \pi \,.\, \lambda \epsilon \epsilon^* \rho_R \psi \sigma \,. \\
\quad\quad (\lambda \epsilon^* \,. \\
\quad\quad\quad \#\epsilon^* \geq 1 \rightarrow \\
\quad\quad\quad\quad \epsilon^* \, 0 : \texttt{E}_\text{p} \rightarrow \\
\quad\quad\quad\quad\quad \pi(\sigma((\epsilon^* \, 0 \mid \texttt{E}_\text{p}) 1)) \\
\quad\quad\quad\quad\quad\quad \langle\rangle \rho_R \psi \sigma, \\
\quad\quad\quad\quad\quad \mathit{wrong} \text{ “attempt to take cdr of non-pair”} \sigma, \\
\quad\quad\quad\quad \mathit{wrong} \text{ “bad arg count”} \sigma) \\
\quad\quad (\mathit{rev}\, \epsilon^*)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{Z}_\tau[\![\langle \texttt{\%\%set-car!} \rangle]\!] = \\
\quad \lambda e \rho \pi \,.\, \lambda \epsilon \epsilon^* \rho_R \psi \,. \\
\quad\quad (\lambda \epsilon^* \,. \\
\quad\quad\quad \#\epsilon^* \geq 2 \rightarrow \\
\quad\quad\quad\quad \epsilon^* \, 1 : \texttt{E}_\text{p} \rightarrow \\
\quad\quad\quad\quad\quad (\epsilon^* \, 1 \mid \texttt{E}_\text{p}) \, 2 = \mathit{mutable} \rightarrow \\
\quad\quad\quad\quad\quad\quad \mathit{assign}((\epsilon^* \, 1 \mid \texttt{E}_\text{p}) \, 0)(\epsilon^* \, 0) \\
\quad\quad\quad\quad\quad\quad\quad (\pi(\mathit{unspecified} \text{ in } \texttt{E}) \langle\rangle \rho_R \psi),
\end{array}
$$

$wrong$ "attempt to set car of immutable pair",
$wrong$ "attempt to set car of non-pair",
$wrong$ "bad arg count")
$(rev\, \epsilon^*)$

$\mathcal{Z}_\tau[\![\langle \texttt{\%\%set-cdr!} \rangle]\!] =$
$\lambda e \rho \pi \,.\, \lambda \epsilon \epsilon^* \rho_R \psi \,.$
$(\lambda \epsilon^* \,.$
$\# \epsilon^* \geq 2 \rightarrow$
$\epsilon^*\, 1 : \texttt{E}_\text{p} \rightarrow$
$(\epsilon^*\, 1 \mid \texttt{E}_\text{p})\, 2 = mutable \rightarrow$
$assign((\epsilon^*\, 1 \mid \texttt{E}_\text{p})\, 1)(\epsilon^*\, 0)$
$(\pi(unspecified \text{ in } \texttt{E})\langle\rangle \rho_R \psi),$
$wrong$ "attempt to set cdr of immutable pair",
$wrong$ "attempt to set cdr of non-pair",
$wrong$ "bad arg count")
$(rev\, \epsilon^*)$

$\mathcal{Z}_\tau[\![\langle \texttt{\%\%cons} \rangle]\!] =$
$\lambda e \rho \pi \,.\, \lambda \epsilon \epsilon^* \rho_R \psi \sigma \,.$
$(\lambda \epsilon^* \,.$
$\# \epsilon^* \geq 2 \rightarrow$
$(\lambda \sigma' \,.$
$\pi \langle new\, \sigma, new\, \sigma', mutable \rangle \text{ in } \texttt{E}$
$\langle\rangle \rho_R \psi (update(new\, \sigma')(\epsilon^*\, 0)\sigma'))$
$(update(new\, \sigma)(\epsilon^*\, 1)\sigma),$
$wrong$ "bad arg count"σ)
$(rev\, \epsilon^*)$

$\mathcal{Z}_\tau[\![\langle \texttt{\%\%apply} \rangle]\!] =$
$\lambda e \rho \pi \,.\, \lambda \epsilon \epsilon^* \rho_R \psi \sigma \,.$
$(\lambda \epsilon^* \,.$
$\# \epsilon^* \geq 2 \rightarrow$
$(\lambda \epsilon^* \epsilon \,.\, call(\# \epsilon^*) \epsilon \epsilon^* \rho_R \psi \sigma)$
$(list\_to\_seq\, (\epsilon^*\, 0)\, (\epsilon^* \dagger 1) \ddagger \# \epsilon^* - 2\, \sigma)(\epsilon^*\, (\# \epsilon^* - 1)),$
$wrong$ "bad arg count"σ)
$(rev\, \epsilon^*)$

$\mathcal{Z}_\tau[\![\langle \texttt{\%\%cwcc} \rangle]\!] =$
$\lambda e \rho \pi \,.\, \lambda \epsilon \epsilon^* \rho_R \psi \sigma \,.$

$\#\epsilon^* = 1 \rightarrow$
$\quad call\ 1\ (\epsilon^*\ 0)\langle make\_escape\ \psi\ \sigma\rangle\ \rho_R \psi$
$\quad\quad (update\ (new\ \sigma)\ (unspecified\ \text{in}\ \mathtt{E})\ \sigma),$
$\quad wrong$ "bad arg count"σ

Auxiliary Functions

$sum\_vals : \mathtt{E}^* \rightarrow \mathtt{E}$
$sum\_vals =$
$\quad fix\ \lambda f\ .\ \lambda \epsilon^*\ .\ \#\epsilon^* = 0 \rightarrow 0,$
$\quad\quad f(\epsilon^* \dagger 1) + (\epsilon^*\ 0)\ |\ \mathtt{R}$

For the definition of the apply primitive operator, we need an auxiliary function *list_to_seq*, and for the treatment of `call/cc` we need *make_escape*.

$list\_to\_seq : \mathtt{E} \rightarrow \mathtt{E}^* \rightarrow \mathtt{S} \rightarrow \mathtt{E}^*$
$list\_to\_seq =$
$\quad fix\ \lambda f\ .\ \lambda \epsilon \epsilon^* \sigma\ .$
$\quad\quad \epsilon = null\ \text{in}\ \mathtt{E} \rightarrow \epsilon^*,$
$\quad\quad\quad \epsilon : \mathtt{E}_\mathrm{p} \rightarrow$
$\quad\quad\quad\quad f(\sigma((\epsilon\ |\ \mathtt{E}_\mathrm{p})1))(\sigma((\epsilon\ |\ \mathtt{E}_\mathrm{p})0)) :: \epsilon^* \sigma,$
$\quad\quad\quad\quad wrong$ "apply with improper list"σ

$make\_escape : \mathtt{K}_1 \rightarrow \mathtt{S} \rightarrow \mathtt{F}$
$make\_escape =$
$\quad \lambda \psi \sigma\ .\ \langle new\ \sigma, single\_arg(\lambda \epsilon \kappa\ .\ \psi\ \epsilon)\rangle\ \text{in}\ \mathtt{E}$

$single\_arg : (\mathtt{E} \rightarrow \mathtt{K} \rightarrow \mathtt{C}) \rightarrow (\mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C})$
$single\_arg =$
$\quad \lambda \zeta \epsilon^* \kappa\ .\ \#\epsilon^* = 1 \rightarrow \zeta(\epsilon^*\ 0)\kappa,$
$\quad\quad\quad\quad wrong$ "wrong number of arguments"

3 Byte Code Compiler Algorithm

In this section, we present the algorithm for the compiler, expressed as a translation in a style reminiscent of natural semantics. Interspersed with the translation, we include the Scheme procedures that implement the compiler. We use two programs to maintain the TEX source for this chapter and the Scheme source for the compiler from a single file, following the general approach of Knuth's "literate programming" [3]. The Scheme programs appear in typewriter font.

The compiler algorithm picks one value for the *permute* and *unpermute* functions used in the semantics of Scheme: namely, it uses the function that evaluates arguments left-to-right and then evaluates the procedure last. So *permute* is the cyclic permutation that advances each sequence element after the first, and places the first element at the end. The *unpermute* function is its inverse: it places the last element of its argument at the head, and shifts each other lelement back one place. As a result, in operational terms, the argument stack will have the rightmost operand of a combination at its top when the combination's operator is invoked.

In this chapter, we will use a bar to distinguish variables over the targets of a translation, which are elements of the byte code syntax such as $\bar{e}$, from variables over source expressions, which are Scheme expressions such as e.

We will also refer to syntactic entities in a more concrete and traditional notation that facilitates comparing the clauses of the algorithm with the applicative Scheme procedures that implement it.

```
(define (compile exp)
  (emit-lap #f (comp-body exp mt-cenv #f)))

(define (compile-library library-exp)
  (emit-lap
   #f
   (append
    primop-initialization-code
    (comp-body library-exp mt-cenv #f))))

(define (comp-body exp cenv name)
  (comp exp cenv 0 name return-code))

(define (comp exp cenv nargs name after)
  (cond ((id? exp)
         (compile-id exp cenv nargs name after))
```

```
        ((self-quoting-constant? exp)
         (compile-constant exp cenv nargs name after))
        ((pair? exp)
         (case (car exp)
           ((quote)
            (compile-quote exp cenv nargs name after))
           ((begin)
            (compile-begin (cdr exp) cenv nargs name after))
           ((lambda)
            (compile-lambda exp cenv nargs name after))
           ((if)
            (compile-if exp cenv nargs name after))
           ((set!)
            (compile-set exp cenv nargs name after))
           (else
            (compile-application exp cenv nargs name after))))
        (else
         (compiler-error 'comp "unrecognized expression" exp))))
```

3.1 Expressions

```
(define (compile-constant exp cenv nargs name after)
  (compiler-prepend-instruction
   (make-instruction 'literal exp)
   after))

(define (compile-quote exp cenv nargs name after)
  (compiler-prepend-instruction
   (make-instruction 'literal (cadr exp))
   after))

(define (compile-id exp cenv nargs name after)
  (let ((local-ref (lookup exp cenv)))
    (if (pair? local-ref)
        (compiler-prepend-instruction
         (make-instruction 'local (car local-ref) (cdr local-ref))
         after)
        (compiler-prepend-instruction
         (make-instruction 'global exp)
         after))))

(define (compile-application exp cenv nargs name after)
  (if (return? after)
      (let* ((proc (car exp))
```

```
            (args (cdr exp))
            (nargs (length args)))
       (comp-args args cenv 0 name
                  (comp proc cenv nargs name
                        (instruction->code-sequence
                         (make-instruction 'call nargs)))))
     (compiler-prepend-instruction
      (make-instruction 'make-cont after nargs)
      (compile-application exp cenv 0 name return-code))))

(define (compile-lambda exp cenv nargs name after)
  (compiler-prepend-instruction
   (make-instruction
    'closure
    (emit-lap
     name
     (cond ((thunk? exp)
            (compile-thunk exp cenv nargs name))
           ((undotted-lambda? exp)
            (compile-undotted-lambda exp cenv nargs name))
           ((dotted-lambda? exp)
            (compile-dotted-lambda exp cenv nargs name))
           (else (compiler-error
                  'compile-lambda "bogus lambda" exp)))))
   after))

(define (compile-thunk exp cenv nargs name)
  (let ((body (caddr exp)))             ; body is a single form
    (comp-entry
     0
     (comp-body body cenv               ; avoid changing environment
                name))))

(define (compile-undotted-lambda exp cenv nargs name)
  (let ((formals (cadr exp))
        (body (caddr exp)))             ; body is a single form
    (let ((rev-formals
           (reverse formals)))
      (comp-entry
       (length rev-formals)
       (comp-body
        body
        (extend-cenv cenv rev-formals)
        name)))))
```

```
(define (compile-dotted-lambda exp cenv nargs name)
  (let ((formals (cadr exp))
        (body (caddr exp)))
    (let ((rev-formals
           (reverse-proper-or-improper-list formals)))
      (comp-rest-entry
       (- (length rev-formals)          ; required args
          1)
       (comp-body
        body
        (extend-cenv cenv rev-formals)
        name)))))

(define (comp-entry nargs after)
  (if (= nargs 0)
      (compiler-prepend-instruction
       (make-instruction 'check-args= nargs)
       after)
      (compiler-prepend-instruction
       (make-instruction 'check-args= nargs)
       (compiler-prepend-instruction
        (make-instruction 'make-env nargs)
        after))))

(define (comp-rest-entry nargs after)
  (let ((unchecked-code
         (compiler-prepend-instruction
          (make-instruction 'make-rest-list nargs)
          (compiler-prepend-instruction
           (make-instruction 'push)
           (compiler-prepend-instruction
            (make-instruction 'make-env (+ 1 nargs))
            after)))))
    (if (= nargs 0)
        unchecked-code                  ; no point checking nargs
        (compiler-prepend-instruction
         (make-instruction 'check-args>= nargs)
         unchecked-code))))

(define (compile-if exp cenv nargs name after)
  (compile-if-or-when
   (cadr exp)
   (caddr exp)
   (cdddr exp)
   cenv nargs name after))
```

```
(define (compile-alt alt cenv nargs name after)
  (if (null? alt)
      (compiler-prepend-instruction
       (make-instruction 'unspecified)
       after)
      (comp (car alt) cenv nargs name after)))

(define (compile-if-or-when test con alt
                            cenv nargs name after)
  (comp test cenv nargs name
        (if (return? after)
            (instruction->code-sequence
             (make-instruction
              'unless-false
              (comp con cenv nargs name after)
              (compile-alt alt cenv nargs name after)))
            (compiler-prepend-instruction
             (make-instruction
              'unless-false
              (comp
               con cenv nargs name empty-open-code-sequence)
              (compile-alt
               alt cenv nargs name empty-open-code-sequence))
             after))))

(define (compile-set exp cenv nargs ignored after)
  (let ((name (cadr exp))
        (exp (caddr exp)))
    (if
     (id? name)
     (let ((local-ref (lookup name cenv)))
       (comp exp cenv nargs name
             (if (pair? local-ref)
                 (compiler-prepend-instruction
                  (make-instruction
                   'set-local!
                   (car local-ref)
                   (cdr local-ref))
                  after)
                 (compiler-prepend-instruction
                  (make-instruction 'set-global! name)
                  after))))
     (compiler-error
      'compile-set "target to set! not identifier: " name))))
```

3.2 Commands

```
(define (compile-begin commands cenv nargs name after)
  (let ((first-command (car commands))
        (remaining-commands (cdr commands)))
    (comp first-command cenv nargs name
          (if (null? remaining-commands)
              after
              (compile-begin
               remaining-commands
               cenv nargs name after)))))
```

3.3 Arguments

```
(define (comp-args args cenv nargs name after)
  (if (null? args)
      after
      (comp (car args)
            cenv nargs name
            (compiler-prepend-instruction
             (make-instruction 'push)
             (comp-args (cdr args) cenv
                        (+ 1 nargs)
                        name after)))))
```

3.4 Byte Code Data Type

```
(define (emit-lap name after) `(lap ,name . ,after))

(define return-code '((return)))

(define empty-open-code-sequence '())

(define (return? code)
  (and (pair? code)
       (eq? (caar code) 'return)))

(define compiler-prepend-instruction cons)

(define make-instruction list)

(define (instruction->code-sequence instruction)
  (list instruction))
```

3.5 Environments

```
(define (lookup exp cenv)
  (cond ((not (id? exp))
         (compiler-error
          'lookup
          "expression not an identifier"
          exp))
        ((keyword? exp)
         (compiler-error
          'lookup
          "keyword used as a variable"
          exp))
        (else (lookup-loop exp cenv 0))))

(define (lookup-loop id cenv back)
  (if (null? cenv)
      'not-lexical
      (let loop ((rib (cdr cenv)) (over 1))
        (cond ((null? rib)
               (lookup-loop id (car cenv) (+ 1 back)))
              ((eq? id (car rib)) (cons back over))
              (else (loop (cdr rib) (+ 1 over)))))))

(define extend-cenv cons)
(define mt-cenv '())
```

A few auxiliary procedures are still needed:

```
(define (thunk? lambda-exp)
  (null? (cadr lambda-exp)))

(define (undotted-lambda? lambda-exp)
  (proper-list? (cadr lambda-exp)))

(define (dotted-lambda? lambda-exp)
  (not (proper-list? (cadr lambda-exp))))

(define (reverse-proper-or-improper-list p)
  (let loop ((accum '())
             (p p))
    (cond ((pair? p) (loop (cons (car p) accum) (cdr p)))
          ((null? p) accum)
          (else (cons p accum)))))
```

```
(define (proper-list? p)
  (or (null? p)
      (and (pair? p)
           (proper-list? (cdr p)))))

(define (keyword? id)
  (memq id '(=> and begin case cond define do else if lambda
             let let* letrec or quasiquote quote set! unquote
             unquote-splicing)))

(define (self-quoting-constant? exp)
  (or (number? exp)
      (boolean? exp)
      (char? exp)
      (string? exp)))

(define (id? exp)
  (and (symbol? exp)
       (not (keyword? exp))))
```

3.6 Opcode Table and Compiler Support for Primitives

The system needs a table to summarize various pieces of information about the BBC instructions. We also associate a small integer with each operation name; the running image actually contains these integers (each represented in one byte) rather than the operation names.

```
(define *opcode-info* (make-vector 256 '(error 0)))
(define (define-opcode-info opcode opname num-params)
  (vector-set! *opcode-info* opcode (list opname num-params)))
(define instr-start 0)
(define-opcode-info 0  'call 1)
(define-opcode-info 1  'return 0)
(define-opcode-info 2  'make-cont 3)
(define-opcode-info 3  'literal 1)
(define-opcode-info 4  'closure 1)
(define-opcode-info 5  'global 1)
(define-opcode-info 6  'local 2)
(define-opcode-info 7  'set-global! 1)
(define-opcode-info 8  'set-local! 2)
(define-opcode-info 9  'push 0)
(define-opcode-info 10 'make-env 1)
(define-opcode-info 11 'make-rest-list 1)
(define-opcode-info 12 'unspecified 0)
```

```
(define-opcode-info 13 'jump 2)
(define-opcode-info 14 'jump-if-false 2)
(define-opcode-info 15 'check-args= 1)
(define-opcode-info 16 'check-args>= 1)
(define-opcode-info 17 'primitive-throw 0)
(define-opcode-info 18 'empty 0)
(define instr-end 18)
(define prim-start 22)
(define-opcode-info
  22 '%%call-with-current-continuation '(check-args= 1))
(define-opcode-info 23 '%%force-output '(check-args= 1))
(define-opcode-info 24 '%%find-symbol-table '(check-args= 0))
(define-opcode-info 25 '%%* '(check-args>= 0))
(define-opcode-info 26 '%%+ '(check-args>= 0))
(define-opcode-info 27 '%%- '(check-args>= 1))
(define-opcode-info 28 '%%< '(check-args>= 2))
(define-opcode-info 29 '%%= '(check-args>= 2))
(define-opcode-info 30 '%%apply '(check-args>= 2))
(define-opcode-info 31 '%%car '(check-args= 1))
(define-opcode-info 32 '%%cdr '(check-args= 1))
(define-opcode-info 33 '%%char->integer '(check-args= 1))
(define-opcode-info 34 '%%char<? '(check-args= 2))
(define-opcode-info 35 '%%char=? '(check-args= 2))
(define-opcode-info 36 '%%char? '(check-args= 1))
(define-opcode-info 37 '%%close-input-port '(check-args= 1))
(define-opcode-info 38 '%%close-output-port '(check-args= 1))
(define-opcode-info 39 '%%procedure? '(check-args= 1))
(define-opcode-info 40 '%%cons '(check-args= 2))
(define-opcode-info 41 '%%current-input-port '(check-args= 0))
(define-opcode-info 42 '%%current-output-port '(check-args= 0))
(define-opcode-info 43 '%%eof-object? '(check-args= 1))
(define-opcode-info 44 '%%eq? '(check-args= 2))
(define-opcode-info 45 '%%abort '(check-args= 0))
(define-opcode-info 46 '%%integer? '(check-args= 1))
(define-opcode-info 47 '%%input-port? '(check-args= 1))
(define-opcode-info 48 '%%integer->char '(check-args= 1))
(define-opcode-info 49 '%%make-string '(check-args>= 1))
(define-opcode-info 50 '%%make-symbol '(check-args= 1))
(define-opcode-info 51 '%%make-vector '(check-args>= 1))
(define-opcode-info 52 '%%open-input-file '(check-args= 1))
(define-opcode-info 53 '%%open-output-file '(check-args= 1))
(define-opcode-info 54 '%%output-port? '(check-args= 1))
(define-opcode-info 55 '%%pair? '(check-args= 1))
(define-opcode-info 56 '%%peek-char '(check-args>= 0))
```

```
(define-opcode-info 57 '%%quotient '(check-args= 2))
(define-opcode-info 58 '%%read-char '(check-args>= 0))
(define-opcode-info 59 '%%remainder '(check-args= 2))
(define-opcode-info 60 '%%set-car! '(check-args= 2))
(define-opcode-info 61 '%%set-cdr! '(check-args= 2))
(define-opcode-info 62 '%%string-length '(check-args= 1))
(define-opcode-info 63 '%%string-ref '(check-args= 2))
(define-opcode-info 64 '%%string-set! '(check-args= 3))
(define-opcode-info 65 '%%string=? '(check-args= 2))
(define-opcode-info 66 '%%string? '(check-args= 1))
(define-opcode-info 67 '%%symbol->string '(check-args= 1))
(define-opcode-info 68 '%%symbol? '(check-args= 1))
(define-opcode-info 69 '%%unspecified '(check-args= 0))
(define-opcode-info 70 '%%error '(check-args= 0))
(define-opcode-info 71 '%%error '(check-args= 0))
(define-opcode-info 72 '%%vector-length '(check-args= 1))
(define-opcode-info 73 '%%vector-ref '(check-args= 2))
(define-opcode-info 74 '%%vector-set! '(check-args= 3))
(define-opcode-info 75 '%%vector? '(check-args= 1))
(define-opcode-info 76 '%%write-char '(check-args>= 1))
(define-opcode-info 77 '%%write-string '(check-args= 2))
(define prim-end 77)
```

A few auxiliary procedures are used to access the opcode information in the table.

```
(define (vector-search vector key start end)
  (if (> start end) #f
      (let ((elem (vector-ref vector start)))
        (if (eq? key (car elem))
            (cons start elem)
            (vector-search vector key (+ start 1) end)))))

(define (opname->opcode name)
  (let ((info
         (vector-search *opcode-info* name instr-start prim-end)))
    (if info (car info)
        (compiler-error 'image-builder 'opname->opcode
                        "invalid instruction name" name))))

(define (opcode->opname opcode)
  (car (vector-ref *opcode-info* opcode)))

(define (num-of-opcode-args opcode)
  (if (> opcode instr-end) 0
      (cadr (vector-ref *opcode-info* opcode))))
```

```
(define (opcode->argument-checking-code opcode)
  (if (and (<= prim-start opcode)
           (<= opcode prim-end))
      (cadr (vector-ref *opcode-info* opcode))
      (compiler-error
       'opcode->argument-checking-code
       "opcode out of bounds:" opcode)))
```

Finally, the compiler uses two procedures to generate the code for the primitives. This code consists of `closure` instructions that provide the initial procedure values for the Scheme identifiers that access virtual machine primitive operations.

```
(define (opcode->standard-lap opcode)
  (let* ((opname (opcode->opname opcode))
         (argument-checking-code
          (opcode->argument-checking-code opcode)))
    `((closure
       (lap ,opname ,argument-checking-code (,opname) (return)))
      (set-global! ,opname))))

(define primop-initialization-code
  (do ((i prim-start (+ i 1))
       (primop-initialization-code
        '()
        (append
         (opcode->standard-lap i)
         primop-initialization-code)))
      ((= i (+ prim-end 1))
       primop-initialization-code)))
```

4 Compiler Correctness

4.1 Syntactic Correctness

In this section, we prove that if the byte code compiler is applied to a syntactically correct Scheme input program, its output is a syntactically correct basic byte code template.

If e is a Scheme expression, we will use $C(e)$ to refer to the result of calling the procedure `compile` on e. We will also use $C(e, \rho_C, n, i, w)$ to refer to the result of calling the procedure `comp` on e together with the compile-time environment ρ_C, the natural number n, the name i, and the after-code w (assumed to be a BBC instruction sequence). We will refer to the empty compile-time environment as MT_C. It is the function $\lambda\nu_1\nu_2$. *not_lexical*.

Theorem 2 (Compiler syntactic correctness)

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

Proof. The proof is by simultaneous structural induction on the Scheme syntax.

1. Since the return code, $\langle\langle\texttt{return}\rangle\rangle$, is a closed instruction list b_0,

   $$C(e) = \langle \texttt{lap}\ \text{FALSE}\ C(e, \text{MT}_C, \text{FALSE}, b_0)\rangle$$

   IH clause 2 entails that $C(e, \text{MT}_C, \text{FALSE}, b_0)$ is a closed instruction list b_1, so $\langle \texttt{lap}\ \text{FALSE}\ b_1\rangle$ is a syntactically correct template.

2. $C(e, \rho_C, n, i, b_0)$ is the result of calling one of eight procedures on the same arguments (or in the case of `compile-begin`, on the embedded sequence of commands). Hence the case follows from the eight lemmas below.

3. $C(e, \rho_C, n, i, y_0)$ is the result of calling one of eight procedures on the same arguments (or in the case of `compile-begin`, on the embedded sequence of commands). Hence the case follows from the eight lemmas below.

4. $C(e, \rho_C, n, i, \langle\rangle)$ is the result of calling one of eight procedures on the same arguments (or in the case of `compile-begin`, on the embedded sequence of commands). Hence the case follows from the eight lemmas below.

In the following eight lemmas, we will use $C_{category}$ to refer to the result, for suitable arguments, of applying the compilation procedure for a particular syntactic category. For instance, C_{id} is the procedure to compile an identifier, with auxiliary arguments ρ_C, n, i, w_0 or $\rho_C, n, i, \langle\rangle$. We will slightly abuse notation by letting w_0 stand for any open or closed instruction sequence or else $\langle\rangle$. By "the right syntactic category," we will mean a closed instruction list if the last argument is a closed instruction list, and an open instruction list if the last argument is either a closed instruction list or else $\langle\rangle$.

Lemma 3 (Compile Identifier)

1. *$C_{id}(i, \rho_C, n, i, b_0)$ is a closed instruction list b_1;*
2. *$C_{id}(i, \rho_C, n, i, y_0)$ is an open instruction list y_1;*
3. *$C_{id}(i, \rho_C, n, i, \langle\rangle)$ is an open instruction list y_1.*

Proof. Since *lookup* returns a pair $\langle n_1, n_2\rangle$ of natural numbers if it returns a pair, $\langle$`local` $n_1\ n_2\rangle$ is a neutral instruction z. $\langle$`global` $i\rangle$ is also a neutral instruction z. So the result is an instruction sequence of the right syntactic class, namely (respectively):

1. $z :: b_0$;
2. $z :: y_0$;
3. $z :: \langle\rangle = \langle z\rangle$.

Lemma 4 (Compile Constant)

1. *$C_{constant}(c_{sq}, \rho_C, n, i, b_0)$ is a closed instruction list b_1;*
2. *$C_{constant}(c_{sq}, \rho_C, n, i, y_0)$ is an open instruction list y_1;*
3. *$C_{constant}(c_{sq}, \rho_C, n, i, \langle\rangle)$ is an open instruction list y_1*

Proof. Similar.

Lemma 5 (Compile Quote)

1. $C_{quote}(\langle \texttt{quote}\ c\rangle, \rho_C, n, i, b_0)$ *is a closed instruction list* b_1*;*
2. $C_{quote}(\langle \texttt{quote}\ c\rangle, \rho_C, n, i, y_0)$ *is an open instruction list* y_1*;*
3. $C_{quote}(\langle \texttt{quote}\ c\rangle, \rho_C, n, i, \langle\rangle)$ *is an open instruction list* y_1*.*

Proof. Similar.

Lemma 6 (Compile Begin)
Suppose that Theorem 2 holds for all proper subexpressions of

$$e = \langle \texttt{begin}\rangle^\frown e^+,$$

and all instruction sequences b_0*,* y_0*, or* $\langle\rangle$*. Then:*

1. $C_{begin}(e^+, \rho_C, n, i, b_0)$ *is a closed instruction list* b_1*;*
2. $C_{begin}(e^+, \rho_C, n, i, y_0)$ *is an open instruction list* y_1*;*
3. $C_{begin}(e^+, \rho_C, n, i, \langle\rangle)$ *is an open instruction list* y_1*;*

Proof. The proof is by induction on e^+.

The base case $e^* = \langle e_0\rangle$ holds:

$$C_{begin}(\langle e_0\rangle, \rho_C, n, i, w_0) = C(e_0, \rho_C, n, i, w_0),$$

which by the assumption that Theorem 2 holds is of the right syntactic category.

Otherwise, $e^* = e_0 :: e_1^+$, and the induction hypothesis is that

$$C_{begin}(e_1^+, \rho_C, n, i, w_0)$$

is of the same syntactic class as w_0. But

$$C_{begin}(e_0 :: e_1^+, \rho_C, n, i, w_0) = C(e_0, \rho_C, n, i, C_{begin}(e_1^+, \rho_C, n, i, w_0)).$$

Hence, applying clause 2 or 3 of Theorem 2 to the proper subexpression e_0, we may infer that this belongs to the same syntactic class as its last argument, which by the induction hypothesis is the right syntactic class.

Lemma 7 (Compile Lambda)
Suppose that Theorem 2 holds for all proper subexpressions of

$$e = \langle \texttt{lambda}\ i^*\ e_0 \rangle,$$

all values ρ'_C, and all instruction sequences w. Then:

1. *$C_\lambda(e, \rho_C, n, i, b_0)$ is a closed instruction list b_1;*
2. *$C_\lambda(e, \rho_C, n, i, y_0)$ is an open instruction list y_1;*
3. *$C_\lambda(e, \rho_C, n, i, \langle\rangle)$ is an open instruction list y_1.*

Proof. Since C_λ simply prepends a `closure` instruction to its last argument, it suffices to ensure that each branch of the embedded conditional returns a closed instruction list b_λ when called with arguments e, ρ_C, n, i.

First note that if `comp-entry` or `comp-rest-entry` is called with a natural number and after code w_0, then it returns a code sequence w_1 of the right syntactic category. This is because the procedures simply prepend neutral instructions.

1. e is a thunk (i.e. a procedure with zero parameters) $\langle \texttt{lambda}\ \langle\rangle\ e_0 \rangle$:

   In this case we call `comp-entry` on the result of calling `comp-body` with arguments e_0, ρ_C, i. As the latter returns $C(e_0, \rho_C, 0, i, \langle\langle \texttt{return} \rangle\rangle)$, we may apply Theorem 2 to the subexpression e_0.

2. e is an undotted lambda expression $\langle \texttt{lambda}\ \mathrm{I}^*\ e_0 \rangle$:

   Similar, except that a different lexical environment argument ρ'_C is passed into `comp-body`.

3. e is a dotted lambda expression $\langle \texttt{dotted\_lambda}\ \mathrm{I}^+\ e_0 \rangle$:

   Similar to the preceding, except that that `comp-rest-entry` is called, and the reversed formals and number of arguments are computed slightly differently.

Lemma 8 (Compile If)
Suppose that Theorem 2 holds for all proper subexpressions of

$$e = \langle \texttt{if} \rangle^\frown e^*,$$

and all instruction sequences w. Then:

1. $C_{if}(e, \rho_C, n, i, b_0)$ *is a closed instruction list* b_1*;*
2. $C_{if}(e, \rho_C, n, i, y_0)$ *is an open instruction list* y_1*;*
3. $C_{if}(e, \rho_C, n, i, \langle\rangle)$ *is an open instruction list* y_1*.*

Proof.

1. We will divide this into four cases, depending whether the conditional has an alternative, and whether the after-code $b_0 = \langle\langle\texttt{return}\rangle\rangle$ or not.

   (a) Consider first the case in which $e = \langle\texttt{if}\ e_0\ e_1\ e_2\rangle$ and $b_0 = \langle\langle\texttt{return}\rangle\rangle$.

$$\begin{aligned} C_{if}(e, \rho_C, n, i, w_0) &= C_{\textit{if-or-when}}(e_0, e_1, \langle e_2\rangle, \rho_C, n, i, b_0) \\ &= C(e_0, \rho_C, n, i, \\ &\quad \langle\langle\texttt{unless-false} \\ &\quad\quad C(e_1, \rho_C, n, i, b_0) \\ &\quad\quad C_{alt}(\langle e_2\rangle, \rho_C, n, i, b_0)\rangle\rangle) \\ &= C(e_0, \rho_C, n, i, \\ &\quad \langle\langle\texttt{unless-false} \\ &\quad\quad C(e_1, \rho_C, n, i, b_0) \\ &\quad\quad C(e_2, \rho_C, n, i, b_0)\rangle\rangle) \end{aligned}$$

   We now apply Theorem 2 Case 2 to the subexpressions e_1 and e_2, inferring that

$$\langle\langle\texttt{unless-false}\ C(e_1, \rho_C, n, i, b_0)\ C(e_2, \rho_C, n, i, b_0)\rangle\rangle$$

   is a closed instruction sequence b_1. We next apply the theorem (again, Case 2) to e_0 (and the after-code b_1).

   (b) Consider next the case in which $e = \langle\texttt{if}\ e_0\ e_1\ e_2\rangle$, but $b_0 \neq \langle\langle\texttt{return}\rangle\rangle$.

$$\begin{aligned} C_{if}(e, \rho_C, n, i, w_0) &= C_{\textit{if-or-when}}(e_0, e_1, \langle e_2\rangle, \rho_C, n, i, b_0) \\ &= C(e_0, \rho_C, n, i, \\ &\quad \langle\texttt{unless-false} \\ &\quad\quad C(e_1, \rho_C, n, i, \langle\rangle) \end{aligned}$$

$$
\begin{array}{rcl}
 & & C_{alt}(\langle e_2\rangle, \rho_C, n, i, \langle\rangle)\rangle \\
 & & :: b_0) \\
 & = & C(e_0, \rho_C, n, i, \\
 & & \quad \langle \texttt{unless-false} \\
 & & \quad\quad C(e_1, \rho_C, n, i, \langle\rangle) \\
 & & \quad\quad C(e_2, \rho_C, n, i, \langle\rangle)\rangle \\
 & & \quad :: b_0)
\end{array}
$$

We now apply Theorem 2 Case 4 to the subexpressions e_1 and e_2, inferring that

$$\langle \texttt{unless-false}\ C(e_1, \rho_C, n, i, \langle\rangle)\ C(e_2, \rho_C, n, i, \langle\rangle)\rangle$$

is a neutral instruction z, so that $z :: b_0$ is a closed instruction sequence b_1. We next apply the theorem (again, Case 2) to e_0 (and the after-code b_1).

(c) Consider next the case in which $e = \langle \texttt{if}\ e_0\ e_1\rangle$, and $b_0 = \langle\langle \texttt{return}\rangle\rangle$.

$$
\begin{array}{rcl}
C_{if}(e, \rho_C, n, i, w_0) & = & C_{\textit{if-or-when}}(e_0, e_1, \langle\rangle, \rho_C, n, i, b_0) \\
 & = & C(e_0, \rho_C, n, i, \\
 & & \quad \langle\langle \texttt{unless-false} \\
 & & \quad\quad C(e_1, \rho_C, n, i, b_0) \\
 & & \quad\quad C_{alt}(\langle\rangle, \rho_C, n, i, b_0)\rangle\rangle) \\
 & = & C(e_0, \rho_C, n, i, \\
 & & \quad \langle\langle \texttt{unless-false} \\
 & & \quad\quad C(e_1, \rho_C, n, i, b_0) \\
 & & \quad\quad \langle \texttt{unspecified}\rangle :: b_0\rangle\rangle)
\end{array}
$$

The remainder of this case is similar to 1(a).

(d) Consider finally the case in which $e = \langle \texttt{if}\ e_0\ e_1\rangle$, and $b_0 \neq \langle\langle \texttt{return}\rangle\rangle$.

$$
\begin{array}{rcl}
C_{if}(e, \rho_C, n, i, w_0) & = & C_{\textit{if-or-when}}(e_0, e_1, \langle\rangle, \rho_C, n, i, b_0) \\
 & = & C(e_0, \rho_C, n, i, \\
 & & \quad \langle \texttt{unless-false}
\end{array}
$$

$$
\begin{array}{rcl}
 & & C(e_1, \rho_C, n, i, \langle\rangle) \\
 & & C_{alt}(\langle\rangle, \rho_C, n, i, \langle\rangle)\rangle \\
 & & :: b_0) \\
= & & C(e_0, \rho_C, n, i, \\
 & & \quad \langle \texttt{unless-false} \\
 & & \quad\quad C(e_1, \rho_C, n, i, \langle\rangle) \\
 & & \quad\quad \langle\langle \texttt{unspecified} \rangle\rangle \\
 & & \quad :: b_0)
\end{array}
$$

The remainder of this case is similar to 1(b).

2,3. In the remaining cases, in which the after-code w_0 is either an open instruction sequence y_0 or else $\langle\rangle$, we need not consider the possibility that $w_0 = \langle\langle \texttt{return} \rangle\rangle$. Thus when $e = \langle \texttt{if}\ e_0\ e_1\ e_2 \rangle$, we have:

$$
\begin{array}{rcl}
C_{if}(e, \rho_C, n, i, w_0) & = & C_{\textit{if-or-when}}(e_0, e_1, \langle e_2 \rangle, \rho_C, n, i, w_0) \\
 & = & C(e_0, \rho_C, n, i, \\
 & & \quad \langle \texttt{unless-false} \\
 & & \quad\quad C(e_1, \rho_C, n, i, \langle\rangle) \\
 & & \quad\quad C_{alt}(\langle e_2 \rangle, \rho_C, n, i, \langle\rangle)\rangle \\
 & & \quad :: w_0) \\
 & = & C(e_0, \rho_C, n, i, \\
 & & \quad \langle \texttt{unless-false} \\
 & & \quad\quad C(e_1, \rho_C, n, i, \langle\rangle) \\
 & & \quad\quad C(e_2, \rho_C, n, i, \langle\rangle)\rangle \\
 & & \quad :: w_0)
\end{array}
$$

We now apply Theorem 2 Case 4 to the subexpressions e_1 and e_2, inferring that

$$\langle \texttt{unless-false}\ C(e_1, \rho_C, n, i, \langle\rangle)\ C(e_2, \rho_C, n, i, \langle\rangle)\rangle$$

is a neutral instruction z, so that $z :: w_0$ is an open instruction sequence y_0. We apply Clause 3 of the theorem.

Finally, when $e = \langle \texttt{if}\ e_0\ e_1\ e_2 \rangle$:

$$C_{if}(e, \rho_C, n, i, w_0) \quad = \quad C_{\textit{if-or-when}}(e_0, e_1, \langle\rangle, \rho_C, n, i, w_0)$$

$$
\begin{aligned}
&= C(e_0, \rho_C, n, i, \\
&\quad \langle \texttt{unless-false} \\
&\quad\quad C(e_1, \rho_C, n, i, \langle\rangle) \\
&\quad\quad C_{alt}(\langle\rangle, \rho_C, n, i, \langle\rangle)\rangle \\
&\quad :: w_0) \\
&= C(e_0, \rho_C, n, i, \\
&\quad \langle \texttt{unless-false} \\
&\quad\quad C(e_1, \rho_C, n, i, \langle\rangle) \\
&\quad\quad \langle\langle \texttt{unspecified} \rangle\rangle \\
&\quad :: w_0)
\end{aligned}
$$

Since $\langle \texttt{unless-false}\ C(e_1, \rho_C, n, i, \langle\rangle)\ \langle\langle \texttt{unspecified} \rangle\rangle\rangle$ is a neutral instruction z, $z :: w_0$ is an open instruction sequence y_0. We again apply Clause 3 of the theorem.

Lemma 9 (Compile Set!)
Suppose that Theorem 2 holds for all proper subexpressions of

$$e = \langle \texttt{set!}\ i\ e_0 \rangle,$$

and all instruction sequences w. Then:

1. $C_{set!}(e, \rho_C, n, i, b_0)$ *is a closed instruction list* b_1*;*
2. $C_{set!}(e, \rho_C, n, i, y_0)$ *is an open instruction list* y_1*;*
3. $C_{set!}(e, \rho_C, n, i, \langle\rangle)$ *is an open instruction list* y_1*.*

Proof. Since *lookup* returns a pair $\langle n_1, n_2 \rangle$ of natural numbers if it returns a pair, $\langle \texttt{set-local!}\ n_1\ n_2 \rangle$ is a neutral instruction z. $\langle \texttt{set-global!}\ i \rangle$ is also a neutral instruction z. So $z :: w_0$ is an instruction sequence of the right syntactic class. Hence, we may apply Clause 2 or Clause 3 of Theorem 2 to the subexpression e with the aftercode $z :: w_0$.

Lemma 10 (Compile Application)
Suppose that Theorem 2 holds for all proper subexpressions of

$$e = e_0 :: e^*,$$

and all instruction sequences w. Then:

1. $C_{application}(e, \rho_C, n, i, b_0)$ *is a closed instruction list* b_1*;*
2. $C_{application}(e, \rho_C, n, i, y_0)$ *is an open instruction list* y_1*;*
3. $C_{application}(e, \rho_C, n, i, \langle\rangle)$ *is an open instruction list* y_1*.*

Proof. First assume that the after-code $w_0 = \langle\langle \texttt{call}\ n\rangle\rangle$. Since the latter is a closed instruction sequence, we may apply Theorem 2, Clause 2 to infer that $C(e_0, \rho_C, n, i, \langle\langle \texttt{call}\ n\rangle\rangle)$ is a closed instruction list b. To prove that $C_{args}(e^*, \rho_C, n, i, b)$ is a closed instruction list, we argue by induction on e^*.

$$C_{args}(\langle\rangle, \rho_C, n, i, b) = b.$$

Moreover, let $e^* = e_1 :: e_1{}^*$. Then

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

Inductively, the latter is of the form $C(e_1, \rho_C, n, i, \langle \texttt{push}\rangle :: b')$, so we may apply Theorem 2, Clause 2.

Next suppose that the after-code w_0 is an instruction sequence other than $\langle\langle \texttt{call}\ n\rangle\rangle$. Then

$$C_{application}(e, \rho_C, n, i, w_0)$$

is of the form:

$$\langle \texttt{make-cont}\ w_0\ n\rangle :: C_{application}(e, \rho_C, 0, i, \langle\langle \texttt{call}\ n\rangle\rangle).$$

Since we have just seen that the latter evaluates to a closed instruction list b_1, we may infer that $\langle \texttt{make-cont}\ w_0\ n\rangle :: b_1$ is an instruction list, and of the same syntactic class as w_0.

Finally, suppose the after-code is $\langle\rangle$. Then $C_{application}(e, \rho_C, n, i, w_0)$ is of the form:

$$\langle \texttt{make-cont}\ \langle\rangle\ n\rangle :: C_{application}(e, \rho_C, 0, i, \langle\langle \texttt{call}\ n\rangle\rangle).$$

We may again infer that the latter is of the form b_1, so that

$$\langle \texttt{make-cont}\ \langle\rangle\ n\rangle :: b_1$$

is an open instruction list.

4.2 Semantic Correctness

In this section, we prove that the byte code compiler preserves meaning.

Unfortunately, there is a sort of semantic mismatch between Scheme and the BBC language. The official semantics for Scheme allows a procedure to return several values to its caller, as would be needed to model the Common Lisp `(values ...)` construct or the **T** `(return ...)` form. However, Scheme has no construct that allows a programmer to construct a procedure that would return more than one value. Assuming that a program is started with an initial store that hides no "multiple value returners," then an implementation may assume that there will never be any in the course of execution. Moreover, the Scheme standard procedures as defined in IEEE report [2], which are the natural procedures to supply as denizens of the initial store, do not involve anything of that kind. So many implementations of Scheme, among them Vlisp, do not support multiple return values.

Still, it is far from trivial to formalize reasoning that justifies an implementation in assuming it need implement only "single-valued" objects, and make no provision for multiple value returns. In this section, we will attempt to do so.

In essence our approach is to introduce a new, "smaller" semantics for Scheme. In this semantics, it is clear that there is no mechanism for multiple return values. The semantics is "smaller" in the sense that domains, such as the domain of expressed values, are subsets of the corresponding domains in the official semantics. They contain only the objects needed for single-value returning procedures. Although we will formally express the semantic functions of the alternate semantics in terms of the same domains that the standard semantics uses, it will be clear that the alternate semantics really only depends on the smaller domains of "single-valued" objects. We use the old domains in the definitions in order to facilitate a comparison between the two semantic theories.

With this alternate semantics in place, there are two separate facts that must be proved to justify the compiler algorithm.

1. The alternate semantics is faithful to the standard semantics, at least in the intended case in which the initial givens harbor no multiple value returns:

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

   when κ and σ, the halt continuation and the initial store respectively, are unproblematic, single-valued objects in a sense to be defined below.

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

Table 6: Pure Procedure Objects

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

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

yield the same computational answer when applied to suitable initial values.

We will also (for convenience) introduce a new explicitly defined domain of *pure procedure objects*. Unlike the procedure objects in $\mathtt{F}$ (which equals $\mathtt{L} \times (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C})$), those in $\mathtt{F}_p$ contain only the function-like part (namely $\mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$), without the location that serves as a tag (see Table 6). The location tags are used to decide whether two procedure objects are the same in the sense of the Scheme standard procedure `eqv?`. So $\mathtt{F} = \mathtt{L} \times \mathtt{F}_p$. It will also be convenient to define a few auxiliary functions:

Definition 11 (Auxiliaries)

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

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

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

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

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

Lemma 12 1. $one\_arg(multiple\,\psi) = \psi$*;*

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

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

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

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

Recall that we assume $\mathtt{E}$ to be indecomposable, and $\mathtt{F} = \mathtt{L} \times \mathtt{F}_p$. We define Ω to be the disjoint sum (in any order) of a set C_Ω of domains, where C_Ω is the least set containing all of the Scheme denotational domains (including E_Σ) and such that

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

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

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

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

(a) if $\omega = \bot_\Omega$, then ω

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

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

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

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

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

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

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

Note that all of the real work of α is done by the truncation in the last case. It is important to note that in all cases if $\omega : D$, then $\alpha f \omega : D$; as αf is strict we see that for all $f : \Omega \to \Omega$, αf respects types.

Definition 13 (Single-valued approximation) $\mathbf{sva} : \Omega \to \Omega$ *is the least fixed point of the above* α.

For $D \in C_\Omega$ and $d \in D$, we will abuse notation by writing $\mathbf{sva}\ d$ for $\mathbf{sva}^D\ d$, i.e., for $(f(d \text{ in } \Omega)|D)$.

Lemma 14 $\mathbf{sva}$ *is idempotent.*

Proof. If we let $f_0 = \bot_{\Omega\to\Omega}$, and $f_{n+1} = \alpha f_n$, then $\mathbf{sva}$ is the supremum of the f_n for $n \in N$. Define a strict $g_0 : \Omega \to \Omega$ by letting $g_0\,\omega = \bot_D\ in\ \Omega$ whenever $\omega : D$ (so each $g_0^D = \bot_{D\to D}$). Then g_0 is least type-respecting element of $\Omega \to \Omega$.

Since f_1 respects types, $f_0 \sqsubseteq g_0 \sqsubseteq f_1$. Also let $g_{n+1} = \alpha g_n$. As g_1 respects types, $g_0 \sqsubseteq g_1$. Hence each $g_n \sqsubseteq g_{n+1}$, and the supremum of the g_n's is also $\mathbf{sva}$.

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

(i) h respects types;

(ii) each h^D is strict; and

(iii) h is idempotent.

It is easy to see that the supremum of idempotent elements of $\Omega \to \Omega$ is idempotent, and that $P(g_0)$ holds. It suffices to establish that α preserves the property P.

Take an arbitrary f such that $P(f)$ and let h be αf. We already know that (i) holds.

For (ii) consider cases on D. The case of an indecomposable D other than $\mathtt{E}$ is trivial. To compute $h^{\mathtt{E}} \bot_{\mathtt{E}}$ we apply case (c) of the definition of αf with $\omega = \bot_{\mathtt{E}}$ in Ω. Projecting this ω to $\mathtt{E}$ yields $\bot_{\mathtt{E}}$; applying ψ^{-1} to that yields $\bot_{\mathtt{E}_\Sigma}$. We now use the assumption that f is strict on $\mathtt{E}_\Sigma$, and end up with $\bot_{\mathtt{E}}$. For D decomposable other than $\mathtt{F}_p$, the definition of lifting shows that a strict functional is produced if all the arguments' functionals are strict, which is the case here. The case of $\bot_{\mathtt{F}_p}$ is easy from the explicit λ term used in this case of the definition of α. This establishes that (ii) of P holds for h.

For idempotence of h, pick an arbitrary $\omega \in \Omega$, and consider the various cases in the definition of $h\omega$. For (a) and (b) there is no problem. For (c) we need to use the fact that f respects types. Each of the subcases of (d) is easy.

For (e), let $\omega : \mathtt{F}_p$; we must show that $(\alpha\, f)(\alpha\, f\, \omega) = (\alpha\, f\, \omega)$. By (e), the former (projected into $\mathtt{F}_p$ for convenience) equals:

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

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

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

Now if $0 < \#\epsilon_0^*$, the idempotency of f and therefore also $f^{\mathtt{E}}$ suffices, as the equation reduces to

$$\langle f^{\mathtt{E}}(f^{\mathtt{E}}(\epsilon_0^*\, 0))\rangle = \mathit{trunc}(f^{\mathtt{E}} \circ \epsilon_0^*)$$

On the other hand if $\epsilon_0{}^*$ is either $\bot_{\mathtt{E}^*}$ or $\langle\rangle$, then we have:

$$\begin{aligned} trunc(f^{\mathtt{E}} \circ (trunc \bot_{\mathtt{E}^*})) &= trunc(f^{\mathtt{E}} \circ \langle \bot_{\mathtt{E}} \rangle) \\ &= trunc\langle \bot_{\mathtt{E}} \rangle \\ &= \langle \bot_{\mathtt{E}} \rangle \end{aligned}$$

using the definitions of α and of *truncate*, and the strictness of $f^{\mathtt{E}}$ (clause (ii) of P). Moreover,

$$\begin{aligned} trunc(f^{\mathtt{E}} \circ \bot_{\mathtt{E}^*}) &= trunc \bot_{\mathtt{E}^*} \\ &= \langle \bot_{\mathtt{E}} \rangle \end{aligned}$$

QED.

4.2.1 Single-valued Scheme Semantics

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

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

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

which is single-valued.

Theorem 15 (Faithfulness of Alternative Semantics)

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

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

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

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

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

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

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

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

QED

Corollary 16 *For all values of e, ρ, κ, and σ:*

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

Proof. Using Equation 1, the idempotence of **sva** on continuations, and then Equation 1 again, from right to left, we have:

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

The other assertions are similar, using also the idempotence of **sva** on stores. QED

Corollary 17 *For all values of e, ρ, κ, and σ, if $\kappa \sim \kappa'$ and $\sigma \sim \sigma'$, then*

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

Proof of Corollary 17. Simply apply Corollary 16 to each of κ, σ and κ', σ', and use the facts that $(\mathbf{sva}\,\kappa) = (\mathbf{sva}\,\kappa')$ and $(\mathbf{sva}\,\sigma) = (\mathbf{sva}\,\sigma')$. QED

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

Theorem 18 (Truncate Continuations)
$(\mathbf{sva}\,\mathcal{E})[\![\mathrm{E}]\!]\rho\,\kappa = (\mathbf{sva}\,\mathcal{E})[\![\mathrm{E}]\!]\rho\,(\lambda\epsilon^*\,.\,\kappa\,(\mathit{truncate}\,\epsilon^*))$

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

To prove

$$(\mathbf{sva}\,\mathcal{E})[\![\langle\mathrm{E}_0\rangle{}^\frown\mathrm{E}^*]\!]\rho\,\kappa = (\mathbf{sva}\,\mathcal{E})[\![\langle\mathrm{E}_0\rangle{}^\frown\mathrm{E}^*]\!]\rho\,(\lambda\epsilon^*\,.\,\kappa\,(\mathit{truncate}\,\epsilon^*))$$

we must show that

$$\begin{array}{l}(\mathbf{sva}\,\mathcal{E}^*)(\mathit{permute}(\langle\mathrm{E}_0\rangle{}^\frown\mathrm{E}^*)) \\ \qquad \rho \\ \qquad (\lambda\epsilon^*\,.\,((\lambda\epsilon^*\,.\,\mathit{applicate}\,(\epsilon^*\,0)\,(\epsilon^*\dagger 1)\,\kappa) \\ \qquad\qquad (\mathit{unpermute}\,\epsilon^*)))\end{array}$$

equals

$$\begin{array}{l}(\mathbf{sva}\,\mathcal{E}^*)(\mathit{permute}(\langle\mathrm{E}_0\rangle{}^\frown\mathrm{E}^*)) \\ \qquad \rho \\ \qquad (\lambda\epsilon^*\,.\,((\lambda\epsilon^*\,.\,\mathit{applicate}\,(\epsilon^*\,0)\,(\epsilon^*\dagger 1)\,(\lambda\epsilon^*\,.\,\kappa\,(\mathit{truncate}\,\epsilon^*))) \\ \qquad\qquad (\mathit{unpermute}\,\epsilon^*))).\end{array}$$

Pushing **sva**s through the left hand side, we obtain:

$$\begin{array}{l}\mathcal{E}^*(\mathit{permute}(\langle \mathrm{E}_0\rangle^\frown \mathrm{E}^*))\\ \quad \rho\\ \quad (\lambda\epsilon^* \,.\, ((\lambda\epsilon^* \,.\, \mathbf{sva}(\mathit{applicate}\ (\mathbf{sva}(\epsilon^*\,0))\ (\mathbf{sva}(\epsilon^* \dagger 1))\\ \qquad\qquad\qquad\qquad\qquad\qquad (\lambda\epsilon^* \,.\, (\mathbf{sva}\kappa)(\mathit{truncate}\,\epsilon^*))))\\ \qquad\quad (\mathbf{sva}(\mathit{unpermute}\ (\mathbf{sva}\epsilon^*)))))\end{array}$$

On the right hand side, we obtain:

$$\begin{array}{l}\mathcal{E}^*(\mathit{permute}(\langle \mathrm{E}_0\rangle^\frown \mathrm{E}^*))\\ \quad \rho\\ \quad (\lambda\epsilon^* \,.\, ((\lambda\epsilon^* \,.\, \mathbf{sva}(\mathit{applicate}\ (\mathbf{sva}(\epsilon^*\,0))\ (\mathbf{sva}(\epsilon^* \dagger 1))\\ \qquad\qquad\qquad\qquad\qquad\qquad (\lambda\epsilon^* \,.\, (\lambda\epsilon^* \,.\, \mathbf{sva}(\kappa\ (\mathbf{sva}(\mathit{truncate}\,\epsilon^*))))\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathit{truncate}\,\epsilon^*))))\\ \qquad\quad (\mathbf{sva}(\mathit{unpermute}\ (\mathbf{sva}\epsilon^*))))).\end{array}$$

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

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

Using the definition of **sva**, we have:

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

On the other hand,

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

QED.

Theorem 19 ("Single" eliminable)
$(\mathbf{sva}\,\mathcal{E})[\![\mathrm{E}]\!]\rho\ (\mathit{single}\,\psi) = (\mathbf{sva}\,\mathcal{E})[\![\mathrm{E}]\!]\rho\ (\lambda\epsilon^* \,.\, \psi\ (\epsilon^*\,0))$

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

To prove

$$(\mathbf{sva}\,\mathcal{E})[\![\langle \mathrm{E}_0\rangle^\frown \mathrm{E}^*]\!]\rho\,(\mathit{single}\,\psi) = (\mathbf{sva}\,\mathcal{E})[\![\langle \mathrm{E}_0\rangle^\frown \mathrm{E}^*]\!]\rho\,(\lambda\epsilon^*\,.\,\psi\,(\epsilon^*\,0))$$

we must show that

$$\begin{array}{l}(\mathbf{sva}\,\mathcal{E}^*)(\mathit{permute}(\langle \mathrm{E}_0\rangle^\frown \mathrm{E}^*)) \\ \qquad \rho \\ \qquad (\lambda\epsilon^*\,.\,((\lambda\epsilon^*\,.\,\mathit{applicate}\,(\epsilon^*\,0)\,(\epsilon^*\dagger 1)\,(\mathit{single}\,\psi)) \\ \qquad\qquad (\mathit{unpermute}\,\epsilon^*)))\end{array}$$

equals

$$\begin{array}{l}(\mathbf{sva}\,\mathcal{E}^*)(\mathit{permute}(\langle \mathrm{E}_0\rangle^\frown \mathrm{E}^*)) \\ \qquad \rho \\ \qquad (\lambda\epsilon^*\,.\,((\lambda\epsilon^*\,.\,\mathit{applicate}\,(\epsilon^*\,0)\,(\epsilon^*\dagger 1)\,(\lambda\epsilon^*\,.\,\psi\,(\epsilon^*\,0))) \\ \qquad\qquad (\mathit{unpermute}\,\epsilon^*))).\end{array}$$

Pushing **sva**s through the left hand side, we obtain:

$$\begin{array}{l}\mathcal{E}^*(\mathit{permute}(\langle \mathrm{E}_0\rangle^\frown \mathrm{E}^*)) \\ \quad \rho \\ \quad (\lambda\epsilon^*\,.\,((\lambda\epsilon^*\,.\,\mathbf{sva}(\mathit{applicate}\,(\mathbf{sva}(\epsilon^*\,0))\,(\mathbf{sva}(\epsilon^*\dagger 1)) \\ \qquad\qquad\qquad\qquad (\lambda\epsilon^*\,.\,(\mathbf{sva}(\mathit{single}\,\psi))(\mathit{truncate}\,\epsilon^*))))) \\ \qquad (\mathbf{sva}(\mathit{unpermute}\,(\mathbf{sva}\epsilon^*)))))\end{array}$$

On the right hand side, we obtain:

$$\begin{array}{l}\mathcal{E}^*(\mathit{permute}(\langle \mathrm{E}_0\rangle^\frown \mathrm{E}^*)) \\ \quad \rho \\ \quad (\lambda\epsilon^*\,.\,((\lambda\epsilon^*\,.\,\mathbf{sva}(\mathit{applicate}\,(\mathbf{sva}(\epsilon^*\,0))\,(\mathbf{sva}(\epsilon^*\dagger 1)) \\ \qquad\qquad\qquad\qquad (\lambda\epsilon^*\,.\,(\lambda\epsilon^*\,.\,\mathbf{sva}(\psi\,(\mathbf{sva}(\epsilon^*\,0))))(\mathit{truncate}\,\epsilon^*)))) \\ \qquad (\mathbf{sva}(\mathit{unpermute}\,(\mathbf{sva}\epsilon^*))))).\end{array}$$

For these two expressions to be equal, it certainly suffices that for all ϵ, ϵ^* and ψ,

$$\lambda\epsilon^*.(\mathbf{sva}(\mathit{single}\psi))(\mathit{truncate}\epsilon^*) = \lambda\epsilon^*.(\lambda\epsilon^*.\mathbf{sva}(\psi\,(\mathbf{sva}(\epsilon^*0))))(\mathit{truncate}\epsilon^*)$$

By β-reduction, the right hand side equals:

$$(\lambda\epsilon^*\,.\,\mathbf{sva}(\psi\,(\mathbf{sva}((\mathit{truncate}\,\epsilon^*)\,0))))$$

Using the definitions of *single* and **sva**, and then β-reduction, we have, for the left hand side (using w to abbreviate (**sva***wrong*“”)):

$$\begin{aligned}
&(\lambda\epsilon^*\,.\,(\mathbf{sva}(\mathit{single}\;\psi))(\mathit{truncate}\;\epsilon^*))\\
&\quad = \quad (\lambda\epsilon^*\,.\,(\mathbf{sva}(\lambda\,\epsilon^*\,.\,\#\epsilon^* = 1 \rightarrow \psi(\epsilon^*\;0), \mathit{wrong}\text{“”}))(\mathit{truncate}\;\epsilon^*))\\
&\quad = \quad (\lambda\epsilon^*\,.\,(\lambda\,\epsilon^*\,.\,\#\mathbf{sva}\epsilon^* = 1 \rightarrow \mathbf{sva}(\psi((\mathbf{sva}\epsilon^*)\;0)), w)(\mathit{truncate}\;\epsilon^*))\\
&\quad = \quad (\lambda\epsilon^*\,.\,\#(\mathbf{sva}(\mathit{truncate}\;\epsilon^*)) = 1 \rightarrow \mathbf{sva}(\psi((\mathbf{sva}(\mathit{truncate}\;\epsilon^*))\;0)), w)
\end{aligned}$$

Since *truncate* always returns a sequence of length 1, and **sva** does not affect length, the body of the latter equals $\mathbf{sva}(\psi((\mathbf{sva}(\mathit{truncate}\,\epsilon^*))\,0))$. The proof is complete, because $((\mathbf{sva}(\mathit{truncate}\;\epsilon^*))\;0) = \mathbf{sva}((\mathit{truncate}\;\epsilon^*)\;0)$. QED

4.2.2 Compiler Correctness for Single-valued Semantics

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

Definition 20 (Computational equivalence $\equiv$)
A Scheme program e and a byte code program b are computationally equivalent if, for every environment ρ and expression continuation κ,

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

Definition 21 (Environment Composition)
Suppose ρ is an environment,

$$\rho_C : \mathit{Ide} \rightarrow (\mathtt{N} \times \mathtt{N} \cup \{\mathit{not\_lexical}\})$$

is a “compile-time environment,” and

$$\rho_R : \mathtt{N} \rightarrow \mathtt{N} \rightarrow \mathtt{L}$$

is a “run-time environment.”
Then their environment composition $\rho\triangleleft^{\rho_R}_{\rho_C}$ *is the environment:*

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

Lemma 22 $\rho\triangleleft^{\rho_R}_{\mathrm{MT}_C} = \rho$.

Proof. For all i, $\text{MT}_C\ i = \mathit{not\_lexical}$. Hence, $\rho\triangleleft^{\rho_R}_{\text{MT}_C}\ i = \rho\, i$, and, by extensionality, $\rho\triangleleft^{\rho_R}_{\text{MT}_C} = \rho$. QED

We state the main theorem next, and we will give its (short) proof using Lemma 25, which is due to Will Clinger [1]. The bulk of the (long) proof of Clinger's lemma, for the Vlisp compiler, is given below.

Theorem 23 (Compiler correctness)
For any Scheme expression e and environment ρ,

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

Proof. The theorem is a direct consequence of Clinger's lemma (Lemma 25 below), Clause 1. Since

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

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

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

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

$$\begin{aligned}\lambda\epsilon\ .\ \mathit{return}\ \epsilon\langle\rangle\rho_R(\mathit{one\_arg}\,\kappa) &= \lambda\epsilon\ .\ (\mathit{one\_arg}\,\kappa)\epsilon\\ &= (\mathit{one\_arg}\,\kappa),\end{aligned}$$

and it follows that

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

By Lemmas 12 and 22, the left hand side equals:

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

By Theorem 18, the latter equals $(\mathbf{sva}\mathcal{E})[\![e]\!]\ \rho\,\kappa$. QED.

Lemma 24 (Compiled expressions set value register)

1. *Let $\pi = (\mathbf{sva}\mathcal{B})[\![C(e, \rho_C, n, i, b)]\!]\rho$. For all ϵ, ϵ',*

$$\pi\epsilon = \pi\epsilon'.$$

2. *Let* $\pi = (\mathbf{sva}\mathcal{Y})[\![C(e, \rho_C, n, i, y)]\!]\rho\pi_0$. *For all* ϵ, ϵ',

$$\pi\epsilon = \pi\epsilon'.$$

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

$$\pi\epsilon = \pi\epsilon'.$$

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

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

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

so that the result follows by Clause 1.

Lemma 25 (Clinger's Lemma)

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

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

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

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

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

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

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

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

   Suppose that the after code sets the value register before it reads it, in the sense that, for all values ϵ',

$$\pi_1\,\epsilon = \pi_1\,\epsilon'.$$

Then

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

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

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

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

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

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

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

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

Applying κ,

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

Hence, the left hand side equals:

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

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

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

Lemma 24 shows (for all ϵ_0):

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

Using this, followed by the assumption that Lemma 24 establishes, and then the definition of *push*, and the semantics of `push`, we have:

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

We will apply Clause 1 with

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

and thus with

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

Hence,

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

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

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

as the latter is the code:

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

5 The Tabulator Algorithm

The purpose of the tabulator is to translate programs from the basic byte code BBC into the tabular byte code TBC. In the former, the instructions:

`closure`, `literal`, `global`, and `set-global!`

contain their argument literally within themselves. In the case of `closure`, this is a full, syntactic template; in the case of `literal`, it is a constant, for instance a large nested list; in the remaining cases it is an identifier. When the code is actually running on the virtual machine, however, we want there to be a single kind of fixed width item in the instruction stream.

We achieve this by replacing the item with a small (one-byte) integer, which is used as index into a table making up the template itself. Indeed, the instruction stream is itself contained in this table as its zeroth item. For this reason, and because we can not actually store the instruction stream there until we have linearized the code, the tabulator ensures that the zeroth item in the table it generates is not used. We will say that these four instructions *use the template table.*

The main content of the tabulator algorithm is contained in Section 5.1, while supplementary procedures (also used in the linker algorithm) are gathered in Section 5.2.

5.1 Tabulate and Related Procedures

Given a BBC template $t = \langle \texttt{lap}\ c\ b \rangle$, the tabulator treats c as the name of the template and b as the code from which literals are to be gleaned.

The algorithm uses a data structure called an *inverse table.* An inverse table is a bijection—implemented by an association list of TBC literals and integer values—whose range is a finite initial segment of the natural numbers; it represents a template table.

Inverse tables are extended, and their values are accessed, by a function `probe`, called with a TBC literal ℓ and an inverse table f. It returns a pair (i.e. cons) of values, the first of which is an index i and the second of which is an inverse table f'. If f is defined for ℓ, then $f' = f$ and $i = f(\ell)$. Otherwise, i is the least integer not in the range of f, and $f' = f \oplus \{\ell \mapsto i\}$. Thus, the new inverse table differs from the given one at most in its behavior for ℓ.

The algorithm first initializes an inverse table with the name of the given template as its entry at position 1, and with a dummy value that cannot

possibly occur as a literal in the code at position 0. For this purpose we use the given template itself, which is surely too large to occur inside itself. The main recursive descent algorithm is then called with the byte code b and the initialized inverse table. Finally the resulting code and inverse table are returned; these are destructured, and the inverse table is converted into a list of its literals in the proper order. The big nasty dummy value is relaced with a small, convenient dummy value, namely 0. Finally, the TBC template $\langle$`template` b' $e\rangle$ is returned.

```
(define (tabulate-top bbc-template)
  (let ((name (cadr bbc-template))
        (b    (cddr bbc-template)))
    (let ((inv-table
           (probe-fun
            (probe
             `(constant ,name)
             (probe-fun
              (probe
               ;; dummy for code seq slot
               ;;
               `(constant ,bbc-template)
               null-inv-table))))))
      (let ((code+inv-table (tabulate b inv-table)))
        `(template
          ,(car code+inv-table)          ; code
          ((constant 0)
           .                             ; put safe dummy
           ,(cdr                         ; in first spot
             (invert-inv-table           ; in table
              (cdr code+inv-table)))))))))
```

The main recursive procedure `tabulate` takes a piece of BBC code w and a partially constructed inverse table f. It returns a pair (i.e. cons) consisting of the corresponding TBC code $\bar{w}$, together with an extended table f'. The main correctness condition is that the semantics of w should be equal to that of $\bar{w}$ (with respect to f'). The algorithm is, in case w is non-empty, first to tabulate all but the first instruction, obtaining code together with an extended inverse table. Then the first instruction is tabulated with respect to the extended inverse table. If w is empty, then no code is returned, represented by `()`, together with the given inverse table.

```
(define (tabulate bbc inv-table)
```

```
  (if (null? bbc)
      (cons '() inv-table)
      (let ((instr (car bbc))
            (code+inv-table (tabulate (cdr bbc) inv-table)))
        (let ((after-code (car code+inv-table))
              (inv-table  (cdr code+inv-table)))
          (let ((instr+inv-table
                 (tabulate-instruction instr inv-table)))
            (cons
             (cons (car instr+inv-table) after-code)
             (cdr instr+inv-table)))))))
```

The main content of the tabulator is to ensure that each individual instruction is properly treated. If the instruction uses the template table, then we must probe the table to get an index and a new table; the index is then inserted in place of the literal item. On the other hand, for closure, `make-cont`, and conditional instructions, a recursive call is needed to tabulate the nested code.

```
(define (tabulate-instruction instr inv-table)
  (case (car instr)

    ;; Must recursively tabulate
    ;; the embedded template
    ((closure)
     (tabulate-closure-instruction instr inv-table))

    ;; replace literal
    ;; by index
    ((literal global set-global!)
     (tabulate-template-instruction instr inv-table))

    ;; Must recursively
    ;; tabulate both branches
    ((unless-false)
     (tabulate-conditional (cadr instr) (caddr instr) inv-table))

    ;; Must recursively
    ;; tabulate return code
    ((make-cont)
     (tabulate-make-cont (cadr instr) (caddr instr) inv-table))

    ;; No change needed
```

```
    (else
     (cons instr inv-table))))
```

For a closure instruction, we first tabulate the embedded template, and then probe with the converted value.

```
(define (tabulate-closure-instruction instr inv-table)
  (let ((index+inv-table
         (probe (tabulate-top (cadr instr)) inv-table)))
    `((closure ,(probe-val index+inv-table))
      .
      ,(probe-fun index+inv-table))))
```

For the other instructions using the template table, we simply probe with the instruction literal.

```
(define (tabulate-template-instruction instr inv-table)
  (let ((index+inv-table
         (probe
          (instruction-literal instr)
          inv-table)))
    `((,(car instr) ,(probe-val index+inv-table))
      .
      ,(probe-fun index+inv-table))))
```

For conditionals, we tabulate the nested code. The consequent and alternative must be done in some fixed order, so that the table resulting from tabulating the first may be supplied when the second is tabulated.

```
(define (tabulate-conditional conseq0 alt0 inv-table)
  (let ((alt+inv-table
         (tabulate alt0 inv-table)))
    (let ((conseq+inv-table
           (tabulate conseq0 (cdr alt+inv-table))))
      `((unless-false ,(car conseq+inv-table) ,(car alt+inv-table))
        .
        ,(cdr conseq+inv-table)))))
```

For a `make-cont` instruction we simply tabulate the nested code.

```
(define (tabulate-make-cont nested0 n inv-table)
  (let ((nested+inv-table
         (tabulate nested0 inv-table)))
    `((make-cont ,(car nested+inv-table) ,n)
      .
      ,(cdr nested+inv-table))))
```

The `instruction-literal` procedure "wraps up" what literally occurs in a byte code instruction with a word tagging whether it is a global variable or a constant. Templates extracted from the `closure` instruction need no tag, as the syntax of a template identifies it as such. We shall use the phrase "the instruction literal of an instruction" to refer to the value returned by this procedure. We will say that an instruction literal occurs in a source byte code expression without template a under the same recursive condition we used for occurrence of an index n in a byte expression with template.

```
(define (instruction-literal instr)
  (let ((literal-content (cadr instr)))
    (case (car instr)
      ((literal)     `(constant        ,literal-content))
      ((global)      `(global-variable ,literal-content))
      ((set-global!) `(global-variable ,literal-content))
      ((closure)     literal-content)
      (else          '#f))))
```

5.2 Probe and Inverse Tables

If f is a partial function with range included in $\mathtt{N}$, then by $\sup f$ we mean the maximum value (if any) that f takes on. That is, $\sup f = n$ if and only if there is an argument a such that $fa = n$, and for all b, if fb is well-defined, then $fb \leq n$.

$probe : (d \times (d \to \mathtt{N})) \to (\mathtt{N} \times (d \to \mathtt{N}))$
$probe(d, f) = \langle j, g \rangle$ where
$g(d) = j$ and
if $f(d)\downarrow$ then $g = f$,
otherwise $g = f + \{d \mapsto 1 + \sup f\}$

One consequence of the definition of *probe* is that it does not alter the behavior of the function on arguments for which it is already defined:

$$f(d)\downarrow \ \wedge \ g = probe\_fun(probe(d', f)) \quad \Rightarrow \quad f(d) = g(d).$$

When we say of two (partial) functions f and g that g *extends* f, we mean that $g(x) = f(x)$ whenever the latter is defined. So g extends f.

A second fact about *probe* is that it preserves the property of being inverse to a finite sequence:

$$finite\_seq(f^{-1}) \ \Rightarrow \ finite\_seq((probe\_fun(probe(d', f)))^{-1}).$$

Thus, if $f^{-1} \in e$, then so is the inverse of the function resulting from a *probe*.

We will introduce aliases for the pair-destructuring operators *first* and *second*:

$probe\_val : \mathtt{N} \times (d \to \mathtt{N}) \to \mathtt{N}$
$probe\_val(\langle j, g \rangle) = j$

$probe\_fun : \mathtt{N} \times (d \to \mathtt{N}) \to (d \to \mathtt{N})$
$probe\_fun(\langle j, g \rangle) = g$

5.2.1 Implementation of Probe

To implement inverse tables, we use Scheme lists containing cons cells. We will call a list an `inv-table` if:

- The `car` of each such pair represents a template literal, and the `cdr` is a number;
- No two pairs have the same `car`;
- The numbers occurring in the `cdr`s are in strictly decreasing order and coincide with an initial segment of the natural numbers.

Hence, each `inv-table` represents an inverse table, when the operation of applying an inverse table to a literal is implemented by the procedure `seek-probe-value` below. Conversely, every inverse table, being finite, may be represented by an `inv-table` (in a large enough heap), simply by providing a cons cell for each $\langle literal, value \rangle$ pair in the function, and building a list containing them in decreasing order of the *values*.

To implement *probe* relative to this representation, we use the following procedures:

```
(define (probe o inv-tab)
  (let ((v (seek-probe-value inv-tab o)))
    (if (fail? v)
        (let ((new-inv-tab (expand-inv-table inv-tab o)))
          (cons (seek-probe-value new-inv-tab o)
                new-inv-tab))
        (cons v inv-tab))))

(define (expand-inv-table inv-tab o)
  (if (eq? inv-tab null-inv-table)
```

```
      (list (cons o 0))                          ; single entry
      (let ((new-val  (+ (cdar inv-tab) 1)))
        (cons (cons o new-val)                   ; new entry
              inv-tab))))                        ; before old table

(define (seek-probe-value inv-tab o)
  (let iter ((inv-tab inv-tab))
    (cond ((null? inv-tab) (fail))               ; not there
          ((equal? o (caar inv-tab))             ; first entry
           (cdar inv-tab))
          (else (iter (cdr inv-tab))))))         ; seek further

(define null-inv-table '())
```

We also introduce aliases for `car` and `cdr`:

```
(define probe-val car)
(define probe-fun cdr)
```

The next applicative Scheme procedure allows us to generate a representation of a template table from an `inv-table` table. The operation of applying the resulting table e to an integer argument n is given by the Scheme form `(nth` e n`)`.

```
(define (invert-inv-table inv-tab)
  ;;
  ;; convert from decreasing order to increasing order
  (reverse
   (map
    ;; extract just the template literals
    car
    inv-tab)))
```

Two final auxiliaries are:

```
(define (fail) 'fail)
(define (fail? value) (eq? value 'fail))
```

6 Correctness of the Tabulator Algorithm

In this section, we will use a bar to distinguish variables over the targets of a translation, which belong to the tabular byte code TBC, such as $\bar{z}$, from variables over source expressions, which belong to the basic byte code BBC, such as z.

We will also write:

- $T(t)$, when t is a BBC template, to refer to the result $\bar{t}$ of calling `tabulate-template` on t;
- $T(w, f)$, when w is a BBC (open or closed) instruction sequence, to refer to the result $\langle \bar{w}, f' \rangle$ of calling `tabulate` on w and f;
- $T(m, f)$, when m is a BBC instruction, to refer to the result $\langle \bar{m}, f' \rangle$ of calling `tabulate` on m and f.

6.1 Syntactic Correctness

Theorem 26 (Tabulator Syntactic Correctness)

1. *If t is a* BBC *template and $\bar{t} = T(t)$, then $\bar{t}$ is a* TBC *template.*
2. *Suppose b is a* BBC *closed instruction list and f is an inverse table. If $\langle \bar{b}, f' \rangle = T(b, f)$, then $\bar{b}$ is a* TBC *closed instruction list and f' is an inverse table.*
3. *Suppose y is a* BBC *open instruction list and f is an inverse table. If $\langle \bar{y}, f' \rangle = T(y, f)$, then $\bar{y}$ is a* TBC *open instruction list and f' is an inverse table.*
4. *Suppose z is a* BBC *instruction and f is an inverse table. If $\langle \bar{z}, f' \rangle = T(m, f)$, then $\bar{z}$ is a* TBC *instruction and f' is an inverse table.*

Proof. The proof is by simultaneous structural induction on the BBC syntax.

1. t is $\langle \texttt{lap}\ c\ b \rangle$. Apply 2. to b and the inverse table

   $$f = \{[\texttt{(constant}\ t\texttt{)} \mapsto 0],\ [c \mapsto 1]\},$$

   obtaining $\langle \bar{b}, f' \rangle$. Assuming that `inv-table` represents f', the following code returns a template table e.

```
`((constant 0)
  .                              ; put safe dummy
  ,(cdr                          ; in first spot
    (invert-inv-table            ; in table
     inv-table)))
```

So `tabulate-template` returns the TBC template $\langle$`template` $\bar{b}\ e\rangle$.

2. Consider the different syntactic cases for b:

   (a) If b is $\langle\langle$`return`$\rangle\rangle$ or $\langle\langle$`call` $n\rangle\rangle$, then $T(b, f) = \langle b, f\rangle$, and b belongs to the TBC syntax.

   (b) Suppose b is $\langle\langle$`unless-false` $b_1\ b_2\rangle\rangle$. By the inductive hypothesis (clause 2), $T(b_2, f)$ is a syntactically correct pair $\langle\bar{b_2}, f_2\rangle$, as is $T(b_1, f_2) = \langle\bar{b_1}, f'\rangle$. Hence, $\langle\langle$`unless-false` $\bar{b_1}\ \bar{b_2}\rangle\rangle$ is a closed TBC instruction list, and f' is an inverse table.

   (c) Suppose b is $\langle\langle$`make-cont` $b_1\ n\rangle :: b_2\rangle$. By the inductive hypothesis (clause 2), $T(b_2, f)$ is a syntactically correct pair $\langle\bar{b_2}, f_2\rangle$, as is $T(b_1, f_2) = \langle\bar{b_1}, f'\rangle$. Hence, $\langle\langle$`make-cont` $\bar{b_1}\ n\rangle :: \bar{b_2}\rangle$ is a closed TBC instruction list, and f' is an inverse table.

   (d) Suppose b is $z :: b_1$. Using the inductive hypothesis (clause 2), $T(b_1, f)$ is a syntactically correct pair $\langle\bar{b_1}, f_1\rangle$. Similarly, using clause 4, $T(z, f_1)$ is a syntactically correct pair $\langle\bar{z}, f'\rangle$. Hence, $\langle\bar{z} :: \bar{b_1}, f'\rangle$ is syntactically correct.

3. Consider the different syntactic cases for y:

   (a) Suppose y is $\langle\langle$`make-cont` $y_1\ n\rangle :: b_2\rangle$. By the inductive hypothesis (clause 2), $T(b_2, f) = \langle\bar{b_2}, f_2\rangle$ is syntactically correct. By clause 3, $T(y_1, f_2) = \langle\bar{y_1}, f'\rangle$ is a syntactically correct pair. Hence, $\langle\langle$`make-cont` $\bar{y_1}\ n\rangle :: \bar{b_2}\rangle$ is a closed TBC instruction list, and f' is an inverse table.

   (b) Suppose y is $\langle\langle$`make-cont` $\langle\rangle\ n\rangle :: b_2\rangle$. As before, $T(b_2, f) = \langle\bar{b_2}, f_2\rangle$ is syntactically correct. Examining the code, $T(\langle\rangle, f_2) = \langle\langle\rangle, f_2\rangle$. Hence, $T(y, f)$ is the syntactically correct

   $$\langle\langle\langle\text{`make-cont`}\ \langle\rangle\ n\rangle :: \bar{b_2}\rangle, f_2\rangle.$$

   (c) Suppose y is $z :: y_1$. By the inductive hypothesis (clause 3), $T(y_1, f) = \langle\bar{y_1}, f_1\rangle$ is a syntactically correct pair. Similarly, using

clause 4, $T(z, f_1) = \langle \bar{z}, f' \rangle$ is a syntactically correct pair. Hence, $\langle \bar{z} :: \bar{y_1}, f' \rangle$ is syntactically correct.

(d) Suppose y is $\langle z \rangle$. Using clause 4, $T(z, f) = \langle \bar{z}, f' \rangle$ is a syntactically correct pair. Hence, $\langle \bar{z}, f' \rangle$ is syntactically correct.

4. This is the only real case in the proof. Consider the different syntactic cases for z:

(a) Suppose z is $\langle$`unless-false` y_1 $y_2\rangle$. By the inductive hypothesis (clause 3), $T(y_2, f) = \langle \bar{y_2}, f_2 \rangle$ is a syntactically correct pair. $T(y_1, f_2) = \langle \bar{y_1}, f' \rangle$ is a syntactically correct pair.
Hence, $\langle$`unless-false` $\bar{y_1}$ $\bar{y_2}\rangle$ is a neutral instruction, and f' is an inverse table.

(b) Suppose z is $\langle$`literal` $c\rangle$. Then `instruction-literal` of z is $d =$ `(constant` c`)`. If $probe(d, f)$ is $\langle n, f' \rangle$, then T returns the syntactically correct value $\langle\langle$`literal` $n\rangle, f' \rangle$.

(c) Suppose z is $\langle$`closure` $t\rangle$. By the inductive hypothesis (clause 1), $T(t)$ is a syntactically correct TBC template $\bar{t}$. So $probe(\bar{t}, f) = \langle n, f' \rangle$, and the final value returned is $\langle\langle$`closure` $n\rangle, f' \rangle$.

(d) Suppose z is $\langle$`global` $i\rangle$ or $\langle$`set-global!` $i\rangle$. Then the procedure `instruction-literal`, applied to z, yields $d =$ `(global` i`)`. If $probe(d, f)$ is $\langle n, f' \rangle$, then T returns the syntactically correct value $\langle\langle$`global` $n\rangle, f' \rangle$ or $\langle\langle$`set-global!` $n\rangle, f' \rangle$.

(e) Otherwise, z is a TBC neutral instruction, and T returns $\langle z, f \rangle$.

Lemma 27 (Tabulator extends inverse tables.)

1. *If* $T(b, f) = \langle \bar{b}, f' \rangle$, *then* f' *extends* f;
2. *If* $T(y, f) = \langle \bar{y}, f' \rangle$, *then* f' *extends* f;
3. *If* $T(z, f) = \langle \bar{z}, f' \rangle$, *then* f' *extends* f.

Proof. The proof is by simultaneous induction on the BNF for b, y, and z.

1. Consider the syntactic cases for b:

(a) $b = \langle\langle$`return`$\rangle\rangle$ or $b = \langle\langle$`call` $n\rangle\rangle$: Then $T(b, f) = \langle b, f \rangle$, and f extends itself.

(b) Suppose b is $\langle\langle$`unless-false` $b_1\ b_2\rangle\rangle$. Let $T(b_2, f) = \langle\bar{b_2}, f_2\rangle$ and $T(b_1, f_2) = \langle\bar{b_2}, f'\rangle$. By the induction hypothesis (clause 1), f_2 extends f, and f' extends f_2. So f' extends f.

(c) If b is $\langle\langle$`make-cont` $b_1\ n\rangle :: b_2\rangle$, the argument is similar to the previous case.

(d) Suppose b is $\langle z :: b_1\rangle$. Then by IH (clause 1), if $T(b_1, f) = \langle\bar{b_1}, f_1\rangle$, then f_1 extends f. So applying IH (clause 3) to z and f_1, the resulting inverse table extends f.

2. The syntactic cases for y are very similar to those for b.

3. Consider the syntactic cases for z:

(a) $z = \langle$`unless-false` $y_1\ y_2\rangle$: Apply IH (clause 2) twice and the transitivity of *extends*.

(b) $z = \langle$`literal` $c\rangle$: f' is $probe(\langle$`constant` $c\rangle, f)$, which extends f.

(c) $z = \langle$`closure` $t\rangle$: f' is $probe(t, f)$, which extends f.

(d) $z = \langle$`global` $i\rangle$ or $z = \langle$`set-global!` $i\rangle$: likewise.

(e) Otherwise $f' = f$.

The flattener algorithm also requires that a special syntactic property holds of the output of the tabulator algorithm. To define that property, we first need various notions of *occurrence* in an instruction or instruction list.

Definition 28 (occurs in)
In the BBC *or* TBC*, an instruction* m occurs in *an instruction sequence* w *if:*

1. $w = \langle m\rangle$;

2. $w = m_1 :: w_1$ *and either* $m = m_1$ *or* m *occurs in* w_1;

3. $w = \langle\langle$`unless-false` $b_1\ b_2\rangle\rangle$ *or* $w = \langle$`make-cont` $b_1\ n\rangle :: b_2$, *and* m *occurs in* b_1 *or* b_2;

4. $w = \langle$`make-cont` $y_1\ n\rangle :: b$ *and* m *occurs in* y_1 *or* b.

If ℓ *is a* BBC *constant* c*, identifier* i*, or template* t*, we say that* ℓ occurs in *the* BBC *neutral instruction* z *if* z *is of the form* $\langle i\ \ell\rangle$*, where* i *is* `literal`, `global`, `set-global!`, *or* `closure`.

We will also say that n occurs as a template index in *the* TBC *neutral instruction* z *if* z *is of the form* $\langle i\ n\rangle$, *where* i *is* `literal`, `global`, `set-global!`, *or* `closure`.

If ℓ *is a* BBC *constant* c, *identifier* i, *or template* t, *we say that* ℓ occurs in *the* BBC *code sequence* w *if there is a* z *such that* z *occurs in* w *and* ℓ *occurs in* z.

Similarly, n occurs as a template index in *the* TBC *code sequence* w *if there is a* z *such that* z *occurs in* w *and* n *occurs as a template index in* z.

Lemma 29 t *does not occur as a constant* c *in any instruction* $\langle$`literal` $c\rangle$ *contained in* t.

Proof. If $z = \langle$`literal` $c\rangle$ is an instruction appearing in t, then the $\mathrm{rank}(c) < \mathrm{rank}(z) < \mathrm{rank}(t)$.

Lemma 30 *Let* ℓ *be a* BBC *literal with* $f(\ell) = n$.[3]

1. *Suppose that* ℓ *does not occur in* z; *and suppose that* $T(z, f) = \langle \bar{z}, f'\rangle$. *Then* n *does not occur as a template index in* $\bar{z}$.

2. *Suppose that* ℓ *does not occur in* w; *and suppose that* $T(w, f) = \langle \bar{w}, f'\rangle$. *Then* n *does not occur as a template index in* $\bar{w}$.

Proof. The proof is by simultaneous structural induction on the BNF for the BBC.

1. Consider the different syntactic cases for z:

   (a) $z = \langle$`unless-false` $y_1\ y_2\rangle$, and $\bar{z} = \langle$`unless-false` $\bar{y_1}\ \bar{y_2}\rangle$. By the induction hypothesis (clause 2), n does not occur as a template index in $\bar{y_1}$ or in $\bar{y_2}$; hence it does not occur in $\bar{z}$.

   (b) $z = \langle$`literal` $c\rangle$, and $c \neq \ell$. Let $\langle n', f'\rangle = \mathit{probe}(\langle$`constant` c$\rangle, f)$ where $f'($(`constant` c)$) = n'$. Since f, being an inverse table, is a bijection, $n' \neq n$. Moreover, $T(z, f) = \langle\langle$`literal` $n'\rangle, f'\rangle$, so n does not occur as a template index in the instruction.

   (c) $z = \langle$`closure` $t\rangle$. Let $\bar{t} = T(t)$, and let $\langle n', f'\rangle = \mathit{probe}(\bar{t}, f)$, where $f'(t) = n'$. Since f, being an inverse table, is a bijection, $n' \neq n$. Moreover, $T(z, f) = \langle\langle$`closure` $n'\rangle, f'\rangle$, so n does not occur as a template index in the instruction.

[3] So ℓ is a constant c or identifier i. If ℓ were a BBC template t, then f would not be defined for ℓ: BBC templates are disjoint from TBC templates.

(d) $z = \langle \texttt{global}\ i \rangle$ [or $z = \langle \texttt{set-global!}\ i \rangle$]. Let

$$\langle n', f' \rangle = \mathit{probe}(\texttt{(global-variable}\ i\texttt{)}, f),$$

where $f'(\texttt{(global-variable}\ i\texttt{)}) = n'$. Since f, being an inverse table, is a bijection, $n' \neq n$. Moreover, $T(z, f) = \langle \langle \texttt{global}\ n' \rangle, f' \rangle$ [or $\langle \texttt{set-global!}\ n' \rangle$], so n does not occur as a template index in the instruction.

(e) Otherwise $\bar{z} = z$, and no n' occurs as a template index in it.

2. Routine enumeration of syntactic cases.

Corollary 31 (Template table entry 0)
If $T(t) = \bar{t}$, then 0 does not occur as a template table entry in $\bar{t}$.

Proof. The procedure `tabulate-template` calls `tabulate` with an f such that $f(t) = 0$. So apply Lemmas 29 and 30.

Lemma 32 (Tabulator output defined for occurrences)

1. *If $T(b, f) = \langle \bar{b}, f' \rangle$, and if n occurs as a template index in b, then n is in the range of f';*

2. *If $T(y, f) = \langle \bar{y}, f' \rangle$, and if n occurs as a template index in y, then n is in the range of f';*

3. *If $T(z, f) = \langle \bar{z}, f' \rangle$, and if n occurs as a template index in z, then n is in the range of f'.*

As a consequence, if $T(t) = \langle \texttt{template}\ \bar{b}\ e \rangle$, and if n occurs as a template index in b, then e is defined at n.

Proof. By the usual syntactic induction:

1. Take cases on the syntactic form of b:

   (a) $b = \langle \langle \texttt{return} \rangle \rangle$ and $b = \langle \langle \texttt{call}\ n \rangle \rangle$ have no occurrences.

   (b) $b = \langle \langle \texttt{unless-false} \rangle\ b_1\ b_2 \rangle$ or $b = \langle \langle \texttt{make-cont}\ b_1\ n \rangle :: b_2 \rangle$: The occurrences in b are precisely those in b_1 or b_2. Moreover, by the IH clause 1 together with Lemma 27, the range of f' includes the occurrences of each separately.

(c) $b = z :: b_1$: The occurrences in b are precisely those in b_1 or z. Moreover, by the IH clauses 1 and 3, together with Lemma 27, the range of f' includes the occurrences of each separately.

2. Take cases on the syntactic form of y:

   (a) $y = \langle\langle \texttt{make-cont}\ y_1\ n\rangle :: b\rangle$: Similar to case 1(b), but using IH clauses 1 and 2.

   (b) $y = \langle\langle \texttt{make-cont}\ \langle\rangle\ n\rangle :: b\rangle$: The occurrences in y are precisely those in b. Hence it suffices to apply IH, Clause 1.

   (c) $y = z :: y_1$: Similar to case 1(c), but using IH clauses 2 and 3.

   (d) $y = \langle z\rangle$: Immediate from IH clause 2.

3. Take cases on the syntactic form of z:

   (a) $z = \langle\langle \texttt{unless-false}\rangle\ y_1\ y_2\rangle$: Similar to case 1(b), but using IH clause 2.

   (b) $z = \langle \texttt{literal}\ c\rangle$: Let $\mathit{probe}(\langle \texttt{constant}\ c\rangle, f) = \langle n, f'\rangle$, so that $T(z, f) = \langle\langle \texttt{literal}\ n\rangle, f'\rangle$. By the definition of *probe*, n is in the range of f'.

   (c) $z = \langle \texttt{closure}\ t\rangle$: Assume that $T(t) = \bar{t}$. Let $\mathit{probe}(t, f) = \langle n, f'\rangle$, so that $T(z, f) = \langle\langle \texttt{closure}\ n\rangle, f'\rangle$. By the definition of *probe*, n is in the range of f'.

   (d) $z = \langle \texttt{global}\ i\rangle$ or $z = \langle \texttt{set-global!}\ i\rangle$: Let

   $$\mathit{probe}(\langle \texttt{global-variable}\ i\rangle, f) = \langle n, f'\rangle,$$

   so that $T(z, f) = \langle\langle \texttt{global}\ n\rangle, f'\rangle$ or $\langle\langle \texttt{set-global!}\ n\rangle, f'\rangle$. By the definition of *probe*, n is in the range of f'.

   (e) Otherwise z has no occurrences.

Definition 33 *If z is a* TBC *neutral instruction, then z* respects the template table e *if:*

1. if $z = \langle \texttt{closure}\ n\rangle$, then $e(n)$ is a template $\langle \texttt{template}\ b_1\ e_1\rangle$, and moreover b_1 (recursively) respects the template table e_1;

2. if $z = \langle \texttt{literal}\ n\rangle$, then $e(n)$ is a literal of the form $\langle \texttt{constant}\ c\rangle$; and

3. if $z = \langle \texttt{global}\ n \rangle$ or $\langle \texttt{set-global!}\ n \rangle$, then $e(n)$ is a global of the form $\langle \texttt{global}\ i \rangle$.

A TBC *closed or open instruction sequence* w respects the template table e *if every* z *that occurs in* w *respects* e.

If t *is a template* $\langle \texttt{template}\ b\ e \rangle$, *we will say that* t *respects its table, or that* t *is* self-respecting, *if* b *respects the template table* e.

We will say that an instruction or sequence respects an inverse table f *if it respects the template table* f^{-1}.

The output of the tabulator always has this property. That is, if the result of the tabulator is a TBC template $\langle \texttt{template}\ b\ e \rangle$, then b respects the template table e. This provides the justification for proving the correctness of the flattener only for input templates with this property.

Lemma 34 *Suppose* w *respects the template table* $e = \langle d_0, \ldots d_n, \ldots d_j \rangle$; *suppose* n *does not occur as a template index in* w; *and suppose* d'_n *is not a template. Then* w *respects* $e' = \langle d_0, \ldots d'_n, \ldots d_j \rangle$.

Proof. For n, each of the four clauses of the definition of *respect* is vacuously fulfilled. Moreover, for $n' \neq n$, $e'(n') = e(n')$, so the clauses are fulfilled for e' assuming that they held for e.

Lemma 35 *If* w *respects the inverse table* f, *and* f_1 *is an inverse table that extends* f *(i.e.* f *is a subfunction of* f_1*), then* w *respects* f_1.

Proof. Since f_1 is an inverse table, f_1^{-1} is defined and extends f^{-1}. Hence, for each n occurring in w, since f^{-1} was defined for n, f_1^{-1} is also defined for n and has the same value.

Theorem 36 (Tabulate Elicits Respect)

1. *If* $t = \langle \texttt{lap}\ c\ b \rangle$ *is a* BBC *template, then* $T(t)$ *is self-respecting.*

2. *Suppose that* b *is a* BBC *closed instruction list and* f *is an inverse table. If* $T(b, f) = \langle \bar{b}, f' \rangle$, *then* $\bar{b}$ *respects* f'.

3. *Suppose that* y *is a* BBC *open instruction list and* f *is an inverse table. If* $T(y, f) = \langle \bar{y}, f' \rangle$, *then* $\bar{y}$ *respects* f'.

4. *Suppose that* z *is a* BBC *neutral instruction and* f *is an inverse table. If* $T(z, f) = \langle \bar{z}, f' \rangle$, *then* $\bar{z}$ *respects* f'.

Proof. The proof is by simultaneous induction on the syntactic structure of the BBC. For convenience we will say that a pair is self-respecting if its first element respects its second element.

1. Let f_0 be the empty (everywhere undefined) inverse table. Let f_1 be:

   $$\mathit{probe\_fun}(\mathit{probe}(\langle \texttt{constant}\ c\rangle, \mathit{probe\_fun}(\mathit{probe}(\langle \texttt{constant}\ b\rangle, f_0)))).$$

   Apply induction hypothesis (Clause 2) to b and f_1, concluding that $T(b, f_1) = \langle \bar{b}, f\rangle$ is self-respecting. By Corollary 31, 0 has no occurrence in $\bar{b}$. Hence, by Lemma 34, $\bar{b}$ still respects f^{-1} with $\langle \texttt{constant}\ 0\rangle$ in place of the zeroth entry of f^{-1}, which we may call e. Hence $\bar{t} = \langle \texttt{template}\ \bar{b}\ e\rangle$ is self-respecting.

2. Consider the syntactic possibilities for b:

   (a) $b = \langle\langle \texttt{return}\rangle\rangle$ or $b = \langle\langle \texttt{call}\ n\rangle\rangle$: Then $T(b, f) = \langle b, f\rangle$ and respect is vacuous.

   (b) $b = \langle\langle \texttt{unless-false}\rangle\ b_1\ b_2\rangle$: Then by the induction hypothesis (clause 2), $T(b_2, f) = \langle \bar{b_2}, f_2\rangle$ is self-respecting, as is $T(b_1, f_2) = \langle \bar{b_1}, f'\rangle$. We may apply Lemmas 27 and 35 to infer that $\langle \bar{b_2}, f'\rangle$ is also self-respecting. However, since the instructions z occurring in $\langle\langle \texttt{unless-false}\rangle\ \bar{b_1}\ \bar{b_2}\rangle$ are pecisely those occurring in either $\bar{b_1}$ or $\bar{b_2}$, $T(b, f)$ is self-respecting.

   (c) $b = \langle\langle \texttt{make-cont}\ b_1\ n\rangle :: b_2\rangle$ is similar.

   (d) $b = z :: b_1$: Then by the induction hypothesis (clause 2),

   $$T(b_1, f) = \langle \bar{b_1}, f_1\rangle$$

   is self-respecting, and by clause 3, $T(z, f_1) = \langle \bar{z}, f'\rangle$ is self respecting. The remainder of the argument is like case (b).

3. The syntactic possibilities for y are similar to those of b.

4. Consider the syntactic possibilities for z:

   (a) $z = \langle\langle \texttt{unless-false}\rangle\ y_1\ y_2\rangle$: Similar to case 2(b), except that clause 3 is cited.

   (b) $z = \langle \texttt{literal}\ c\rangle$: $T(\langle \texttt{literal}\ c\rangle, f) = \langle\langle \texttt{literal}\ n\rangle, f'\rangle$, is self-respecting if $f'(n) = \langle \texttt{constant}\ c\rangle$. This is guaranteed because $\mathit{probe}(\langle \texttt{constant}\ c\rangle, f) = \langle n, f'\rangle$.

(c) $z = \langle \texttt{closure}\ t \rangle$: $T(\langle \texttt{closure}\ t \rangle, f) = \langle \langle \texttt{closure}\ n \rangle, f' \rangle$, is self-respecting if $f'(n) = t$. This is guaranteed because $probe(t, f) = \langle n, f' \rangle$.

(d) $z = \langle \texttt{global}\ i \rangle$ or $z = \langle \texttt{set-global!}\ i \rangle$: likewise.

(e) Otherwise respect is vacuous.

6.2 Semantic Correctness

The main correctness theorem states that the procedure `tabulate-top` and `tabulate` are correct in the sense that their output has the same denotation as their (expression) input.

Lemma 37 *1. If $e(n) = e'(n)$ for every n occurring in b, then $\mathcal{B}_\tau[\![b]\!]e = \mathcal{B}_\tau[\![b]\!]e'$;*

2. If $e(n) = e'(n)$ for every n occurring in y, then $\mathcal{Y}_\tau[\![y]\!]e = \mathcal{Y}_\tau[\![y]\!]e'$;

3. If $e(n) = e'(n)$ for every n occurring in z, then $\mathcal{Z}_\tau[\![z]\!]e = \mathcal{Z}_\tau[\![z]\!]e'$.

Proof. The proof is by the usual simultaneous induction.

1. In each case, use the inductive hypotheses and substitute equals for equals in the semantic clauses.

2. As in case 1.

3. Here we show the syntactic cases for z:

   (a) $z = \langle \langle \texttt{unless-false} \rangle\ y_1\ y_2 \rangle$. Since every occurrence in y_1 or y_2 is an occurrence in z, IH clause 2 implies:

   $$\mathcal{Y}_\tau[\![y_1]\!]e = \mathcal{Y}_\tau[\![y_1]\!]e' \quad \text{and} \quad \mathcal{Y}_\tau[\![y_2]\!]e = \mathcal{Y}_\tau[\![y_2]\!]e'.$$

   Thus:

   $$\begin{aligned} \mathcal{Z}_\tau[\![z]\!]e &= \lambda e\rho\pi\,.\, \mathit{if\_truish}(\mathcal{Y}_\tau[\![y_1]\!]e\rho\pi)(\mathcal{Y}_\tau[\![y_2]\!]e\rho\pi) \\ &= \lambda e\rho\pi\,.\, \mathit{if\_truish}(\mathcal{Y}_\tau[\![y_1]\!]e'\rho\pi)(\mathcal{Y}_\tau[\![y_2]\!]e'\rho\pi) \\ &= \mathcal{Z}_\tau[\![z]\!]e'. \end{aligned}$$

   (b) $z = \langle \texttt{literal}\ n \rangle$: As n occurs, $e(n) = e'(n)$.

   $$\begin{aligned} \mathcal{Z}_\tau[\![z]\!]e &= \lambda\rho\,.\, \mathit{literal}\,(\mathcal{K}[\![(e(n))(1)]\!]) \\ &= \lambda\rho\,.\, \mathit{literal}\,(\mathcal{K}[\![(e'(n))(1)]\!]) \\ &= \mathcal{Z}_\tau[\![z]\!]e'. \end{aligned}$$

(c) $z = \langle \texttt{closure}\ n \rangle$: Again, n occurs, so $e(n) = e'(n)$.

$$\begin{aligned} \mathcal{Z}_\tau[\![z]\!]e &= \lambda\rho\,.\,\mathit{closure}\,(\mathcal{T}_\tau[\![e(n)]\!]\rho) \\ &= \lambda\rho\,.\,\mathit{closure}\,(\mathcal{T}_\tau[\![e'(n)]\!]\rho) \\ &= \mathcal{Z}_\tau[\![z]\!]e'. \end{aligned}$$

(d) $z = \langle \texttt{global}\ n \rangle$: Again $e(n) = e'(n)$.

$$\begin{aligned} \mathcal{Z}_\tau[\![z]\!]e &= \lambda\rho\,.\,\mathit{global}\,(\mathit{lookup}\,\rho\,(e(n))(1)) \\ &= \lambda\rho\,.\,\mathit{global}\,(\mathit{lookup}\,\rho\,(e'(n))(1)) \\ &= \mathcal{Z}_\tau[\![z]\!]e'. \end{aligned}$$

(e) $z = \langle \texttt{set-global!}\ n \rangle$: Again $e(n) = e'(n)$.

$$\begin{aligned} \mathcal{Z}_\tau[\![z]\!]e &= \lambda\rho\,.\,\mathit{set\_global}\,(\mathit{lookup}\,\rho\,(e(n))(1)) \\ &= \lambda\rho\,.\,\mathit{set\_global}\,(\mathit{lookup}\,\rho\,(e'(n))(1)) \\ &= \mathcal{Z}_\tau[\![z]\!]e'. \end{aligned}$$

(f) In all other cases, e occurs vacuously in the semantic clause.

Lemma 38 (Meaning stable)
Let $T(b, f) = \langle \bar{b}, f_1 \rangle$, *and assume* $T(w_1, f_1) = \langle \bar{w}, f_2 \rangle$ *or* $T(z_1, f_1) = \langle \bar{z}, f_2 \rangle$. *Then*

$$\mathcal{B}_\tau[\![b]\!]f_1^{-1} = \mathcal{B}_\tau[\![b]\!]f_2^{-1}.$$

Let $T(y, f) = \langle \bar{y}, f_1 \rangle$, *and assume* $T(w_1, f_1) = \langle \bar{w}, f_2 \rangle$ *or* $T(z_1, f_1) = \langle \bar{z}, f_2 \rangle$. *Then*

$$\mathcal{Y}_\tau[\![y]\!]f_1^{-1} = \mathcal{Y}_\tau[\![y]\!]f_2^{-1}.$$

Similarly, let $T(z, f) = \langle \bar{z}, f_1 \rangle$, *and assume* $T(w_1, f_1) = \langle \bar{w}, f_2 \rangle$ *or* $T(z_1, f_1) = \langle \bar{z_1}, f_2 \rangle$. *Then*

$$\mathcal{Z}_\tau[\![z]\!]f_1^{-1} = \mathcal{Z}_\tau[\![z]\!]f_2^{-1}.$$

Proof. Because f_2 extends f_1 (Lemma 27), and because f_1 is defined for indices occurring in the TBC expression (Lemma 32), Lemma 37 ensures that the denotations are equal.

Theorem 39 (Semantic Correctness of the Tabulator)

1. $\mathcal{T}[\![t]\!] = \mathcal{T}_\tau[\![T(t)]\!]$;

2. *For any inverse table* f_0, $T(b, f_0) = \langle \bar{b}, f \rangle$ *implies*

$$\mathcal{B}[\![b]\!] = \mathcal{B}_\tau[\![\bar{b}]\!] f^{-1};$$

3. *For any inverse table* f_0, $T(y, f_0) = \langle \bar{y}, f \rangle$ *implies*

$$\mathcal{Y}[\![y]\!] = \mathcal{Y}_\tau[\![\bar{y}]\!] f^{-1};$$

4. *For any inverse table* f_0, $T(z, f_0) = \langle \bar{z}, f \rangle$ *implies*

$$\mathcal{Z}[\![z]\!] = \mathcal{Z}_\tau[\![\bar{z}]\!] f^{-1};$$

Proof. The proof is by the usual simultaneous induction.

1. Suppose $t = \langle \texttt{lap}\ c\ b \rangle$. Let

$$f_t = \{[\texttt{(constant}\ t\texttt{)} \mapsto 0],\ [c \mapsto 1]\}$$

and let $T(b, f_t) = \langle \bar{b}, f_t' \rangle$. Then using the semantic definition and IH clause 2,

$$\begin{aligned} \mathcal{T}[\![t]\!] &= \mathcal{B}[\![b]\!] \\ &= \mathcal{B}_\tau[\![\bar{b}]\!] e, \end{aligned}$$

where $e = (f_t')^{-1}$. Moreover, by Lemmas 29 and 30, 0 does not occur as a template index in $\bar{b}$. By Lemma 37, letting e' be e with $[\texttt{(constant 0)} \mapsto 0]$, it follows that

$$\begin{aligned} \mathcal{B}_\tau[\![\bar{b}]\!] e &= \mathcal{B}_\tau[\![\bar{b}]\!] e' \\ &= \mathcal{B}_\tau[\![\bar{b}]\!] e' \\ &= \mathcal{T}_\tau[\![\langle \texttt{template}\ \bar{b}\ e' \rangle]\!], \end{aligned}$$

which is $T(\langle \texttt{lap}\ c\ b \rangle)$.

2. Take cases on the syntactic form of b:

   (a) $b = \langle\langle \texttt{return} \rangle\rangle$ or $b = \langle\langle \texttt{call}\ n \rangle\rangle$: $\bar{b} = b$, and the semantic clauses ensure that $\mathcal{B}[\![b]\!] = \mathcal{B}_\tau[\![b]\!] e$, for all e.

(b) $b = \langle\langle \texttt{unless-false}\ b_1\ b_2\rangle\rangle$: Let $T(b_2, f) = \langle \bar{b_2}, f_2\rangle$, and let $T(b_1, f_2) = \langle \bar{b_1}, f'\rangle$, so that

$$T(b, f) = \langle\langle\langle \texttt{unless-false}\ \bar{b_1}\ \bar{b_2}\rangle\rangle, f'\rangle.$$

Let e be $(f')^{-1}$. By Lemma 38, $\mathcal{B}_\tau[\![\bar{b_2}]\!]e = \mathcal{B}_\tau[\![\bar{b_2}]\!]f_2^{-1}$. Hence, using the semantic clauses and IH, clause 2:

$$\begin{aligned}
\mathcal{B}_\tau[\![\langle\langle \texttt{unless-false} \quad \bar{b_1} \quad \bar{b_2}\rangle\rangle]\!]e & \\
&= \lambda\rho\,.\, \mathit{if_truish}(\mathcal{B}_\tau[\![\bar{b_1}]\!]e\rho)(\mathcal{B}_\tau[\![\bar{b_2}]\!]e\rho) \\
&= \lambda\rho\,.\, \mathit{if_truish}(\mathcal{B}_\tau[\![b_1]\!]\rho)(\mathcal{B}_\tau[\![b_2]\!]\rho) \\
&= \mathcal{B}[\![\langle\langle \texttt{unless-false}\ b_1\ b_2\rangle\rangle]\!]
\end{aligned}$$

(c) $b = \langle \texttt{make-cont}\ b_1\ n\rangle :: b_2$ is similar.

(d) Suppose $b = z :: b_1$. Let $T(b_1, f) = \langle \bar{b_1}, f_1\rangle$, and let $T(z, f_1) = \langle \bar{z}, f'\rangle$, so that

$$T(b, f) = \langle \bar{z} :: \bar{b_1}, f'\rangle.$$

Let e be $(f')^{-1}$. Citing Lemma 38, the semantic clauses, and IH, clauses 2 and 4:

$$\begin{aligned}
\mathcal{B}_\tau[\![\bar{z} :: \bar{b_1}]\!]e &= \lambda\rho\,.\, \mathcal{Z}_\tau[\![\bar{z}]\!]e\rho\,(\mathcal{B}_\tau[\![\bar{b_1}]\!]e\rho) \\
&= \lambda\rho\,.\, \mathcal{Z}[\![z]\!]\rho\,(\mathcal{B}[\![b]\!]\rho) \\
&= \mathcal{B}[\![z :: b]\!]
\end{aligned}$$

3. Open instruction lists y are similar to closed ones. We wiull show only one case:

(b) $y = \langle \texttt{make-cont}\ \langle\rangle\ n\rangle :: b$: Let $T(b, f) = \langle \bar{b}, f'\rangle$, so that

$$T(y, f) = \langle\langle \texttt{make-cont}\ \langle\rangle\ n\rangle :: \bar{b}, f'\rangle.$$

Let e be $(f')^{-1}$. Hence, using the semantic clauses, and IH, Clause 2:

$$\begin{aligned}
\mathcal{Y}_\tau[\![\bar{y}]\!]e\rho\pi &= \mathit{make_cont}\,(\pi)\ n\ (\mathcal{B}_\tau[\![\bar{b}]\!]e\rho) \\
&= \mathit{make_cont}\,(\pi)\ n\ (\mathcal{B}[\![b]\!]\rho) \\
&= \mathcal{Y}[\![y]\!]\rho\pi
\end{aligned}$$

4. Neutral machine instructions z:

(a) $z = \langle \texttt{unless-false}\ y_1\ y_2 \rangle$: Let $T(y_2, f) = \langle \bar{y_2}, f_2 \rangle$, and let $T(y_1, f_2) = \langle \bar{y_1}, f' \rangle$, so that

$$T(z, f) = \langle \langle \langle \texttt{unless-false}\ \bar{y_1}\ \bar{y_2} \rangle \rangle, f' \rangle.$$

Citing Lemma 38, the semantic clauses, and IH, clause 3,

$$\begin{array}{lll} \mathcal{Z}_\tau[\![\langle \texttt{unless-false} & \bar{y_1} & \bar{y_2} \rangle]\!]e \\ & = & \lambda\rho\pi\,.\,\mathit{if\_truish}(\mathcal{Y}_\tau[\![\bar{y_1}]\!]e\rho\pi)(\mathcal{Y}_\tau[\![\bar{y_2}]\!]e\rho\pi) \\ & = & \lambda\rho\,.\,\mathit{if\_truish}(\mathcal{Y}[\![y_1]\!]\rho)(\mathcal{Y}[\![y_2]\!]\rho) \\ & = & \mathcal{Z}[\![\langle \langle \texttt{unless-false}\ y_1\ y_2 \rangle \rangle]\!] \end{array}$$

(b) $z = \langle \texttt{literal}\ c \rangle$: So, letting $\mathit{probe}(\langle \texttt{constant}\ c \rangle, f) = \langle n, f' \rangle$, and let $(f')^{-1} = e$. So

$$T(z, f) = \langle \langle \texttt{literal}\ n \rangle, f' \rangle$$

and $f'(\langle \texttt{constant}\ c \rangle) = n$. Hence $e(n)(1) = c$, and

$$\begin{array}{rcl} \mathcal{Z}[\![z]\!] & = & \lambda\rho\,.\,\mathit{literal}\,(\mathcal{K}[\![c]\!]) \\ & = & \lambda\rho\,.\,\mathit{literal}\,(\mathcal{K}[\![e(n)(1)]\!]) \\ & = & \mathcal{Z}_\tau[\![\langle \texttt{literal}\ n \rangle]\!]e \end{array}$$

(c) $z = \langle \texttt{closure}\ t \rangle$: Let $T(t) = \bar{t}$; by the IH, clause 1, $\mathcal{T}[\![t]\!] = \mathcal{T}_\tau[\![\bar{t}]\!]$. Let $\mathit{probe}(t, f) = \langle n, f' \rangle$, and let $(f')^{-1} = e$. So

$$T(z, f) = \langle \langle \texttt{closure}\ n \rangle, f' \rangle,$$

and $f'(t) = n$. Hence

$$\begin{array}{rcl} \mathcal{Z}[\![z]\!] & = & \lambda\rho\,.\,\mathit{closure}\,(\mathcal{T}[\![t]\!]\rho) \\ & = & \lambda\rho\,.\,\mathit{closure}\,(\mathcal{T}_\tau[\![e(n)]\!]\rho) \\ & = & \mathcal{Z}_\tau[\![\langle \texttt{closure}\ n \rangle]\!]e \end{array}$$

(d) `global` and `set-global!` are similar to `literal`.

(e) In all other cases, $T(z, f) = \langle z, f \rangle$, and the semantic for TBC simply ignore their argument e, and return the same value as the corresponding clause for BBC.

References

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

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

[3] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2), 1984.

[4] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Wm. C. Brown, Dubuque, IA, 1986.

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