The VLISP Linker

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

The MITRE Corporation[1]

M92B095
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.
©1992 The MITRE Corporation. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the MITRE copyright notice and the title of the publication and its date appear, and notice in given that copying is by permission of the MITRE Corporation.

Abstract

The Verified Programming Language Implementation project has developed a formally verified implementation of the Scheme programming language. This report documents the linker, which lays out a single data structure for a program consisting of constants, variables, and procedures. It contains detailed proofs that the operational semantics of the linked output matches the operational semantics of the input.

Contents

1 Introduction

This paper discusses the VLISP linker, an algorithm which lays out a single data structure for a program consisting of constants, variables, and procedures. The main procedure of the linker, `link-template`, produces a program of Linked Byte Code (LBC) from a template of Flattened Byte Code (FBC). LBC and its semantics are given in section 2. The linker algorithm is presented in section 3, and its syntactic and semantic correctness are proved in section 4.

1.1 Notation

Finite sequences will be formed using $\langle \ldots \rangle$ notation. Commas are optional between sequence members; we will usually not use commas in expressions of FBC and LBC. $f(a)\downarrow$ means the function f is defined at a. Similarly, $f(a)\uparrow$ means the function f is undefined at a.

2 Linked Byte Code State Machines

This section presents the LBC and gives it an (operational) semantics. Both the syntax and semantics of LBC are very similar to the syntax and semantics of FBC presented in [1]. This paper assumes familiarity with the [1], particularly the sections on state machines in general and FBC state machines.

2.1 Syntax of the Linked Byte Code Language

In the higher-level VLISP languages, the only notion of program that was relevant was a template, including whatever other templates it referenced (possibly through intermediate templates). LBC, by contrast, has a particular syntax for a program, in which all the templates are gathered together in a single linear sequence. In addition, LBC has two other sequences, gathering (respectively) all constants and all global variables referenced throughout the whole program. These sequences are arranged so that if one template refers to another, or if one constant has another as a component, then the template or constant referred to occurs earlier in its sequence than the one referring to it. This property makes it easy for the loader to lay the objects out sequentially in a binary format, and to find an address for an object whenever a later object refers to it.

The BNF grammar for LBC (given in Table 1) is very similar to the grammar for FBC. In fact, the productions for t, e, w, and m are exactly the same. There are two new syntactic classes: p is for *programs* and $\tilde{o}$ is for *pseudoliterals*. An LBC program

$$p = \langle n \ \texttt{constants} :: c^* \ \texttt{global-variables} :: n^* \rangle ^\frown t^+$$

is *proper* if:

(1) For all $k < \#c^*$, if $c^*(k) = \langle \texttt{immutable-pair} \ n_1 \ n_2 \rangle$, then $1 \leq n_1, n_2 \leq k$, and if $c^*(k) = \langle \texttt{immutable-vector} \ n_1 \cdots n_j \rangle$, then $1 \leq n_1, \ldots, n_j \leq k$.

(2) For all $k < \#n^*$, $1 \leq n^*(k) \leq \#c^*$ and $c^*(n^*(k) - 1)$ is a symbol.

(3) For all $k < \#t^+$, if $t^+(k) = \langle \texttt{template} \ w \ \langle o_1 \cdots o_j \rangle \rangle$, then for all o_i with $1 \leq i \leq j$:

   (a) if $o_i = \langle \texttt{constant} \ n' \rangle$, then $1 \leq n' \leq \#c^*$;

   (b) if $o_i = \langle \texttt{global-variable} \ n' \rangle$, then $1 \leq n' \leq \#n^*$;

   (c) if $o_i = \langle \texttt{template} \ n' \rangle$, then $1 \leq n' \leq k$.

(4) $n = \#t^+$.[1]

Hence, a proper program satisfies the reference conditions mentioned above.

The Augmented Linked Byte Code (ALBC) is an expansion of LBC in exactly the same way AFBC is an expansion of FBC [1].

2.2 States and Actions

LBC states are defined in just the same way as FBC states; the only difference is that variables over syntactic classes of code are interpreted as ranging over LBC expressions rather than FBC expressions.

Although LBC states are sequences of the form

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

like FBC states, the behavior of an LBC machine depends on two parameters, instead of just one (i.e., *globals*), which do not change during the execution of the state machine. The first parameter, called *globals'*, is a function from

[1]This n clearly adds no new information to p since $n = \#p - 3$, provided p is proper.

$$\begin{array}{rcl}
p & ::= & \langle n \; \mathtt{constants} :: c^* \; \mathtt{global\text{-}variables} :: n^* \rangle {}^\frown t^+ \\
\tilde{o} & ::= & o \mid t \mid \langle \mathtt{constant} \; c \rangle \\
t & ::= & \langle \mathtt{template} \; w \; e \rangle \\
e & ::= & o^* \\
c & ::= & c_{pr} \mid \text{strings} \mid \langle \mathtt{immutable\text{-}pair} \; n_1 \; n_2 \rangle \mid \mathtt{immutable\text{-}vector} :: n^* \\
o & ::= & \langle \mathtt{constant} \; n \rangle \mid \langle \mathtt{global\text{-}variable} \; n \rangle \mid \langle \mathtt{template} \; n \rangle \\
w & ::= & \langle\rangle \mid m {}^\frown w \\
m & ::= & \langle \mathtt{call} \; n \rangle \mid \langle \mathtt{return} \rangle \mid \langle \mathtt{make\text{-}cont} \; n_0 \; n_1 \; n_2 \rangle \mid \langle \mathtt{literal} \; n \rangle \\
 & \mid & \langle \mathtt{closure} \; n \rangle \mid \langle \mathtt{global} \; n \rangle \mid \langle \mathtt{local} \; n_0 \; n_1 \rangle \mid \langle \mathtt{set\text{-}global!} \; n \rangle \\
 & \mid & \langle \mathtt{set\text{-}local!} \; n_0 \; n_1 \rangle \mid \langle \mathtt{push} \rangle \mid \langle \mathtt{make\text{-}env} \; n \rangle \mid \langle \mathtt{make\text{-}rest\text{-}list} \; n \rangle \\
 & \mid & \langle \mathtt{unspecified} \rangle \mid \langle \mathtt{jump} \; n_0 \; n_1 \rangle \mid \langle \mathtt{jump\text{-}if\text{-}false} \; n_0 \; n_1 \rangle \\
 & \mid & \langle \mathtt{checkargs=} \; n \rangle \mid \langle \mathtt{checkargs>=} \; n \rangle \mid \langle i \rangle
\end{array}$$

Table 1: Grammar for Linked Byte Code

some finite set of natural numbers to locations. The second parameter is the program that the machine is executing, a value in the LBC syntactic class p given in Table 1. However, the parameter p plays a role only in determining two auxiliary functions: u_c and u_t. For

$$p = \langle n \; \mathtt{constants} :: c^* \; \mathtt{global\text{-}variables} :: n^* \rangle {}^\frown t^+,$$

if $1 \leq k \leq \#c^*$, then $u_c(k) = c^*(k-1)$, and if $1 \leq k \leq \#t^+$, $u_t(k) = t^+(k-1)$. Hence, p will not appear below; only these auxiliary functions will appear.

All but four of the pure rules of LBC are syntactically identical to the corresponding rules for FBC. The four rules that differ are those that use the literals. We will not repeat the other rules here.

Rule: **Literal**
Domain conditions:
$w \dagger n = \langle \texttt{literal}\ n_1 \rangle \frown w_1$
$e(n_1) = \langle \texttt{constant}\ n_2 \rangle$
$u_c(n_2) = c$
Changes:
$n' = n + 2$
$v' = c$

Rule: **Closure**
Domain conditions:
$w \dagger n = \langle \texttt{closure}\ n_1 \rangle \frown w_1$
$e(n_1) = \langle \texttt{template}\ n_2 \rangle$
$u_t(n_2) = t_1$
Changes:
$n' = n + 2$
$v' = \langle \textsc{closure}\ t_1\ u\ \#s \rangle$
$s' = s \frown \langle \textsc{not-specified} \rangle$

Rule: **Global**
Domain conditions:
$w \dagger n = \langle \texttt{global}\ n_1 \rangle \frown w_1$
$e(n_1) = \langle \texttt{global-variable}\ n_2 \rangle$
$l = \mathit{globals}'(n_2)$
$v_1 = s(l)$
$v_1 \neq \textsc{undefined}$
Changes:
$n' = n + 2$
$v' = v_1$

Rule: **Set Global**
Domain conditions:
$w \dagger n = \langle \texttt{set-global!}\ n_1 \rangle \frown w_1$
$e(n_1) = \langle \texttt{global-variable}\ n_2 \rangle$
$l = \mathit{globals}'(n_2)$
$l \leq \#s$
Changes:
$n' = n + 2$
$v' = \textsc{not-specified}$
$s' = s + \{l \mapsto v\}$

3 The Linker Algorithm

The procedure `link-template` generates a literal table from an FBC template. The literal table is then converted into an LBC program by `assemble-linked-template`. (The notion of a "literal table" is defined in subsection 4.2.)

```
(define (link-template template)
  (let ((tab-lit (link-literal empty-literal-table template)))
    (assemble-linked-template
        (tab-lit-table tab-lit)
        (tab-lit-literal tab-lit))))

(define (assemble-linked-template tab lit)
  `(,(cadr lit)
    (constants
     . ,(hash-table-data (literal-table-constants tab)))
    (global-variables
     . ,(hash-table-data (literal-table-global-variables tab)))
    . ,(template-table-data (literal-table-templates tab))))
```

The central procedure of the linker is `link-literal`. Its code is very simple since it treats literals as abstract objects. Its subprocedures—`component-literals`, `make-linked-literal`, and `lookup-in-literal-table`—dispatch on the syntax of their arguments.

```
(define (link-literal tab lit)
  (let* ((lits (component-literals lit))
         (tab-lits (link-literals tab lits))
         (new-lit (make-linked-literal
                   lit
                   (tab-lits-literals tab-lits))))
    (lookup-in-literal-table
     (tab-lits-table tab-lits)
     new-lit)))

(define (link-literals tab lits)
  (link-literals-aux tab lits '()))
```

```
(define (link-literals-aux tab old-lits new-lits)
  (if (null? old-lits)
      (cons tab (reverse new-lits))
      (let ((tab-lit (link-literal tab (car old-lits))))
        (link-literals-aux
         (tab-lit-table tab-lit)
         (cdr old-lits)
         (cons (tab-lit-literal tab-lit) new-lits)))))

(define tab-lit-table car)
(define tab-lit-literal cdr)
(define tab-lits-table car)
(define tab-lits-literals cdr)
```

The three dispatch procedures are given below.

```
(define (component-literals lit)
  (cond ((or (atomic-constant? lit)
             (global-variable? lit))
         '())
        ((pair-constant? lit)
         (list (list 'constant (car (cadr lit)))
               (list 'constant (cdr (cadr lit)))))
        ((vector-constant? lit)
         (map
          (lambda (x) (list 'constant x))
          (vector->list (cadr lit))))
        ((template? lit)
         (caddr lit))
        (else (compiler-error "Bad literal." lit))))

(define (make-linked-literal lit lits)
  (cond ((or (atomic-constant? lit)
             (global-variable? lit))
         lit)
        ((pair-constant? lit)
         (list 'constant (cons 'pair (map cadr lits))))
        ((vector-constant? lit)
         (list 'constant (cons 'vector (map cadr lits))))
        ((template? lit)
         (list 'template (cadr lit) lits))
        (else (compiler-error "Bad literal." lit))))

(define (lookup-in-literal-table tab lit)
  (cond ((or (atomic-constant? lit)
             (pair-constant? lit)
             (vector-constant? lit))
         (lookup-constant-in-literal-table tab (cadr lit)))
        ((global-variable? lit)
         (lookup-global-variable-in-literal-table
          tab
          (cadr lit)))
        ((template? lit)
         (lookup-template-in-literal-table tab lit))
        (else (compiler-error "Bad literal." lit))))

(define (atomic-constant? lit)
  (and (eq? (car lit) 'constant)
       (not (pair? (cadr lit)))
       (not (vector? (cadr lit)))))

(define (pair-constant? lit)
  (and (eq? (car lit) 'constant)
       (pair? (cadr lit))))

(define (vector-constant? lit)
  (and (eq? (car lit) 'constant)
       (vector? (cadr lit))))

(define (global-variable? lit)
  (eq? (car lit) 'global-variable))

(define (template? lit)
  (eq? (car lit) 'template))
```

The following are the low-level procedures for building and manipulating literal tables. These procedures could be pared down somewhat.

```
(define (make-literal-table c-table gv-table t-table)
  (vector c-table gv-table t-table))
```

```
(define (literal-table-constants tab)
  (vector-ref tab 0))

(define (literal-table-global-variables tab)
  (vector-ref tab 1))

(define (literal-table-templates tab)
  (vector-ref tab 2))

(define empty-literal-table
  (make-literal-table  (list 1 (make-vector 512 '()))
                       (list 1 (make-vector 512 '()))
                       '(1)))

(define (literal-table->pairs tab)
  (let ((c-pairs
         (reverse
          (map
           (lambda (x)
             (cons (cons 'constant (cdr x))
                   (list 'constant (car x))))
           (vector->list
            (cdr (literal-table-constants tab))))))
        (gv-pairs
         (reverse
          (map
           (lambda (x)
             (cons (cons 'global-variable (cdr x))
                   (list 'global-variable (car x))))
           (vector->list
            (cdr (literal-table-global-variables tab))))))
        (t-pairs
         (reverse
          (map
           (lambda (x)
             (cons (cons 'template (cdr x))
                   (car x)))
           (cdr (literal-table-templates tab))))))
    (append c-pairs gv-pairs t-pairs)))
```

```
(define (lookup-constant-in-literal-table tab constant)
  (let ((probe (lookup-data-in-hash-table
                (literal-table-constants tab)
                constant)))
    (cons
     (make-literal-table
      (probe-table probe)
      (literal-table-global-variables tab)
      (literal-table-templates tab))
     (list 'constant (probe-value probe)))))

(define (lookup-global-variable-in-literal-table
         tab global-variable)
  (let* ((tab-lit
          (lookup-constant-in-literal-table
           tab
           global-variable))
         (probe (lookup-data-in-hash-table
                 (literal-table-global-variables tab)
                 (cadr (tab-lit-literal tab-lit)))))
    (cons
     (make-literal-table
      (literal-table-constants (tab-lit-table tab-lit))
      (probe-table probe)
      (literal-table-templates (tab-lit-table tab-lit)))
     (list 'global-variable (probe-value probe)))))

(define (lookup-template-in-literal-table tab template)
  (let ((probe (enter-data-in-template-table
                (literal-table-templates tab)
                template)))
    (cons
     (make-literal-table
      (literal-table-constants tab)
      (literal-table-global-variables tab)
      (probe-table probe))
     (list 'template (probe-value probe)))))
```

```
(define probe-table car)

(define probe-value cdr)

(define make-probe cons)

(define (template-table-data table)
  (reverse (cdr table)))

(define (hash-table-data table)
  (reverse (cddr table)))

(define (lookup-data-in-hash-table table key)
  (let* ((counter (car table))
         (hash-table (cadr table))
         (data-list (cddr table))
         (hash-index (linker-hash key))
         (indexed-list (vector-ref hash-table hash-index))
         (query (assoc key indexed-list)))
    (if query
        (make-probe table (cdr query))
        (begin
          (vector-set!
           hash-table
           hash-index
           (cons (cons key counter) indexed-list))
          (make-probe
           (cons (+ 1 counter)
                 (cons hash-table (cons key data-list)))
           counter)))))

(define (enter-data-in-template-table table key)
  (make-probe (cons (+ 1 (car table))
                    (cons key (cdr table)))
              (car table)))
```

```
(define (hash-string s hash-prime)
  (let ((len (string-length s)))
    (let loop ((i 0)
               (acc 0))
      (if (= i len) acc
          (loop (+ i 1)
                (remainder
                 (+ (* acc 256)
                    (char->integer (string-ref s i)))
                 hash-prime))))))

(define (hash-vector x hash-prime)
  (let loop ((lst (cdr x))
             (acc 0))
      (if (null? lst) acc
          (loop (cdr lst)
                (remainder
                 (+ (* acc 512) (car lst))
                 hash-prime)))))

(define (linker-hash x)
  (let ((hash-prime 509))
    (cond ((number? x) (remainder (abs x) hash-prime))
          ((symbol? x)
           (hash-string (symbol->string x) hash-prime))
          ((string? x) (hash-string x hash-prime))
          ((and (pair? x) (eq? (car x) 'pair))
           (remainder (* (cadr x) (caddr x)) hash-prime))
          ((char? x) (char->integer x))
          ((and (pair? x) (eq? (car x) 'vector))
           (hash-vector x hash-prime))
          ((boolean? x) (if x 510 511))
          ((null? x) 511)
          (#t
           (compiler-error
            'linker 'hash "Invalid constant" x)))))
```

4 Correctness of the Linker Algorithm

We show in this section that `link-template` is correct in two ways:

(1) `link-template` always produces a proper LBC program from an FBC template (Corollary 4.9).

(2) The execution of an LBC program p produced by `link-template` from an FBC template t mirrors the execution of t (see Theorem 4.10).

For the most part, the verification of the low-level procedures for building and manipulating literal tables is left to the reader. Except for the hash function routines, these procedures are unchallenging.

Throughout this section we shall refer to "template literals," "immutable pairs," and "immutable vectors" of both FBC and LBC as simply "literals," "pairs," and "vectors," respectively. By a "pseudoliteral" we shall mean an FBC literal or an LBC pseudoliteral; both FBC and LBC pseudoliterals will be denoted by the symbol $\tilde{o}$.

We shall assume that ζ is an arbitrary injective function from identifiers to symbols.[2]

4.1 Preliminary Definitions

A *kind* is a member of the set

$$kinds = \{\texttt{constant}, \texttt{global-variable}, \texttt{template}\}.$$

We shall sometimes use `c`, `g`, and `t` as shorthand for the three members of *kinds*. The *kind* of a pseudoliteral $\tilde{o}$, written $kind(\tilde{o})$, is the kind $\tilde{o}(0)$.

The *components* of a pseudoliteral $\tilde{o}$, written $comp(\tilde{o})$, is a list of pseudoliterals defined as follows:

[2] In the linker algorithm, the function corresponding to ζ maps identifiers to the symbols from which they are constructed. However, the correctness of the linker algorithm does not depend on what ζ actually is.

$$comp(\tilde{o}) = \begin{cases} \langle\rangle & \text{if } \tilde{o} \text{ is } \langle\texttt{constant } x\rangle \text{ and } x \text{ is neither a pair nor a vector} \\ \langle x'_1\ x'_2\rangle & \text{if } \tilde{o} \text{ is } \langle\texttt{constant } x\rangle \text{ and } x \text{ is } \langle\texttt{immutable-pair } x_1\ x_2\rangle \\ \langle x'_1 \cdots x'_m\rangle & \text{if } \tilde{o} \text{ is } \langle\texttt{constant } x\rangle \text{ and } x \text{ is } \langle\texttt{immutable-vector } x_1 \cdots x_m\rangle \\ \langle\rangle & \text{if } \tilde{o} \text{ is } \langle\texttt{global-variable } x\rangle \\ \langle\rangle & \text{if } \tilde{o} \text{ is } \langle\texttt{template } n\rangle \\ o^* & \text{if } \tilde{o} \text{ is } \langle\texttt{template } w\ o^*\rangle \end{cases}$$

where $x'_i = \langle\texttt{constant } x_i\rangle$. A pseudoliteral $\tilde{o}$ is *atomic* if $comp(\tilde{o}) = \langle\rangle$ and is *compound* otherwise. The *depth* of a pseudoliteral $\tilde{o}$, written $depth(\tilde{o})$, is defined inductively by the following:

(1) If $\tilde{o}$ is atomic, then $depth(\tilde{o}) = 0$.

(2) If $\tilde{o}$ is compound, then $depth(\tilde{o}) = 1 + max\{depth(\tilde{o}') : \tilde{o}' \in comp(\tilde{o})\}$.

Let $\mathcal{L}_{\rm fbc}$, $\mathcal{A}_{\rm fbc}$, and $\mathcal{C}_{\rm fbc}$ denote the sets of literals, atomic literals, and compound literals, respectively, of FBC. Also, let $\tilde{\mathcal{L}}_{\rm lbc}$, $\tilde{\mathcal{A}}_{\rm lbc}$, and $\tilde{\mathcal{C}}_{\rm lbc}$ denote the sets of pseudoliterals, atomic pseudoliterals, and compound pseudoliterals, respectively, of LBC. Obviously, $\mathcal{L}_{\rm fbc} = \mathcal{A}_{\rm fbc} \cup \mathcal{C}_{\rm fbc}$ and $\tilde{\mathcal{L}}_{\rm lbc} = \tilde{\mathcal{A}}_{\rm lbc} \cup \tilde{\mathcal{C}}_{\rm lbc}$. Also, let $\mathcal{L}_{\rm lbc}$ denote the set of LBC literals (which are always atomic). Notice that $(\tilde{\mathcal{A}}_{\rm lbc} \setminus \mathcal{L}_{\rm lbc}) \subseteq \mathcal{A}_{\rm fbc}$.

Suppose $\tilde{o} \in \tilde{\mathcal{C}}_{\rm lbc}$ and $o_1, \ldots, o_m \in \mathcal{L}_{\rm fbc}$ with $m = \#comp(\tilde{o})$. Then $make\text{-}flat\text{-}lit(\tilde{o}, \langle o_1 \cdots o_m\rangle)$ is:

(1) $\langle\texttt{constant } \langle\texttt{immutable-pair } c_1\ c_2\rangle\rangle$
if $\tilde{o} = \langle\texttt{constant } \langle\texttt{immutable-pair } n_1\ n_2\rangle\rangle$
and $o_i = \langle\texttt{constant } c_i\rangle$ for $i = 1, 2$;

(2) $\langle\texttt{constant } \langle\texttt{immutable-vector } c_1 \cdots c_m\rangle\rangle$
if $\tilde{o} = \langle\texttt{constant } \langle\texttt{immutable-vector } n_1 \cdots n_m\rangle\rangle$
and $o_i = \langle\texttt{constant } c_i\rangle$ for $i = 1, \ldots, m$;

(3) $\langle\texttt{template } w\ \langle o_1 \cdots o_m\rangle\rangle$ if $\tilde{o} = \langle\texttt{template } w\ o^*\rangle$; and

(4) undefined otherwise.

It is easy to see that $make\text{-}flat\text{-}lit(\tilde{o}, \langle o_1 \cdots o_m\rangle) \in \mathcal{L}_{\rm fbc}$ provided it is defined.

4.2 Literal Stores and Tables

Let $\tilde{\mathcal{L}}'_{\rm lbc} = \tilde{\mathcal{L}}_{\rm lbc} \setminus \{\langle k\ n\rangle : k = \texttt{c}, \texttt{t}\}$. A *literal store* is an injective function $f : \mathit{kinds} \times \mathbf{N}^+ \to \tilde{\mathcal{L}}'_{\rm lbc}$ with finite domain such that, whenever $f(k,n)\!\downarrow$, the following hold:

(1) $\mathit{kind}(f(k,n)) = k$.

(2) $f(k,m)\!\downarrow$ for all m with $1 \leq m < n$.

(3) If $k = \texttt{c}$, then, for all $\langle \texttt{c}\ m\rangle \in \mathit{comp}(f(k,n))$, $m < n$.

(4) If $k = \texttt{g}$, then for some $m \in \mathbf{N}^+$ $f(\texttt{g},n) = \langle \texttt{g}\ m\rangle$ and $f(\texttt{c},m) = \langle \texttt{c}\ c\rangle$ where $c \in \mathit{ran}(\zeta)$.

(5) If $k = \texttt{t}$, then, for all $\langle k'm\rangle \in \mathit{comp}(f(k,n))$, $f(k',m)\!\downarrow$ if $k' = \texttt{c}, \texttt{g}$, and $m < n$ if $k' = \texttt{t}$.

Let *stores* denote the set of literal stores.

A Scheme data structure τ is a *literal table* if one of the following hold:

(1) τ is produced by `empty-literal-table`.

(2) $\langle \tau, o\rangle$ is produced by `lookup-in-literal-table` from a literal table and a member of $\tilde{\mathcal{L}}'_{\rm lbc}$.

A literal table τ *represents* a literal store f if `literal-table->pairs` produces a list $\langle\langle\langle a_1, b_1\rangle, c_1\rangle \cdots \langle\langle a_m, b_m\rangle, c_m\rangle\rangle$ from τ such that:

(1) $f(a_i, b_i) = c_i$ for each i with $1 \leq i \leq m$; and

(2) $f(k,n)\!\downarrow$ implies $k = a_i$ and $n = b_i$ for some i with $1 \leq i \leq m$.

Lemma 4.1 *The procedure* `empty-literal-table` *produces a literal table which represents the literal store with the empty domain.*

Proof By inspection of the Scheme code for `empty-literal-table` and `literal-table->pairs`. □

The function $\mathit{lookup} : (\mathit{stores} \times \mathcal{A}_{\rm fbc} \cup \tilde{\mathcal{C}}_{\rm lbc}) \to (\mathit{stores} \times \mathcal{L}_{\rm lbc})$ is defined as follows:

(1) Let $\langle f, \tilde{o}\rangle \in (\mathit{stores} \times \mathcal{A}_{\rm fbc} \cup \tilde{\mathcal{C}}_{\rm lbc})$ with $\tilde{o}$ in the range of f. Then $\mathit{lookup}(f, \tilde{o}) = \langle f, \langle k\ n\rangle\rangle$, where $f(k,n) = \tilde{o}$.

(2) Let $\langle f, \tilde{o} \rangle \in (\mathit{stores} \times \mathcal{A}_{\mathrm{fbc}} \cup \tilde{\mathcal{C}}_{\mathrm{lbc}})$ such that (a) $k = \mathit{kind}(\tilde{o}) \neq \mathtt{g}$ and (b) $\tilde{o}$ not is in the range of f but each $o' \in \mathit{comp}(\tilde{o})$ is in the range of f. Suppose n is the least positive integer such that $f(k, n)\!\uparrow$. Then $\mathit{lookup}(f, \tilde{o}) = \langle f', \langle k\ n \rangle \rangle$, where $f' = f \cup \{\langle \langle k, n \rangle, \tilde{o} \rangle\}$.

(3) Let $\langle f, \tilde{o} \rangle \in (\mathit{stores} \times \mathcal{A}_{\mathrm{fbc}})$ such that (a) $\tilde{o} = \langle \mathtt{g}\ i \rangle$ and (b) $\tilde{o}$ not is in the range of f. Let $\langle f', \langle \mathtt{c}\ m \rangle \rangle = \mathit{lookup}(f, \langle \mathtt{c}\ \zeta(i) \rangle)$. Suppose n is the least positive integer such that $f'(\mathtt{g}, n)\!\uparrow$. Then $\mathit{lookup}(f, \tilde{o}) = \langle f'', \langle \mathtt{g}\ n \rangle \rangle$, where $f'' = f' \cup \{\langle \langle \mathtt{g}, n \rangle, \langle \mathtt{g}\ m \rangle \rangle\}$.

(4) Let $\langle f, \tilde{o} \rangle \in (\mathit{stores} \times \mathcal{A}_{\mathrm{fbc}} \cup \tilde{\mathcal{C}}_{\mathrm{lbc}})$ with $\tilde{o}$ and some $o' \in \mathit{comp}(\tilde{o})$ not in the range of f. Then $\mathit{lookup}(f, \tilde{o})\!\uparrow$.

Proposition 4.2 *The function lookup is well-defined, i.e., if* $\langle f, \tilde{o} \rangle \in (\mathit{stores} \times \mathcal{A}_{\mathrm{fbc}} \cup \tilde{\mathcal{C}}_{\mathrm{lbc}})$, *then* $\mathit{lookup}(f, \tilde{o}) \in (\mathit{stores} \times \mathcal{L}_{\mathrm{lbc}})$ *for cases (1), (2), and (3). Moreover,* $\mathit{lookup}(f, \tilde{o}) = \langle f', o \rangle$ *implies* f' *extends* f *and* $\mathit{kind}(o) = \mathit{kind}(\tilde{o})$.

Proof Straightforward. □

Lemma 4.3 *Let* τ *be a literal table representing* f, $\tilde{o} \in \mathcal{A}_{\mathrm{fbc}} \cup \tilde{\mathcal{C}}_{\mathrm{lbc}}$, $\langle \tau', o \rangle$ *be produced by* `lookup-in-literal-table` *from* τ *and* $\tilde{o}$, *and* $\langle f', o' \rangle = \mathit{lookup}(f, \tilde{o})$. *Then* τ' *represents* f' *and* $o = o'$.

Proof By a comparison of the Scheme code for `lookup-in-literal-table` with the definition of *lookup*. □

Proposition 4.4 *Every literal table represents a literal store.*

Proof Follows from Lemma 4.1 and the previous lemma. □

The function $\mathit{extract} : (\mathit{stores} \times \mathcal{L}_{\mathrm{lbc}} \cup \tilde{\mathcal{C}}_{\mathrm{lbc}}) \to \mathcal{L}_{\mathrm{fbc}}$ is defined recursively by:

(1) If $f \in \mathit{stores}$ and $o = \langle k\ n \rangle \in \mathcal{L}_{\mathrm{lbc}}$ with $f(k, n) \in \mathcal{A}_{\mathrm{fbc}}$, then $\mathit{extract}(f, o) = f(k, n)$.

(2) If $f \in \mathit{stores}$ and $o = \langle k\ n \rangle \in \mathcal{L}_{\mathrm{lbc}}$ with $f(k, n) \in \tilde{\mathcal{C}}_{\mathrm{lbc}}$ and $k \neq \mathtt{g}$, then $\mathit{extract}(f, o) = \mathit{extract}(f, f(k, n))$.

(3) If $f \in \mathit{stores}$ and $o = \langle \mathtt{g}\ n \rangle \in \mathcal{L}_{\mathrm{lbc}}$, then $\mathit{extract}(f, o) = \langle \mathtt{g}\ i \rangle$, where i is the unique identifier such that, for some $m \in \mathbf{N}^{+}$, $f(\mathtt{g}, n) = \langle \mathtt{g}\ m \rangle$ and $f(\mathtt{c}, m) = \langle \mathtt{c}\ \zeta(i) \rangle$.

(4) If $f \in \mathit{stores}$, $\tilde{o} \in \tilde{\mathcal{C}}_{\mathrm{lbc}}$, and $\mathit{comp}(\tilde{o}) = \langle \tilde{o}_1 \cdots \tilde{o}_m \rangle$, then $\mathit{extract}(f, \tilde{o}) =$

$$\mathit{make\text{-}flat\text{-}lit}(\tilde{o}, \langle \mathit{extract}(f, \tilde{o}_1) \cdots \mathit{extract}(f, \tilde{o}_m) \rangle).$$

(5) Otherwise, $\mathit{extract}(f, \tilde{o})\uparrow$.

4.3 Syntactic Correctness of the Linker

We show in this section that `link-template` always produces a proper LBC program from an FBC template. The principal result we prove is the following theorem:

Theorem 4.5 (Main Linker Correctness Theorem) *Let $f \in \mathit{stores}$ be represented by the literal table τ, and let $o \in \mathcal{L}_{\mathrm{fbc}}$. If* `link-literal` *produces $\langle \tau', o' \rangle$ from τ and o, then:*

(1) τ' is a literal table which represents a literal store f' extending f;

(2) $o' \in \mathcal{L}_{\mathrm{lbc}}$ and $\mathit{kind}(o') = \mathit{kind}(o)$;

(3) $\mathit{extract}(f', o') = o$.

Before proving this theorem, we present three elementary lemmas.

Lemma 4.6 `component-literals` *produces $\mathit{comp}(o)$ from $o \in \mathcal{L}_{\mathrm{fbc}}$.*

Proof `component-literals` is a direct implementation in Scheme of the definition of *comp* for members of $\mathcal{L}_{\mathrm{fbc}}$. □

The following lemma shows that `link-literals` simply iterates `link-literal` over a list of members of $\mathcal{L}_{\mathrm{fbc}}$.

Lemma 4.7 *Let τ be a literal table and $o_1, \ldots, o_m \in \mathcal{L}_{\mathrm{fbc}}$.*

(1) `link-literals` *produces $\langle \tau, \langle\rangle \rangle$ from τ and $\langle\rangle$.*

(2) Assume `link-literal` *produces $\langle \tau_1, o'_1 \rangle$ from τ and o_1, and* `link-literals` *produces $\langle \tau_m, \langle o'_2 \cdots o'_m \rangle \rangle$ from τ_1 and $\langle o_2 \cdots o_m \rangle$. Then* `link-literals` *produces $\langle \tau_m, \langle o'_1 \cdots o'_m \rangle \rangle$ from τ and $\langle o_1 \cdots o_m \rangle$.*

Proof By inspection of the Scheme code for `link-literals`.□

Lemma 4.8 *Suppose* $o \in \mathcal{L}_{\mathrm{fbc}}$ *and* $o_1, \ldots, o_m \in \mathcal{L}_{\mathrm{lbc}}$ *with* $m = \#comp(o)$. *Then* `make-linked-literal` *produces a pseudoliteral* $\tilde{o} \in \mathcal{A}_{\mathrm{fbc}} \cup \tilde{\mathcal{C}}_{\mathrm{lbc}}$ *from* o *and* $\langle o_1 \cdots o_m \rangle$ *such that:*

(1) $\tilde{o} = o$ *if* o *is atomic;*

(2) $\tilde{o} = \langle \texttt{constant}\ \langle \texttt{immutable-pair}\ n_1\ n_2 \rangle\rangle$ *if*
$o = \langle \texttt{constant}\ \langle \texttt{immutable-pair}\ c_1\ c_2 \rangle\rangle$ *and*
$o_i = \langle \texttt{constant}\ n_i \rangle$ *for* $i = 1, 2$*;*

(3) $\tilde{o} = \langle \texttt{constant}\ \langle \texttt{immutable-vector}\ n_1 \cdots n_m \rangle\rangle$ *if*
$o = \langle \texttt{constant}\ \langle \texttt{immutable-vector}\ c_1 \cdots c_m \rangle\rangle$ *and*
$o_i = \langle \texttt{constant}\ n_i \rangle$ *for* $i = 1, \ldots, m$*; and*

(4) $\tilde{o} = \langle \texttt{template}\ w\ \langle o_1 \cdots o_m \rangle\rangle$ *if* $o = \langle \texttt{template}\ w\ \langle o'_1 \cdots o'_m \rangle\rangle$.

Proof By inspection of the Scheme code for `make-linked-literal`.□

Proof of Main Linker Correctness Theorem The proof is by induction on *depth*(o).

Basis. Assume *depth*$(o) = 0$, i.e., $o \in \mathcal{A}_{\mathrm{fbc}}$. Also, assume `link-literal` produces $\langle \tau', o' \rangle$ from τ and o.

By the three lemmas immediately above, `lookup-in-literal-table` produces $\langle \tau', o' \rangle$ from τ and o. By the definition of a literal table, τ' is a literal table in virtue of τ being a literal table. Since $o \in \mathcal{A}_{\mathrm{fbc}}$, $lookup(f, o) = \langle f', o'' \rangle$, where $o'' = \langle k\ n \rangle \in \mathcal{L}_{\mathrm{lbc}}$. By Proposition 4.2, f' extends f, and $kind(o'') = kind(o)$, and by Lemma 4.3, τ' represents f' and $o' = o''$. Hence, statements (1) and (2) of the theorem hold. If $k = \mathtt{c}$, then

$$\begin{aligned} extract(f', o') &= extract(f', o'') \\ &= f'(\mathtt{c}, n)) \\ &= o \end{aligned}$$

by definitions of *extract* and *lookup*. The argument is the same if $k = \mathtt{t}$. If $o = \langle \mathtt{g}\ i \rangle$, then for some $m \in \mathbf{N}^+$ $f(\mathtt{g}, n) = \langle \mathtt{g}\ m \rangle$ such that $f(\mathtt{c}, m) = \langle \mathtt{c}\ \zeta(i) \rangle$, and so $extract(f', o') = extract(f', o'') = o$ by the definition of *extract*. Hence, statement (3) of the theorem holds.

Induction step. Assume $0 < n$ and the theorem is true for all $o \in \mathcal{L}_{\mathrm{fbc}}$ with $depth(o) < n$. Let $o \in \mathcal{C}_{\mathrm{fbc}}$ with $depth(o) = n$. It suffices to show the theorem is true for o.

Assume $(*)$ `link-literal` produces $\langle \tau', o' \rangle$ from τ and o. Consider `link-literal` acting on τ and o. `link-literal` has three local variables: `lits`, `tab-lits`, and `new-lit`. Since $\mathit{depth}(o) = n > 0$, $\mathit{comp}(o) = \langle o_1 \cdots o_m \rangle$ where $m \neq 0$ and $\mathit{depth}(o_i) < n$ for all i with $1 \leq i \leq m$. By Lemma 4.6, `lits` is bound to $\mathit{comp}(o)$. Hence, by Lemma 4.7, the induction hypothesis, and the assumption $(*)$, it follows that there are literal stores $f_1, \ldots, f_m$; literal tables $\tau_1, \ldots, \tau_m$; and $o'_1, \ldots, o'_m \in \mathcal{L}_{\mathrm{lbc}}$ such that:

(a) `tab-lits` is bound to $\langle \tau_m, \langle o'_1 \cdots o'_m \rangle \rangle$.

(b) τ_i represents f_i for all i with $1 \leq i \leq m$.

(c) f_1 extends f, and f_{i+1} extends f_i for all i with $1 \leq i \leq m - 1$.

(d) $\mathit{extract}(f_i, o'_i) = o_i$ for all i with $1 \leq i \leq m$.

By Lemma 4.8, `make-linked-literal` produces $\tilde{o} \in \tilde{\mathcal{C}}_{\mathrm{lbc}}$ from o and $\langle o'_1 \cdots o'_m \rangle$, and `new-lit` is bound to $\tilde{o}$. It is easy to see that $\mathit{kind}(\tilde{o}) = \mathit{kind}(o)$ and that each member of $\mathit{comp}(\tilde{o})$ is in the range of f_m.

By assumption $(*)$, `lookup-in-literal-table` produces $\langle \tau', o' \rangle$ from τ_m and `new-lit`. By the definition of a literal table, τ' is a literal table in virtue of τ_m being a literal table. Since each member of $\mathit{comp}(\tilde{o})$ is in the range of f_m, $\mathit{lookup}(f_m, \tilde{o}) = \langle f', o'' \rangle$, where $o'' = \langle k\ n \rangle \in \mathcal{L}_{\mathrm{lbc}}$ and $f'(k, n) = \tilde{o}$. By Proposition 4.2, f' extends f_m and $\mathit{kind}(o'') = \mathit{kind}(\tilde{o})$, and by Lemma 4.3, τ' represents f' and $o' = o''$. Hence, statements (1) and (2) of the theorem hold.

By the definition of *extract*,

$$\begin{aligned} \mathit{extract}(f', o') &= \mathit{extract}(f', o'') \\ &= \mathit{extract}(f', f'(k, n)) \\ &= \mathit{extract}(f', \tilde{o}) \end{aligned}$$

Furthermore, by the definitions of *extract* and *make-flat-lit*,

$$\begin{aligned} \mathit{extract}(f', \tilde{o}) &= \textit{make-flat-lit}(\tilde{o}, \langle \mathit{extract}(f', o'_1) \cdots \mathit{extract}(f', o'_m) \rangle) \\ &= \textit{make-flat-lit}(\tilde{o}, \langle o_1 \cdots o_m \rangle) \\ &= o \end{aligned}$$

Hence, statement (3) of the theorem holds.□

Corollary 4.9 `link-template` *produces a proper* LBC *program from an* FBC *template.*

Proof Let t be an FBC template. Let τ be the literal table with empty domain. Clearly, `link-literal` produces some $\langle \tau', o \rangle$ from τ and t. By Theorem 4.5,

(1) τ' is a literal table which represents a literal store f, and

(2) $o = \langle \mathtt{t}\ n \rangle$ for some $n \in \mathbf{N}^+$.

The procedure `assemble-linked-template` produces, from τ' and o, an LBC program

$$p = \langle n\ \mathtt{constants} :: c^*\ \mathtt{global\text{-}variables} :: n^* \rangle ^\frown t^+$$

such that:

(a) $f(\mathtt{c}, k) = \langle \mathtt{c}\ c^*(k-1) \rangle$ for all k with $1 \leq k \leq \#c^*$, and $f(\mathtt{c}, \#c^* + 1)\uparrow$.

(b) $f(\mathtt{g}, k) = \langle \mathtt{g}\ n^*(k-1) \rangle$ and $f(\mathtt{c}, n^*(k-1))$ is a symbol for all k with $1 \leq k \leq \#n^*$, and $f(\mathtt{g}, \#n^* + 1)\uparrow$.

(c) $f(\mathtt{t}, k) = t^+(k-1)$ for all k with $1 \leq k \leq \#t^+$, and $f(\mathtt{t}, \#t^+ + 1)\uparrow$.

(d) $n = \#t^+$.

From these three statements and the fact that f is a store, it follows that p is proper. $\Box$

4.4 Semantic Correctness of the Linker

We show in this section that an LBC program p produced by `link-template` from an FBC template t is semantically correct in the sense that the execution of p in the LBC state machine mirrors the execution of t in FBC state machine. Theorem 4.10 is a precise formulation of this assertion.

Before we can state Theorem 4.10, we must define a function Φ_ξ from FBC states to LBC states, which depends on a given FBC template ξ. Φ_ξ will be defined using a series of functions $\Phi_{\xi,o}$, $\Phi_{\xi,c}$, $\Phi_{\xi,t}$, $\Phi_{\xi,v}$, $\Phi_{\xi,a}$, $\Phi_{\xi,s}$, and $\Phi_{\xi,k}$ that map FBC literals, constants, templates, values, argument stacks, stores, and continuations to LBC literals, constants, templates, values, argument stacks, stores, and continuations, respectively. We shall also define a function Ψ_ξ which maps a function $\rho : i^k \to \mathtt{L}$ to a function $\rho' : n^k \to \mathtt{L}$ with $k \geq 0$.

Let ξ be a fixed FBC template. τ_ξ, f_ξ, and p_ξ are defined as follows: τ_ξ is the literal table which is produced by `link-literal` from the literal table with empty domain and ξ. f_ξ is the literal store represented by τ_ξ. p_ξ is the program produced by `link-template` from ξ. Note that p_ξ and τ_ξ are just different "representations" of f_ξ. Let $u_{\xi,c}$ and $u_{\xi,t}$ be the functions u_c and u_t for $p = p_\xi$.

$\Phi_{\xi,o}(o)$ is the literal (if there is one) produced by `link-literal` from τ_ξ and $o \in \mathcal{L}_{\text{fbc}}$; otherwise, $\Phi_{\xi,o}(o)$ is undefined.

$\Phi_{\xi,c}$ is defined by:

$$\Phi_{\xi,c}(c) = \begin{cases} u_c(n) & \text{if } \langle \texttt{c}\ n\rangle = \Phi_{\xi,o}(\langle \texttt{c}\ c\rangle) \\ \text{undefined} & \text{if } \Phi_{\xi,o}(\langle \texttt{c}\ c\rangle) \text{ is undefined} \end{cases}$$

$\Phi_{\xi,t}$ is defined by:

$$\Phi_{\xi,t}(t) = \begin{cases} u_t(n) & \text{if } \langle \texttt{t}\ n\rangle = \Phi_{\xi,o}(t) \\ \text{undefined} & \text{if } \Phi_{\xi,o}(t) \text{ is undefined} \end{cases}$$

$\Phi_{\xi,v}$ is defined by:

(1) If $v = \langle \texttt{immutable-pair}\ c_1\ c_2\rangle$, then

$$\Phi_{\xi,v}(v) = \langle \texttt{immutable-pair}\ \Phi_{\xi,c}(c_1)\ \Phi_{\xi,c}(c_2)\rangle,$$

provided $\Phi_{\xi,c}(c_1)\downarrow$ and $\Phi_{\xi,c}(c_2)\downarrow$; otherwise $\Phi_{\xi,v}(v)\uparrow$.

(2) If $v = \langle \texttt{immutable-vector}\ c_1\ \cdots c_m\rangle$, then

$$\Phi_{\xi,v}(v) = \langle \texttt{immutable-vector}\ \Phi_{\xi,c}(c_1)\ \cdots\ \Phi_{\xi,c}(c_m)\rangle,$$

provided $\Phi_{\xi,c}(c_1)\downarrow, \ldots, \Phi_{\xi,c}(c_m)\downarrow$; otherwise $\Phi_{\xi,v}(v)\uparrow$.

(3) If $v = \langle \textsc{closure}\ t\ u\ l\rangle$, then

$$\Phi_{\xi,v}(v) = \langle \textsc{closure}\ \Phi_{\xi,t}(t)\ u\ l\rangle,$$

provided $\Phi_{\xi,t}(t)\downarrow$; otherwise $\Phi_{\xi,v}(v)\uparrow$.

(4) If $v = \langle \textsc{escape}\ k\ l\rangle$, then

$$\Phi_{\xi,v}(v) = \langle \textsc{escape}\ \Phi_{\xi,k}(k)\ l\rangle,$$

provided $\Phi_{\xi,k}(k)\downarrow$; otherwise $\Phi_{\xi,v}(v)\uparrow$.

(5) Otherwise, $\Phi_{\xi,v}(v) = v$.

$\Phi_{\xi,a}$ is defined by: For $a = \langle v_1 \cdots v_m \rangle$,

$$\Phi_{\xi,a}(a) = \langle \Phi_{\xi,v}(v_1) \cdots \Phi_{\xi,v}(v_m) \rangle,$$

provided $\Phi_{\xi,v}(v_1)\downarrow, \ldots, \Phi_{\xi,v}(v_m)\downarrow$; otherwise $\Phi_{\xi,a}(a)\uparrow$.

$\Phi_{\xi,s} = \Phi_{\xi,a}$.

$\Phi_{\xi,k}$ is defined by:

(1) $\Phi_{\xi,k}(\text{HALT}) = \text{HALT}$.

(2) If $k = \langle \text{CONT}\ t\ n\ a\ u\ k' \rangle$, then

$$\Phi_{\xi,k}(k) = \langle \text{CONT}\ \Phi_{\xi,t}(t)\ n\ \Phi_{\xi,a}(a)\ u\ \Phi_{\xi,k}(k') \rangle,$$

provided $\Phi_{\xi,t}(t)\downarrow, \Phi_{\xi,a}(a)\downarrow, \Phi_{\xi,k}(k')\downarrow$; otherwise $\Phi_{\xi,k}(k)\uparrow$.

Φ_ξ is then defined by: For an FBC state $\Sigma = \langle t, n, v, a, u, k, s \rangle$,

$$\Phi_\xi(\Sigma) = \langle \Phi_{\xi,t}(t), n, \Phi_{\xi,v}(v), \Phi_{\xi,a}(a), u, \Phi_{\xi,k}(k), \Phi_{\xi,s}(s) \rangle.$$

Finally, Ψ_ξ is defined by: For a function $\rho : i^* \to \mathtt{L}$,

$$\Psi_\xi(\rho)(n) = \begin{cases} \rho(i) & \text{if } \mathit{extract}(f_\xi, \langle \mathtt{g}\ n \rangle) = \langle \mathtt{g}\ i \rangle, \text{ and } \rho(i)\downarrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

For each (pure) FBC rule R, let $\bar{R}$ be the corresponding pure LBC rule. Given an FBC state Σ, an FBC rule R, and $\rho : i^* \to \mathtt{L}$, $R(\Sigma, \rho)$ is the FBC state that results from applying R to Σ when $\mathit{globals} = \rho$. ($R(\Sigma, \rho) = \Sigma$ when R is not applicable to Σ given $\mathit{globals} = \rho$). Similarly, given an LBC state Σ', an LBC rule R', $\rho' : n^* \to \mathtt{L}$, and an LBC program p', $R'(\Sigma', \rho', p')$ is the LBC state that results from applying R' to Σ' when $\mathit{globals}' = \rho'$ and $p = p'$.

Theorem 4.10 *Let ξ be an* FBC *template, Σ be an* FBC *state, R be an* FBC *rule, and globals $= \rho$. Then*

$$\Phi_\xi(R(\Sigma, \rho)) = \bar{R}(\Phi_\xi(\Sigma), \Psi_\xi(\rho), p_\xi).$$

Proof The theorem clearly holds for all the FBC rules except possibly Literal, Closure, Global, and Set Global. The theorem holds for Literal since $\Phi_{\xi,o}(\langle \mathtt{c}\ c \rangle) = \langle \mathtt{c}\ n \rangle$ for some $n \in \mathbf{N}^+$ such that $\Phi_{\xi,c}(c) = u_c(n)$. The theorem holds for Closure since $\Phi_{\xi,o}(t) = \langle \mathtt{t}\ n \rangle$ for some $n \in \mathbf{N}^+$ such that $\Phi_{\xi,t}(t) = u_t(n)$. And, the theorem holds for Global and Set Global since $\Phi_{\xi,o}(\langle \mathtt{g}\ i \rangle) = \langle \mathtt{g}\ n \rangle$ for some $n \in \mathbf{N}^+$ such that $\mathit{globals}(i) = \mathit{globals}'(n)$. $\square$

References

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